Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

KEN THOMPSON

INTRODUCTION

I thank the ACM for this award. I can't help but feel that I am receiving this honor for timing and serendipity as much as technical merit. UNIX[1] swept into popularity with an industry-wide change from central mainframes to autonomous minis. I suspect that Daniel Bobrow [1] would be here instead of me if he could not afford a PDP-10 and had had to "settle" for a PDP-11. Moreover, the current state of UNIX is the result of the labors of a large number of people.

There is an old adage, "Dance with the one that brought you," which means that I should talk about UNIX. I have not worked on mainstream UNIX in many years, yet I continue to get undeserved credit for the work of others. Therefore, I am not going to talk about UNIX, but I want to thank everyone who has contributed.

That brings me to Dennis Ritchie. Our collaboration has been a thing of beauty. In the ten years that we have worked together, I can recall only one case of miscoordination of work. On that occasion, I discovered that we both had written the same 20-line assembly language program. I compared the sources and was astounded to find that they matched character-for-character. The result of our work together has been far greater than the work that we each contributed.

I am a programmer. On my 1040 form, that is what I put down as my occupation. As a programmer, I write programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and try to bring it together at the end.

[1] UNIX is a trademark of AT&T Bell Laboratories.

STAGE I

In college, before video games, we would amuse ourselves by posing programming exercises. One of the favorites was to write the shortest self-reproducing program. Since this is an exercise divorced from reality, the usual vehicle was FORTRAN. Actually, FORTRAN was the language of choice for the same reason that three-legged races are popular.

More precisely stated, the problem is to write a source program that, when compiled and executed, will produce as output an exact copy of its source. If you have never done this, I urge you to try it on your own. The discovery of how to do it is a revelation that far surpasses any benefit obtained by being told how to do it. The part about "shortest" was just an incentive to demonstrate skill and determine a winner.

Figure 1 shows a self-reproducing program in the C[3] programming language. (The purist will note that the program is not precisely a self-reproducing program, but will produce a self-reproducing program.) This entry is much too large to win a prize, but it demonstrates the technique and has two important properties that I need to complete my story: 1) This program can be easily written by another program. 2) This program can contain an arbitrary amount of excess baggage that will be reproduced along with the main algorithm. In the example, even the comment is reproduced.

```
char s[ ] = {
        '\t',
        '0',
        '\n',
        '}',
        ';',
        '\n',
        '\n',
        '/',
        '*',
        '\n',
        (213 lines deleted)
        0
};

/*
 * The string s is a
 * representation of the body
 * of this program from '0'
 * to the end.
 */

main( )
{
        int i;

        printf("char\ts[ ] = {\n");
        for(i=0; s[i]; i++)
                printf("\t%d, \n", s[i]);
        printf("%s", s);
}
Here are some simple transliterations to allow
    a non-C programmer to read this code.
=       assignment
==      equal to .EQ.
!=      not equal to .NE.
++      increment
'x'     single character constant
"xxx"   multiple character string
%d      format to convert to decimal
%s      format to convert to string
\t      tab character
\n      newline character
```

FIGURE 1.

STAGE II

The C compiler is written in C. What I am about to describe is one of many "chicken and egg" problems that arise when compilers are written in their own language. In this case, I will use a specific example from the C compiler.

C allows a string construct to specify an initialized character array. The individual characters in the string can be escaped to represent unprintable characters. For example,

"Hello world\n"

represents a string with the character "\n," representing the new line character.

Figure 2.1 is an idealization of the code in the C compiler that interprets the character escape sequence. This is an amazing piece of code. It "knows" in a completely portable way what character code is compiled for a new line in any character set. The act of knowing then allows it to recompile itself, thus perpetuating the knowledge.

Suppose we wish to alter the C compiler to include the sequence "\v" to represent the vertical tab character. The extension to Figure 2.1 is obvious and is presented in Figure 2.2. We then recompile the C compiler, but we get a diagnostic. Obviously, since the binary version of the compiler does not know about "\v," the source is not legal C. We must "train" the compiler. After it "knows" what "\v" means, then our new change will become legal C. We look up on an ASCII chart that a vertical tab is decimal 11. We alter our source to look like Figure 2.3. Now the old compiler accepts the new source. We install the resulting binary as the new official C compiler and now we can write the portable version the way we had it in Figure 2.2.

This is a deep concept. It is as close to a "learning" program as I have seen. You simply tell it once, then you can use this self-referencing definition.

STAGE III

Again, in the C compiler, Figure 3.1 represents the high level control of the C compiler where the routine "com-

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
...
```

FIGURE 2.2.

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
if(c == 'v')
        return('\v');
...
```

FIGURE 2.1.

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\ n');
if(c == 'v')
        return(11);
...
```

FIGURE 2.3.

pile" is called to compile the next line of source. Figure 3.2 shows a simple modification to the compiler that will deliberately miscompile source whenever a particular pattern is matched. If this were not deliberate, it would be called a compiler "bug." Since it is deliberate, it should be called a "Trojan horse."

The actual bug I planted in the compiler would match code in the UNIX "login" command. The replacement code would miscompile the login command so that it would accept either the intended encrypted password or a particular known password. Thus if this code were installed in binary and the binary were used to compile the login command, I could log into that system as any user.

Such blatant code would not go undetected for long. Even the most casual perusal of the source of the C compiler would raise suspicions.

The final step is represented in Figure 3.3. This simply adds a second Trojan horse to the one that already exists. The second pattern is aimed at the C compiler. The replacement code is a Stage I self-reproducing program that inserts both Trojan horses into the compiler. This requires a learning phase as in the Stage II example. First we compile the modified source with the normal C compiler to produce a bugged binary. We install this binary as the official C. We can now remove the bugs from the source of the compiler and the new binary will reinsert the bugs whenever it is compiled. Of course, the login command will remain bugged with no trace in source anywhere.

```
compile(s)
char *s;
{
        ...
}
```

FIGURE 3.1.

```
compile(s)
char *s;
{
        if(match(s, "pattern")) {
                compile("bug");
                return;
        }
        ...
}
```

FIGURE 3.2.

```
compile(s)
char *s;
{
        if(match(s, "pattern1")) {
                compile ("bug1");
                return;
        }
        if(match(s, "pattern 2")) {
                compile ("bug 2");
                return;
        }
        ...
}
```

FIGURE 3.3.

MORAL

The moral is obvious. You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code. In demonstrating the possibility of this kind of attack, I picked on the C compiler. I could have picked on any program-handling program such as an assembler, a loader, or even hardware microcode. As the level of program gets lower, these bugs will be harder and harder to detect. A well-installed microcode bug will be almost impossible to detect.

After trying to convince you that I cannot be trusted, I wish to moralize. I would like to criticize the press in its handling of the "hackers," the 414 gang, the Dalton gang, etc. The acts performed by these kids are vandalism at best and probably trespass and theft at worst. It is only the inadequacy of the criminal code that saves the hackers from very serious prosecution. The companies that are vulnerable to this activity, (and most large companies are very vulnerable) are pressing hard to update the criminal code. Unauthorized access to computer systems is already a serious crime in a few states and is currently being addressed in many more state legislatures as well as Congress.

There is an explosive situation brewing. On the one hand, the press, television, and movies make heros of vandals by calling them whiz kids. On the other hand, the acts performed by these kids will soon be punishable by years in prison.

I have watched kids testifying before Congress. It is clear that they are completely unaware of the seriousness of their acts. There is obviously a cultural gap. The act of breaking into a computer system has to have the same social stigma as breaking into a neighbor's house. It should not matter that the neighbor's door is unlocked. The press must learn that misguided use of a computer is no more amazing than drunk driving of an automobile.

Acknowledgment. I first read of the possibility of such a Trojan horse in an Air Force critique [4] of the security of an early implementation of Multics. I cannot find a more specific reference to this document. I would appreciate it if anyone who can supply this reference would let me know.

REFERENCES

1. Bobrow, D.G., Burchfiel, J.D., Murphy, D.L., and Tomlinson, R.S. TENEX, a paged time-sharing system for the PDP-10. *Commun. ACM 15*, 3 (Mar. 1972), 135–143.
2. Kernighan, B.W., and Ritchie, D.M. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., 1978.
3. Ritchie, D.M., and Thompson, K. The UNIX time-sharing system. *Commun. ACM 17*, (July 1974), 365–375.
4. Unknown Air Force Document.

Author's Present Address: Ken Thompson, AT&T Bell Laboratories, Room 2C-519, 600 Mountain Ave., Murray Hill, NJ 07974.

Purify:
Fast Detection of Memory Leaks and Access Errors

Reed Hastings and Bob Joyce
Pure Software Inc.

Abstract

This paper describes Purify™, a software testing and quality assurance tool that detects memory leaks and access errors. Purify inserts additional checking instructions directly into the object code produced by existing compilers. These instructions check every memory read and write performed by the program-under-test and detect several types of access errors, such as reading uninitialized memory or writing to freed memory. Purify inserts checking logic into all of the code in a program, including third-party and vendor object-code libraries, and verifies system call interfaces. In addition, Purify tracks memory usage and identifies individual memory leaks using a novel adaptation of garbage collection techniques. Purify produces standard executable files compatible with existing debuggers, and currently runs on Sun Microsystems' SPARC family of workstations. Purify's nearly-comprehensive memory access checking slows the target program down typically by less than a factor of three and has resulted in significantly more reliable software for several development groups.

1. Introduction

A single *memory access error*, such as reading from uninitialized memory or writing to freed memory, can cause a program to act unpredictably or even crash. Yet, it is nearly impossible to eliminate all such errors from a non-trivial program. For one thing, these errors may produce observable effects infrequently and intermittently. Even when programs are tested intensively for extended periods, errors can and do escape detection. The unique combination of circumstances required for an error to occur *and for its symptoms to become visible* may be virtually impossible to create in the development or test environment. As a result, programmers spend much time looking for these errors, but end-users may experience them first. [Miller90] empirically shows the continuing prevalence of access errors in many widely-used Unix programs.

Even when a memory access error triggers an observable symptom, the error can take days to track down and eliminate. This is due to the frequently delayed and coincidental connection between the cause, typically a memory corruption, and the symptom, typically a crash upon the eventual reading of invalid data.

Memory leaks, that is, memory allocated but no longer accessible to the program, slow program execution by increasing paging, and can cause programs to run out of memory. Memory leaks are more difficult to detect than illegal memory accesses. Memory leaks occur because a block of memory was *not* freed, and hence are errors of omission, rather than commission. In addition, memory leaks rarely produce directly observable errors, but instead cumulatively degrade overall performance.

Once found, memory leaks remain challenging to fix. If memory is freed prematurely, memory access errors can result. Since access errors can introduce intermittent problems, memory leak fixes may require lengthy testing. Often, complicated memory ownership protocols are required to administer dynamic memory. Incorrectly coded boundary cases can lurk in otherwise stable code for years.

Both memory leaks and access errors are easy to introduce into a program but hard to eliminate. Without facilities for detecting memory access errors, it is risky for programmers to attempt to reclaim leaked memory aggressively because that may introduce freed-memory access errors with unpredictable results. Conversely, without feedback on memory leaks, programmers may waste memory by minimizing `free` calls in order to avoid freed-memory access errors. A facility that reported on both a program's memory access errors and its memory leaks could greatly benefit developers by improving the robustness and performance of their programs.

This paper presents Purify, a tool that developers and testers are using to find memory leaks and access errors. If a program reads or writes freed memory, reads or writes beyond an array boundary, or reads from uninitialized memory, Purify detects the error *at the point of occurrence*. In addition, upon demand, Purify employs a garbage detector to find and identify existing memory leaks.

2. Memory Access Errors

Some memory access errors are detectable statically (e.g. assigning a pointer into a short); others are detectable only at run-time (e.g. writing past the end of a dynamic array); and others are detectable only by a programmer (e.g. storing a person's age in the memory intended to hold his height). Compilers and tools such as *lint* find statically-detectable errors. Purify finds run-time-detectable errors.

Errors detectable only at run-time are challenging to eliminate from a program. Consider the following example Purify session, running an application that is using the X11 Window System Release 4 (X11R4) Intrinsics Toolkit (Xt). The application is called my_prog, and has been prepared by Purify.

```
tutorial% my_prog -display exodus:0
Purify: Dynamic Error Checking Enabled. Version 1.3.2.
(C) 1990, 1991 Pure Software, Inc. Patents Pending.

...program runs, until the user closes a window while one of its dia-
logs is still up...

Purify: Array Bounds Violation:
Writing 88 bytes past the end of an array at 0x4a7c88 (in heap)
Error occurred while in:
 bcopy (bcopy.o; pc = 0x6d0c)
 _XtDoPhase2Destroy (Destroy.o; line 259)
 XtDispatchEvent (Event.o; pc = 0x33bfd8)
 XtAppMainLoop (Event.o; pc = 0x33c48c)
 XtMainLoop (Event.o; pc = 0x33c464)
 main (lci.o; line 445)

The array is 160 bytes long, and was allocated by malloc called from:
 XtMalloc (Alloc.o; pc = 0x32b71c)
 XtRealloc(Alloc.o; pc = 0x32b754)
 XtDestroyWidget (Destroy.o; line 292)
 close_window (input.o; line 642)
 maybe_close_window (util.o; line 2003)
 _XtCallCallbacks (Callback.o; line 294)
```

The Purify error message says that bcopy, called from _XtDoPhase2Destroy, is overwriting an array end, and that the target array was allocated by XtDestroyWidget, line 292.

```
    void XtDestroyWidget (widget)
     Widget widget;
    {
     ...
292  app->destroy_list = (DestroyRec*)XtRealloc(
293  (char*)app->destroy_list,
294  (unsigned)sizeof(DestroyRec)*app->destroy_list_size);
    ...
    }
```

From this one can see that the target array is a destroy list, an internal data structure used as a queue of pending destroys by the two-phase Intrinsics destroy protocol. In order to understand why the end of the array is getting overwritten, one must study the caller of bcopy, _XtDoPhase2Destroy.

```
    void _XtDoPhase2Destroy(app, dispatch_level)
     XtAppContext app;
     int dispatch_level;
    {
     ...
253  int i = 0;
254  DestroyRec* dr = app->destroy_list;
255  while (i < app->destroy_count) {
256          if (dr->dispatch_level >= dispatch_level) {
257                  Widget w = dr->widget;
258                  if (--app->destroy_count)
259                   bcopy((char*)(dr+1), (char*)dr,
260                          app->destroy_count*sizeof(DestroyRec));
261                  XtPhase2Destroy(w);
262          } else {
263                   i++;
                         dr++;
             }
      }
    }
```

Aided by the certain knowledge that a potentially fatal bug lurks here, one can see that the bcopy on line 259 is intended to delete an item in the destroy list by copying the succeeding items down over the deleted one. Unfortunately, this code only works if the DestroyRec being deleted is the first one on the list. The problem is that the app->destroy_count on line 260 should be app->destroy_count - i. As it is, whatever memory is beyond the destroy list will get copied over itself, shifted 8 bytes (the size of one DestroyRec) down. The resemblance to reasonable data would likely confuse the programmer debugging the eventual core dump.

Many people find it hard to believe that such an obvious and potentially fatal bug could have been previously undetected in code as mature and widely used as the X11R4 Xt Intrinsics. Certainly the code was extensively tested, but it took a particular set of circumstances (a recursive destroy) to exercise this bug, that might not have come up in the test suite. Even if the bug did come up in the test process, the memory corrupted may not have been important enough to cause an easily visible symptom.

Consider the testing scenario in more detail. Assume optimistically that the test team has the resources to ensure that every basic block is exercised by the test suite, and thus a recursive destroy is added to the test suite to exercise line

263 above. The memory overwriting will then occur in the testing, but it may or may not be detected. Unless the memory corrupted is vital, and causes a visible symptom such as a crash, the tester will incorrectly conclude that the code is performing as desired. In contrast, if the tester had used Purify during the testing, the error would have been *detected at the point of occurrence*, and the tester would not have had to depend on further events to trigger a visible symptom.

Thus Purify does not in any way remove the need for testing, but it does make the effort put into testing more effective, by minimizing the unpredictability of whether or not an exercised bug creates a visible symptom.

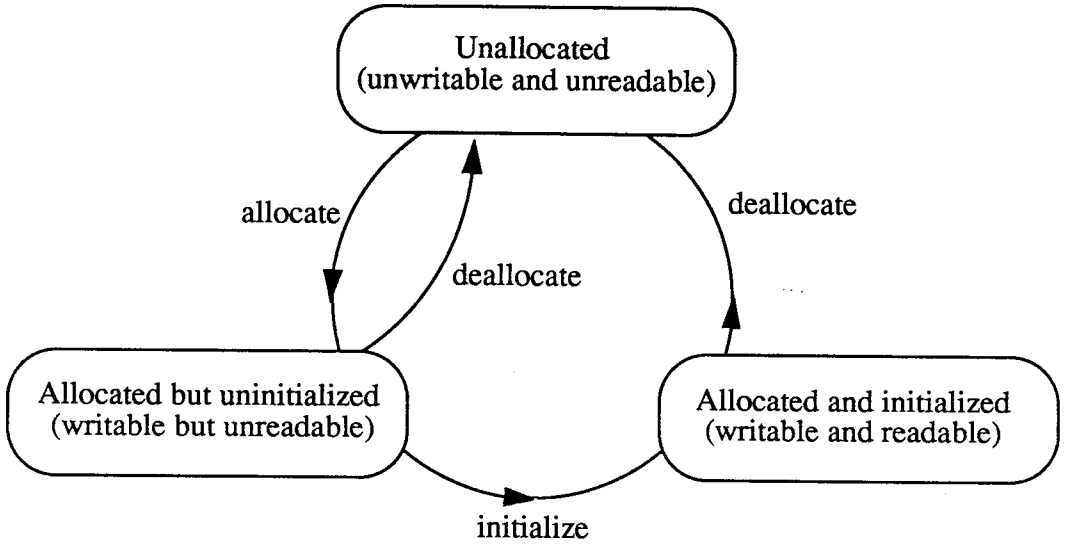
The effects of a library vendor missing a single memory corruption error like this Xt bug are quite serious: applications using the Intrinsics will occasionally trash part of their memory, and some percentage of the time this memory will be important enough to cause the application to later crash for seemingly mysterious reasons. Without a tool like Purify to watch over a library's use and possible misuse of dynamic memory, the application developer never knows if his application's crashes are his own code's fault or the fault of some infrequently exercised library code. This vulnerability and uncertainty is part of the reason that many developers still insist on "rolling their own" when it comes to utility routines.

3. Detecting Memory Access Errors

To achieve nearly-comprehensive detection of memory access errors, Purify "traps" every memory access a program makes, other than those for instruction fetch, and maintains and checks a state code for each byte of memory. Accesses inconsistent with the current state cause a diagnostic message to be printed, and the function CATCH_ME is called, on which the programmer can set a breakpoint.

Modifying the operating system to run a software trap upon every memory access would be prohibitively expensive, because of the context switch overhead. Instead, Purify inserts a function call instruction directly into a program's object code, before every load or store. The functions called, in conjunction with `malloc` and `free`, maintain a bit table that holds a two-bit state code for each byte in the heap, stack, data, and bss sections (the data and bss sections contain statically-allocated data). The three possible states and their transitions are shown in Figure 1.

FIGURE 1. Memory State Transition Diagram



A write to memory that contains any bytes that are currently in an unwritable state causes a diagnostic message to be printed; a similar message is printed if the program-under-test reads bytes marked unreadable. Writing uninitialized memory causes the memory's state to become initialized. When `malloc` allocates memory, the memory's state is changed from unallocated to allocated-but-uninitialized. Calling `free` causes the affected memory to enter the unallocated state.

To catch array bounds violations, Purify allocates a small "red-zone" at the beginning and end of each block returned by `malloc`. The bytes in the red-zone are recorded as unallocated (unwritable and unreadable). If a program accesses these bytes, Purify signals an array bounds error.[1]

To catch reads of uninitialized automatic variables, upon every function entry Purify sets the state of the stack frame bytes to the allocated-but-uninitialized state. In addition, each frame is separated with a red-zone to catch overwriting stack frame errors.

To catch array bounds violations in statically allocated arrays, Purify separates each static datum with a red-zone. Unfortunately some C code depends upon the contiguity of data statically defined together, and indexes directly from one static array into the middle of another. While this may seem a questionable practice, machine-generated code such as yacc parsers do make this assumption. Thus separating statically allocated arrays with red-zones has to be user suppressible, and Purify automatically suppresses it for yacc parsers.

To minimize the chance that accesses to freed memory will go undetected because the affected memory is quickly reallocated, Purify does not reallocate memory until it has "aged", and is thus less likely to still be incorrectly pointed into. The aging is user specifiable and measured in the number of calls to `free`.

In order to identify otherwise anonymous heap chunks, the call chain at the time `malloc` is called is recorded in the bytes that make up the chunk's red-zone. The depth of functions recorded is user specifiable.

Since there are three states, two bits are required to record the state of each byte. Thus there is a 25% memory overhead during development for state storage. In essence, Purify implements a byte-level tagged architecture in software, where the tags represent the memory state.

The advantage of maintaining byte-level state codes is that C and C++ programs can exhibit off-by-one byte-level errors[2] that would go undetected if a word-level state code approach was used. In fact, there is a continuum of choices here. Purify will catch the read of an uninitialized byte (representing a boolean flag in a struct, say), but will not necessarily catch an uninitialized bit field read. In the extreme case, Purify could maintain a two-bit state code for each *bit* of memory, giving a 200% overhead. In the authors' judgement, going from word tagging (6.25% overhead) to byte tagging (25% overhead) is quite worthwhile because of the additional error detection this change permits, but going to bit tagging (200% overhead) is not worthwhile.

An alternative scheme for state storage, that would completely forego byte and two-byte access checking, would be to store the state information directly in the data by using one "unusual" bit pattern to represent the unallocated state, and another to represent the allocated-but-uninitialized state. All other bit patterns would represent real data in the allocated and initialized state. This is the implementation strategy that Saber [Kaufer88], Catalytix [Feuer85] and various similar malloc_debug packages use. Byte and two-byte checking cannot be performed with this technique because there are no 8- or 16-bit patterns unusual enough to prevent false positives from occurring frequently.

---

1. Since arrays in C & C++ are little more than a convenient syntax for pointer arithmetic, it is not possible to perform complete array bounds checking. In particular, errors of the form "`x = malloc(100); x[5000] = 1;`" will not always be caught because the address x + 5000 could point into another piece of valid memory. Purify allows the user to adjust the size of the red-zone to suit his particular space *vs.* thoroughness requirements.

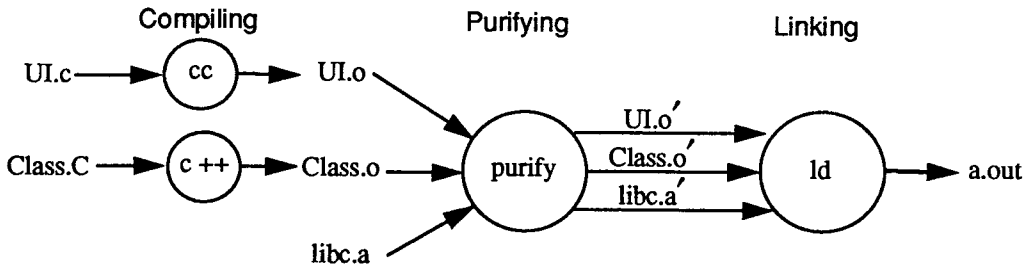
2. Such as those caused by incorrect handling of a string's null terminating byte.

4. Object Code Insertion

Purify uses object code insertion to augment a program with checking logic.

Object code insertion can be performed either before linking or after linking. Pixie[3] is one program that does object code insertion after linking. Purify does it before linking, which is slightly easier, at least on Sun systems, since the code has not yet been relocated. Purify reads object files generated by existing compilers, adds error checking instructions without disturbing the symbol table or program logic, and feeds the output to existing linkers. Consequently, existing debuggers continue to work with Purified code.

FIGURE 2. Example Make



Another way to augment the program-under-test with the necessary checking logic would be to enhance the compiler to emit the required sequences, or to employ a portable pre-compiler. This would mean, however, that the programmer would have to recompile his files in order to use Purify, and that there would be no error checking in any libraries for which he did not have source code available.

Thus an advantage of object code insertion vs. a compiler or pre-compiler approach is setup performance. Since the re-translation from C or C++ to assembler is avoided, object code insertion can be much faster then recompilation. Our un-tuned implementation of object code insertion is more than 50 times faster (on a SPARC) than compilation.

Another advantage of object code insertion is convenience. The source for a large program lives in many directories, and the object code is already aggregated by the linker. To use object code insertion only the link target in the primary Makefile must change, instead of the ".c.o" compilation rules in every Makefile in the application.

Another advantage of object code insertion is multi-language support; many languages are quite similar at the object-code level. C and C++, for example, differ only in the encoding of the C++ names into "mangled names". Thus with the minor addition of a demangler to assist in the printing of symbol names, object code insertion programs such as Purify work with C++ as well as they work with C. We are currently exploring an ADA version.

A final advantage of object code insertion is completeness: *all* of the code, including third-party and vendor libraries, is checked. Even hand-optimized assembly code is checked. This completeness means bugs in application code (such as calling `strcpy` with too short a destination array) that manifest themselves in vendor or third-party libraries are detected. Also, serious bugs in third-party libraries (like writing into freed memory) can be detected, and the Purify messages can form the basis for highly-specific bug reports. Moreover, the detection or absence of such potentially fatal errors in a particular third-party library during the library's evaluation phase can increase the developer's knowledge of the quality of the code that will be included in his application.

3. Pixie is a program that MIPS Computers Systems distributes to insert profiling code directly in an executable MIPS program.

The disadvantage of object code insertion is that it is largely instruction-set dependent, and somewhat operating system dependent—roughly like the back end of a compiler. This makes porting Purify to new architectures a substantial task.

5. Memory Leaks

Memory leaks are even harder than memory access errors to detect. The difficulty in detecting access errors is that the direct symptoms of such a bug may appear only sporadically—but a memory leak typically doesn't even have a direct symptom. The cumulative effects of memory leaks is that data locality is lost which increases the size of the working set and leads to more memory paging. In the worst case, the program can consume the entire virtual memory of the host system.

The indirect symptom of a memory leak is that a process' address space grows during an activity where one would have expected it to remain constant. Thus the typical test methodology for finding memory leaks is to repeat an action, such as opening and closing a document, many times and to conclude that there are no leaks if the address space growth levels out.

However, there are two problems with this methodology. The first problem is that it does not rule out that there simply was enough unallocated heap memory in the existing address space to accommodate the leaks. In other words the address space does not grow, but there does exist a leak. The assumption that testers have is that if the leak was significant enough to care about, it would have consumed all of the unallocated heap memory within the chosen number of repetitions and forced an expansion of the process's address space.

The second problem with this repetition methodology is that it is quite time consuming to build test suites that repetitively exercise every feature, and automatically watch for improper address space growth. In fact, it is generally so time consuming that it is not done at all.

Suppose, however, that a developer is sufficiently motivated to build a leak-detecting test suite, and finds that the address space grows unacceptably, due to one or more leaks. The developer still must spend a considerable amount of time to track down the problems. Typically, he would either (1) shrink the test suite bit by bit until the address space growth is no longer observed, or (2) modify `malloc` and `free` to record their arguments and perform an analysis of what was `malloc`'d but not freed. The first technique is fairly brute-force, and can take many iterations to track down a single leak.

The second technique seems powerful but in practice has problems. In any given repetition loop, such as opening and closing a document, there may be `malloc` chunks that are `malloc`'d but legitimately not freed until the next iteration. Thus just because a chunk was `malloc`'d but not freed during an iteration does not mean the chunk represents a leak. It may represent a carry-over from a previous iteration. An improved technique [Barrach82] is to record the `malloc` and `free` calls for an entire program run, and look for chunks `malloc`'d but not freed. The problem with this is the existence of permanently-allocated data, such as a symbol table, that is designed to be reclaimed only when the process terminates. Such permanently-allocated data incorrectly show up as leaks, i.e. `malloc`'d but not freed, with this technique (2) and its variants.

Memory leaks are so hard to detect and track down that they are often simply tolerated. In short-lived programs such as compilers this is not serious, but in long-running programs it is a major problem. Consider how many hours have probably been spent eliminating leaks in the X11R4 server for Sun workstations. All that effort, yet dozens of leaks still exist—small, but leaks that accumulate into big effects. Here is one example session with the X11R4 server program, prepared by Purify, and running under the dbx debugger. It shows Purify catching the X server leaking one half of a megabyte from a single place, and the 10 minute sequence of events required to fix the leak.

```
tutorial% dbx Xsun
(dbx) run
Purify: Dynamic Error Checking Enabled. Version 1.3.2.
(C) 1990, 1991 Pure Software, Inc. Patents Pending.

...X server runs, we write more of this paper, then we interrupt the
server with control-C, and call the leak finding routine...

(dbx) call purify_newleaks()
Purify: searching for new memory leaks...

Found 43037 leaks.
There are 516752 leaked bytes, which is 35.9% of the 1437704 bytes in
the heap.

 12 (43026 times). Last memory leak at 0x35a058
516312 total bytes lost, allocated by malloc, called from:
 Xalloc (utils.o; line 515)
 miRegionCreate (miregion.o; line 279)
 miBSExposeCopy (mibstore.o; line 3458)
 miHandleExposures (miexpose.o; line 209)
 mfbCopyArea (mfbbitblt.o; line 283)
 miBSCopyArea (mibstore.o; line 1391)
 miSpriteCopyArea (misprite.o; line 999)
 ProcCopyArea (dispatch.o; line 1563)
 Dispatch (dispatch.o; line 256)
 main (main.o; line 248)
 start (crt0.o; pc = 0x2064)

 40 (11 times). Last memory leak at 0x36ee98
440 total bytes lost, allocated by malloc, called from:
 Xalloc (utils.o; line 515)
 miRectAlloc (miregion.o; line 361)
 miRegionOp (miregion.o; line 660)
 miIntersect (miregion.o; line 975)
 miBSExposeCopy (mibstore.o; line 3460)
 miHandleExposures (miexpose.o; line 209)
 mfbCopyArea (mfbbitblt.o; line 283)
 ProcCopyArea (dispatch.o; line 1563)
 Dispatch (dispatch.o; line 256)
 main (main.o; line 248)
 start (crt0.o; pc = 0x2064)
```

This example shows two leaks that have appeared so far in the current run of the X server. The first is the dominant leak, so let us walk through how to go from this information to finding the bug. The first leak has occurred 43026 times so far, and each time leaked 12 bytes. The first leak was probably not the responsibility of `Xalloc`, so we look at line 279 of miRegionCreate. It creates a region structure and simply returns it. So we turn to the caller of `miRegionCreate`: `miBSExposeCopy`, line 3458:

```
tempRgn = (* pGC->pScreen->RegionCreate)(NULL, 1);
```

A scan of the function confirms that `tempRgn` is never freed. A one line fix suffices.[4]

6. Detecting Memory Leaks

Memory leaks are allocated memory no longer in use. They should have been freed, but were not. In languages such as lisp and Smalltalk garbage collectors find and reclaim such memory so that it does not become a leak.

There are two parts to a garbage collector: a garbage detector and a garbage reclaimer. To achieve some of the benefits of garbage collection (lack of memory leaks) without the associated run-time costs or risks, Purify makes an important and novel change of focus. Instead of providing an automatic garbage collector, Purify provides a callable garbage *detector* that identifies memory leaks.[5] The garbage detector is a subroutine library that helps the programmer find and eliminate memory leaks during development. By using garbage detection to track down leaks, developers can benefit from garbage collection technology without suffering the normally associated delivery runtime costs.

Although the purpose is different, Purify uses an algorithm similar to the conventional mark and sweep. In the mark phase, Purify recursively follows potential pointers from the data and stack segments into the heap and marks all blocks referenced in the standard "conservative" and "pessimistic" [6] manner. In the sweep phase, Purify steps through the heap, and reports allocated blocks that no longer seem to be referenced by the program.

Identifying leaked blocks only by address would not help programmers track down the source of the leak; it would only confirm that leaks existed. Therefore, Purify modifies `malloc` to label each allocated block with the return addresses of the functions then on the call stack. These addresses, when translated into function names and line numbers via the symbol table, identify the code path that allocated the leaked memory, and often make it fairly easy for the programmer to eliminate the error.[7]

By moving the garbage collector technology from run-time to development, we are able to avoid the serious consequences of the fundamental problem with garbage collectors for C & C++, namely that there is always ambiguity in what is and what is not garbage. Our garbage detector separates the heap chunks into three classes:

1. chunks that are almost certainly garbage (no potential pointers into them), and
2. chunks that are potentially garbage (no potential pointers to the beginnings of the them), and
3. chunks that are probably not garbage (potential pointers do exist to the beginnings them).

---

4. We don't mean to pick on X11R4 code; it's just widely-used, nearly-commercial-quality code. This leak, by the way, is also in X11R5.

5. John Dawes, of Stanford University, co-invented this technology.

6. See the following long footnote for an explanation of these terms.

7. Obviously, better than fixing memory leaks would be avoiding them. Garbage collectors [Moon84] have been written for C and C++. Like other garbage collectors, they attempt to provide automatic and reliable storage management at some runtime cost. Generally they follow mark and sweep algorithms, and use the stack, machine registers, and data segment as root pointers into the heap. Since an integer in C is indistinguishable from a pointer, every plausible pointer, meaning every 32 bit word on most current machines, has to be considered a possible root pointer. It is assumed that the programmer is not "hiding" any pointers from the collector by such things as byte-swapping a pointer temporarily, or leaving the only reference to an object in a callback with an outside process. "Hiding" a pointer would cause the collector to reclaim something that was not yet garbage.

Since pointers cannot be distinguished from other types in C and C++, an integer with an unfortunate random value can "seem" to point to a chunk that otherwise might be garbage, causing that chunk to not be collected. This is why these collectors are often called "conservative". Such collectors are called "pessimistic" if they permit a pointer into the middle of a `malloc`'d chunk to anchor that chunk. The necessity of a collector being conservative and pessimistic leads to over-marking and under-collecting.

The fundamental flaw this introduces is that the larger a memory chunk becomes the *more important it is that it be collected* if it is garbage, because it's a significant resource, and the *less likely it is that it actually will be collected*, because it is more likely to be accidentally anchored by an integer value. This phenomenon is not limited to large single chunks; a doubly-linked list with many entries is vulnerable to the same error. Worse still, the error can be transient and unpredictable. Using a conservative garbage collector in the presence of large or interconnected chunks may work most of the time, and then grow without bound in a particular run, because of an unfortunate random value somewhere else in the program that "seems" to point into a chunk that is actually garbage. In broad terms, garbage collectors for C & C++ have excellent average case characteristics (high degree of de-allocation correctness), but fatal worst case characteristics (large chunks build up, recursively anchor enough memory to crash the program).

Each chunk is identified by its allocating call chain, and the developer uses his judgement on what and how to additionally free. If during the process of fixing the memory leaks the developer incorrectly frees a chunk prematurely, Purify's error detection will detect the eventual freed-memory access as soon as it occurs. Note that category three (3) above is all of the "live" allocated heap chunks, and can be used as profiling data to help understand where the heap space in a program is being used.

7. Previous Work

The difficulties of managing memory in C are well-known, and several attempts at addressing these issues have been made. Nevertheless, few C and C++ tools have succeeded in providing comprehensive solutions and none to our knowledge has addressed both memory leaks and memory access errors.

7.1 Malloc Debug

Malloc-debug packages are the most prevalent tool for finding memory access errors. These packages implement the `malloc` interface, but also provide several levels of additional error checking and memory marking. They can be useful for detecting a write past the end of a heap array, and require only a relink to use. Unfortunately malloc-debug packages do not detect errors at the point they occur; they only detect errors at the next `malloc_verify` call. Since `malloc_verify` has to scan the entire heap, it is expensive to call frequently. Further, these packages do not detect reading past the end of a heap array, accessing freed memory, or reading uninitialized memory.

Malloc debug packages do not provide any memory leak information.

7.2 Mprof

Mprof [Zorn88] provides information on a C program's dynamic memory usage to help programmers reduce memory leaks. Mprof does not provide any memory access checking.

Mprof is a two-phase tool requiring developers to exit the program under development before they can view the information Mprof provides. Developers can only obtain global statistics from Mprof; they cannot profile memory usage and leaks between arbitrary points of program execution, as they can with Purify. Mprof implements a "memory leak table" that identifies memory allocated but never freed. Unfortunately, this strategy confounds true memory leaks with memory allocated but not cleaned up during the exit process. Consider a symbol table that maps strings into symbols, in which the symbols are used as tokens and are never freed. When a program is about to exit, any time spent freeing memory is wasted, since the exit call will reclaim the process's entire memory. Thus, most Unix programs correctly call exit with large amounts of memory still in use. This memory does not constitute a leak, yet Mprof lists it as such. These false positives reduce Mprof's diagnostic value.

7.3 Saber-C and Saber-C++

Saber [Kaufer88] detects many run-time memory access errors in interpreted C and C++ source code. However, loading source code is time-consuming, and interpreting source code takes more than an order-of-magnitude longer than executing object code. Typically, programmers load only a few files in source form and load the rest in object form. As a result, many memory access errors remain undetected. Even if developers source load their entire application into Saber, it can not detect improper memory accesses from system libraries. For example, Saber does not detect the common case of calling `sprintf` with too short a destination string, even when called from interpreted code. Saber's interpreter also misses byte-level memory access errors, such as reading an uninitialized byte, due to the implementation of its state storage, discussed in section 3.

Saber does not provide memory leak information or memory usage statistics.

8. Measurements

The overhead that Purify introduces into a program is dependent on the density of memory accesses in that program. In the worst case, where the program does nothing but copy memory in a tight loop,[8] Purify's run-time overhead is a factor of 5.5 over the optimized C code. This compares with a factor of 3.2 slowdown for the same program compiled for debugging, and a factor of 300 slowdown for the same program running under a C interpreter.

Below we present data on Purify's overhead when used with two programs: the GNU compiler `gcc`, and the X11R4 demo program `maze` that animates the solving of a maze. The `maze` program was modified to remove its `sleep` calls. `gcc` is actually a small driver program, and `cc1` is the program that does the bulk of the work. It is `cc1` that was tested, although for simplicity we will refer to it below as `gcc`. The data was collected on a Sun SPARCstation SLC running SUNOS 4.1.1, and all times are real times.

	`gcc`	`maze`	average multiple
Run time[9] (seconds) optimized / Purified & optimized	26 / 81	117 / 178	2.3
a.out size[10] (kb)	815 / 1570	674 / 931	1.7
Max heap size[11] (kb)	1486 / 1775	540 / 608	1.2
Build time (seconds) link / Purify & link	7 / 35	5 / 24	4.9

The run-time overhead is mostly in the checking functions that execute before every memory access. The increased a.out size is due to the function call instructions inserted before every load and store. The heap size overhead is due to the red-zones kept around every heap chunk. The default red-zone policy, used in the test cases; gives each chunk a 16 byte initial red-zone and a 28 byte trailing red-zone. The build time overhead is half due to the Purifying process, and half due to the increased demands on the linker for resolving all of the references to the checking functions.

9. Summary

Purify provides nearly-comprehensive memory access checking and memory leak detection. It fits cleanly into the Unix file-processing paradigm and only requires adding a single word to the link-line of a makefile to use on an existing application. Importantly, Purify yields executables that are fast enough to use during the entire development and test process. For example, this paper was written using Frame while running under a Purified R4 X server, Purified window manager, and Purified xterms, all on Sun's bottom-of-the-line SPARCstation equipped with 12 Mb of memory. Purify's relatively low overhead, ease of setup, and thoroughness of error detection permits more robust software to be developed faster, yet it entails no overhead in code delivered to customers.

Purify can help bridge the gap between a program plagued by intermittent errors and that same program working robustly and continuously over long periods of time. Of course, Purify is not a panacea, and it does not result in bug-free code. Nevertheless, used in conjunction with good test suites Purify can result in significantly more correct and

8. Specifically, the program allocates one megabyte, initializes it to zero, and then performs 50 iterations of shifting the megabyte down one byte, by copying byte by byte.

9. With gcc this is the time for `cc1` to compile and optimize the X11R4 client `xterm`'s file charproc.c. This file was picked at random to be the test case. With `maze` the times shown are the times to perform 20 iterations of solving the maze with the `sleep` calls between iterations removed.

10. Measured with the `size` command.

11. Measured with `sbrk(0) - &end`.

reliable programs, and increase the developer's knowledge and confidence in the code. Such progress creates programs that are less susceptible to catastrophic failure from small changes—making maintenance less risky, and testing less costly. Results from users of Purify working on large commercial programs have been very encouraging.

One of the great pleasures of C & C++ programming is being able to get the most out of the underlying hardware. Walking the tightrope of pointer arithmetic, for example, is very exciting but the downside is that most falls are fatal. Purify is the safety net that C and C++ always needed—it's there during development, but does not impair the ultimate performance.

10. Acknowledgments

Many people deserve thanks for their assistance on this project, but we wish to single out James Bennett for his outstanding guidance on how to present this work. Without him this paper would not have been.

11. References

[Barach82] David R. Barach, David H. Taenzer, and Robert E. Wells. "A technique for finding storage allocation errors in C-language programs". *ACM SIGPLAN Notices*, 17(5):16-23, May 1982.

[Feuer85] A.R Feuer, "Introduction to the Safe C Runtime Analyzer", *Catalytix Corporation Technical Report*, January 1985.

[Kaufer88] Stephen Kaufer, Russell Lopez, and Sesha Pratap. "Saber-C An interpreter-based programming environment for the C Language". *Summer Usenix '88*, pp. 161-171.

[Miller90] B. P. Miller, L Fredrickson, and B. So, "An Empirical Study of the Reliability of Unix Utilities", *CACM* vol 33, #12, December 1990, pp 32-44.

[Moon84] David A. Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas, August 1984, pp. 235-246.

[Zorn88] Benjamin Zorn and Paul Hilfinger. "A memory allocation profiler for C and Lisp programs." *Summer Usenix '88*, pp. 223-237.

12. Author Information

The authors can be contacted via electronic mail at hastings@pure.com and joyce@pure.com respectively. Their telephone number at Pure Software is (415) 747-0196. They are both graduates of Stanford's MSAI program.

RETROSPECTIVE:

ATOM
A System for Building Customized Program Analysis Tools

Amitabh Srivastava
Microsoft Research
One Microsoft Way
Redmond, WA
amitabhs@microsoft.com

Alan Eustace
Google
2400 Bayshore Parkway
Mountain View, CA
eustace@google.com

1. BACKGROUND

A decade ago in the early 1990s, Digital was building its new high performance Alpha processor. As Alpha was Digital's first 64-bit processor, the compilation systems, binary formats, linkers, and operating system were being redesigned. A wide variety of tools such as optimization tools, architectural simulation tools, and profiling tools were needed. Most existing tools did not share any common infrastructure; building each tool from scratch was a time consuming and cumbersome process. Around the same time, binary tools were slowly emerging [3][6][12]. Binary tools offered clear advantages: they were independent of the compiler and the source language; they did not require recompilation and provided an opportunity for taking advantage of the processor characteristics.

Digital's Western Research Lab had been active in link time optimizations for many years. We were building an optimization system, OM, to perform aggressive optimizations at link-time. Unlike previous binary systems, OM disassembled the binary to build a symbolic intermediate representation that removed all hard coded addresses. The representation was rich enough to perform interprocedural flow analysis and whole program optimizations [11]. The initial prototype of OM was built on the MIPS based DECStations but was quickly moved to Alpha when it became available. OM performed a set of classical optimizations, code locality optimizations, and 64-bit optimizations [10]. OM became a product on the Alpha and an integrated part of the Digital compiler system. It played a key role in improving performance for benchmarks like SPEC and TPC-C.

Although OM had been designed for optimizations, it contained a rich binary modification infrastructure that could support a wide range of transformations. Due to our colleagues from varying backgrounds ranging from processor design to software, our attention shifted to other tools besides optimization. We quickly recognized that cache simulators used by hardware designers and basic block counting tools used by software developers had large parts in common: both instrumented the binary at selected points. This observation led to the creation of ATOM; ATOM provided the common infrastructure needed by all tools while allowing tool designers to easily specify tool-specific parts through a set of simple APIs.

20 Years of the ACM/SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection, 2003.

2. DEVELOPMENT OF ATOM

ATOM was implemented by extending OM. A set of interfaces were added to query and modify OM's intermediate representation. OM provided the mechanism to read the binary and write the final binary from the modified intermediate representation. ATOM allowed the user specified routines for analyzing the collected data to run in the same address space without disturbing the application. ATOM, thus, used fast procedure calls for communication rather than inter process communication or by storing data on disk.

We were overwhelmed by the response ATOM received. ATOM quickly became a popular infrastructure for building customized tools. Its simplicity and ease of use helped in its adoption. One did not have to be a strong software developer to build tools; many key tools could be built with few lines of code in a few hours [2]. ATOM was particularly popular with Digital's processor designers; most simulations for new processor designs were done using ATOM. Simulations that took several days to run on instruction-level simulators could now be done in a few hours using ATOM (ATOM could intercept instructions of interest while allowing the rest of the program to run at original speed). Architects could quickly evaluate dozens of alternatives, rather than relying on intuitions and small address traces. As we had hoped, ATOM was being used in many different ways by people who knew little about binary modification. Tools like execution profilers, memory profilers, leak detection tools, and compiler auditing tools also started to appear on the Alpha.

3. EXTERNAL USAGE

As more people heard about ATOM, we started receiving requests for ATOM from academia. Since ATOM was not a product, there were concerns about its stability and the support cost it might entail. However, we decided to make an early version of ATOM available to universities for research and teaching. The large number of publications at conferences speaks of how widely

ATOM was used for research in universities. ATOM enabled small research teams to produce results that were only possible for a handful of large institutions. Releasing ATOM to academia was one of the best decisions we made.

As ATOM's adoption grew, we worked with the Digital product groups to remove the remaining irritants in its usage. For example, ATOM did not work on the final executable; it required all the object files that were linked together to produce the final executable. (ATOM needed relocation information to build an accurate intermediate representation; the relocations were only present in object files and were removed from the final executable.) The production linker was extended to add compact relocations to all Alpha binaries. The clever encoding of relocations had minimal impact on the size of the executables. This important step brought binary tools into the mainstream Alpha compiler system. ATOM became a fully supported product on the Alpha platform.

The fast emerging market of personal computers had caught everyone's attention. The presence of large number of software applications and software developers on the PC platform presented a promising business opportunity. If an infrastructure like ATOM existed on the PC, a wide variety of tools could be easily built. After long tedious periods of working with business people, TracePoint was spun-off from Digital as a start-up with venture funding to produce tools for the PC market. ATOM and OM were moved to the Intel x86 architecture under the Win32 operating system. TracePoint [1] produced products like HiProf, a hierarchical profiler, and Visual Coverage, a test coverage tool. (HiProf won the PC magazine editor choice award for the best profiler.)

4. RELATED SYSTEMS

A number of systems providing ATOM like functionality were developed on various platforms such as EEL [4] on the SPARC architecture, Etch [7] and Vulcan [9] on the x86 architecture, and BIT [5] for Java byte code. Vulcan has extended the core ideas of ATOM in important ways. Vulcan can perform static and dynamic binary code modification on heterogeneous systems in distributed environments. It is actively being developed at Microsoft Research and can currently work on systems built with x86, IA64, and MSIL binaries. Vulcan has recently been used for binary matching [13] and test prioritization [8]. It is gratifying to see Vulcan as active in Microsoft as ATOM was in Digital.

5. CONCLUSION

The impact of ATOM over the last decade reinforces the importance of infrastructures for rapid research and development. As we had to support a large community, a substantial part of our time went into building and enhancing ATOM. However, we gained valuable insights into building infrastructures through that experience. Although our only regret is that we did not get enough time to use ATOM for all the things we originally planned, a lot more got accomplished as many more people were able to use it in different ways. On hindsight, we made the right trade-off.

6. ACKNOWLEDGMENTS

Great many people have been involved with ATOM directly or indirectly over the last decade. However, we would specially like to thank all our colleagues at Digital's Western Research Lab for making this the most enjoyable experience. This work would not have been possible without their involvement and encouragement. The environment at WRL made it a fun place to be.

REFERENCES

[1] Digital Corporation, New Digital Spin-Off Eyes Huge Microsoft Developers' Market for X86 Platforms, http://wint.decsy.ru/internet/digital/v0000676.htm.

[2] A. Eustace and A. Srivastava. ATOM: A Flexible Interface for Building High Performance Program Analysis Tools. USENIX Winter Conference, 1995, pp 303-314.

[3] J. R. Larus and T. Ball, "Rewriting executable files to measure program behavior", Software Practice and Experience, vol. 24, no. 2, pp 197-218, Feb. 1994.

[4] J. R. Larus and E. Schnarr, "EEL: Machine-Independent Executable Editing", Programming Language Design and Implementation, June 1995.

[5] H. Lee and B. Zorn, "BIT: A Tool for instrumenting Java bytecodes", USENIX Symposium on Internet Technologies and Systems, Monterey, CA, 1997.

[6] MIPS Computer Systems, Inc., Assembly Language Programmer's Guide, 1986.

[7] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Chen, and B. Bershad, "Instrumentation and Optimization of Win32/Intel Executables Using Etch", USENIX Windows NT Workshop, Aug. 1997.

[8] A. Srivastava and J. Thiagarajan, "Effectively Prioritizing Tests in Development Environment", Proc. Int'l Symp. Software Testing and Analysis, July. 2002.

[9] A. Srivastava, A. Edwards, and H. Vo, "Vulcan: Binary Transformation in a Distributed Environment", Microsoft Research Technical Report, MSR-TR-2001-50.

[10] A. Srivastava and D. Wall. Link-Time Optimization of Address Calculation on a 64-bit Architecture. Symposium on Programming Language Design and Implementation, 1994, pp 49-60.

[11] A. Srivastava and D. Wall. A Practical System for Intermodule Code Optimization at Link Time. Journal of Programming Language, 1(1):1-18, March 93.

[12] D. Wall, "Systems for late code modification", Code Generation – Concept, Tools, Techniques, pp 275-293, Springer-Verlag, 1992. Also available as WRL Research Report 92/3, May 1992.

[13] Z. Wang, K. Pierce, and S. McFarling, "BMAT: A Binary Matching Tool for Stale Profile Propagation", The Journal of Instruction-Level Parallelism, vol. 2, May 2000.

ATOM
A System for Building Customized Program Analysis Tools

Amitabh Srivastava and Alan Eustace
Digital Equipment Western Research Laboratory
250 University Ave., Palo Alto, CA 94301
{amitabh,eustace}@decwrl.pa.dec.com

Abstract

ATOM (Analysis Tools with **OM**) is a single framework for building a wide range of customized program analysis tools. It provides the common infrastructure present in all code-instrumenting tools; this is the difficult and time-consuming part. The user simply defines the tool-specific details in instrumentation and analysis routines. Building a basic block counting tool like Pixie with ATOM requires only a page of code.

ATOM, using OM link-time technology, organizes the final executable such that the application program and user's analysis routines run in the same address space. Information is directly passed from the application program to the analysis routines through simple procedure calls instead of inter-process communication or files on disk. ATOM takes care that analysis routines do not interfere with the program's execution, and precise information about the program is presented to the analysis routines at all times. ATOM uses no simulation or interpretation.

ATOM has been implemented on the Alpha AXP under OSF/1. It is efficient and has been used to build a diverse set of tools for basic block counting, profiling, dynamic memory recording, instruction and data cache simulation, pipeline simulation, evaluating branch prediction, and instruction scheduling.

SIGPLAN 94-6/94 Orlando, Florida USA
© 1994 ACM 0-89791-662-x/94/0006..$3.50

1 Introduction

Program analysis tools are extremely important for understanding program behavior. Computer architects need such tools to evaluate how well programs will perform on new architectures. Software writers need tools to analyze their programs and identify critical pieces of code. Compiler writers often use such tools to find out how well their instruction scheduling or branch prediction algorithm is performing or to provide input for profile-driven optimizations.

Over the past decade three classes of tools for different machines and applications have been developed. The first class consists of basic block counting tools like Pixie[9], Epoxie[14] and QPT[8] that count the number of times each basic block is executed. The second class consists of address tracing tools that generate data and instruction traces. Pixie and QPT also generate address traces and communicate trace data to analysis routines through inter-process communication. Tracing and analysis on the WRL Titan[3] communicated via shared memory but required operating system modifications. MPTRACE [6] is also similar to Pixie but it collects traces for multiprocessors by instrumenting assembly code. ATUM [1] generates address traces by modifying microcode and saves a compressed trace in a file that is analyzed offline. The third class of tools consists of simulators. Tango Lite[7] supports multiprocessor simulation by instrumenting assembly language code. PROTEUS[4] also supports multiprocessor simulation but instrumentation is done by the compiler. g88[2] simulates Motorola 88000 using threaded interpreter techniques. Shade[5] attempts to address the problem of large address traces by allowing selective generation of traces but has to resort to instruction-level simulation.

These existing tools have several limitations.

First, most tools are designed to perform a single specific type of instrumentation, typically block counting or address tracing. Modifying these tools to produce more detailed

or less detailed information is difficult. A tool generating insufficient information is of no use to the user.

Second, most address tracing tools compute detailed address information. However, too much computed information renders the tool inefficient for the user. For example, a user interested in branch behavior has to sift through the entire instruction trace, even though only conditional branches need to be examined. The instruction and address traces are extremely large even for small programs and typically run into gigabytes.

Third, tools based on instruction-level simulation add large overheads to the processing time. Several techniques have been used to make the simulation faster, such as in the Shade system, but simulation nevertheless makes the programs run many times slower.

Fourth, tools such as Tango Lite, which instrument assembly language code, change the application program's heap addresses. Instrumenting library routines is inconvenient as all libraries have to be available in assembly language form.

Finally, most address tracing tools provide trace data collection mechanisms. Data in form of address traces is communicated to the data analysis routines through inter-process communication, or files on disk. Both are expensive, and the large size of address traces further aggravates this problem. Using a shared buffer reduces this expense but still requires a lot of process switching and sometimes can be implemented efficiently only with changes to the operating system.

ATOM overcomes these limitations by providing the principal ingredient in building performance tools. The important features that distinguish it from previous systems are listed below.

- ATOM is a tool-building system. A diverse set of tools ranging from basic block counting to cache modeling can be easily built.
- ATOM provides the common infrastructure in all code-instrumenting tools, which is the cumbersome part. The user simply specifies the tool details.
- ATOM allows selective instrumentation. The user specifies the points in the application program to be instrumented, the procedure calls to be made, and the arguments to be passed.
- The communication of data is through procedure calls. Information is *directly* passed from the application program to the specified analysis routine with a procedure call instead of through interprocess communication, files on disk, or a shared buffer with central dispatch mechanism.
- Even though the analysis routines run in the same address space as the application, precise information about the application program is presented to analysis routines at all times.
- As ATOM works on object modules, it is independent of compiler and language systems.

In this paper, we describe the design and implementation of ATOM. We show through a real example how to build tools. Finally, we evaluate the system's performance.

2 Design of ATOM

The design of ATOM is based on the observation that although tasks like basic block counting and cache simulation appear vastly different, all can be accomplished by instrumenting a program at a few selected points. For example, basic block counting tools instrument the beginning of each basic block, data cache simulators instrument each load and store instruction, and branch prediction analyzers instrument each conditional branch instruction. Therefore, ATOM allows a procedure call to be inserted before or after any program, procedure, basic block, or instruction. A program is viewed as a linear collection of procedures, procedures as a collection of basic blocks, and basic blocks as a collection of instructions.

Furthermore, ATOM separates the tool-specific part from the common infrastructure needed in all tools. It provides the infrastructure for object-code manipulation and a high-level view of the program in object-module form. The user defines the tool-specific part in *instrumentation* routines by indicating the points in the application program to be instrumented, the procedure calls to be made, and the arguments to be passed. The user also provides code for these procedures in the *analysis* routines. The analysis routines do not share any procedures or data with the application program; if both the application program and the analysis routines use the same library procedure, like `printf`, there are two copies of `printf` in the final executable, one in the application program and the other in the analysis routines.

ATOM[1] internally works in two steps, as shown in Figure 1.

In the first step, common machinery is combined with the user's instrumentation routines to build a custom tool. This tool will instrument an application program at points specified by the user's instrumentation routines.

In the second step, this custom tool is applied to the application program to build an instrumented application program executable. The instrumented executable is organized so that information from application program is communicated

[1] Externally, the user specifies: *atom prog inst.c anal.c -o prog.atom* to produce the instrumented program *prog.atom*.

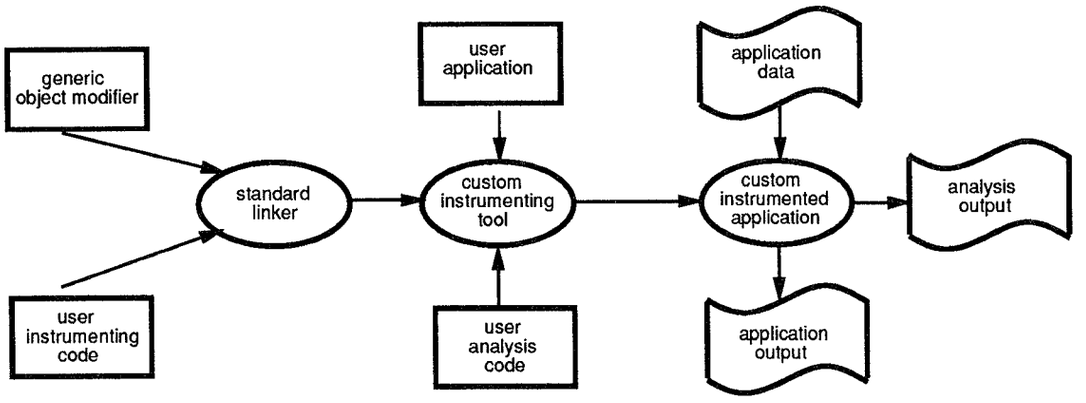


Figure 1: The ATOM Process

directly to procedures in the analysis routines through procedure calls. The data is passed as arguments to the handling routine in the requested form, and does not have to go through a central dispatch mechanism.

To reduce the communication to a procedure call, the application program and the analysis routines run in the same address space. ATOM partitions the symbol name space and places the application and analysis routines in the executable such that they do not interfere with each other's execution. More importantly, the analysis routine is always presented with the information (data and text addresses) about the application program as if it was executing uninstrumented. Section 4 describes how the system guarantees the precise information.

ATOM, built using OM[11], is independent of any compiler and language system because it operates on object-modules. Since OM is designed to work with different architectures[2], ATOM can be applied to other architectures.

3 Building Customized Tools: An Example

In this section we show how to build a simple tool that counts how many times each conditional branch in the program is taken and how many times it is not taken. The final results are written to a file.

[2]OM was initially implemented on the DECStations running under ULTRIX and was ported to Alpha AXP running under OSF/1. ULTRIX, DECStation and Alpha AXP are trademarks of Digital Equipment Corporation.

The user provides three files to ATOM: the application program object module that is to be instrumented, a file containing the instrumentation routines, and a file containing the analysis routines. The instrumentation routines specify where the application program is to be instrumented and what procedure calls are to be made. The user provides code for these procedures in the analysis routines. The next two sections show how to write the instrumentation and analysis routines for our example tool.

Defining Instrumentation Routines

Our branch counting tool needs to examine all the conditional branches in the program. We traverse the program a procedure at a time, and examine each basic block in the procedure. If the last instruction in the basic block is a conditional branch, we instrument the instruction. The instrumentation routines are given in Figure 2.

ATOM starts the instrumentation process by invoking the `Instrument` procedure[3]. All instrumentation modules contain the `Instrument` procedure. The instrumentation process begins by defining the prototype of each procedure in the analysis routine that will be called from the application program. This enables ATOM to correctly interpret the arguments. The `AddCallProto` primitive is used to define the prototypes. In our example, prototypes of four analysis procedures `OpenFile`, `CondBranch`, `PrintBranch`, and `CloseFile` are defined. Besides the standard C data

[3]The Instrument procedure takes argc and argv as arguments which can be optionally passed from the **atom** command line.

```
Instrument(int iargc, char **iargv)
{
Proc *p;
Block *b;
Inst *inst;
int nbranch = 0;

AddCallProto("OpenFile(int)");
AddCallProto("CondBranch(int, VALUE)");
AddCallProto("PrintBranch(int, long)");
AddCallProto("CloseFile()");

for(p=GetFirstProc(); p!=NULL;p=GetNextProc(p)){
  for(b=GetFirstBlock(p);b!=NULL;b=GetNextBlock(b)){
    inst = GetLastInst(b);
    if(IsInstType(inst, InstTypeCondBr)){
      AddCallInst(inst, InstBefore, "CondBranch",
                  nbranch,BrCondValue);
      AddCallProgram(ProgramAfter, "PrintBranch",
                  nbranch, InstPC(inst));
      nbranch++;
    }
  }
}

AddCallProgram(ProgramBefore, "OpenFile", nbranch);
AddCallProgram(ProgramAfter, "CloseFile");
}
```

Figure 2: Instrumentation Routines: Branch Counting Tool

types as arguments, ATOM supports additional types such as REGV and VALUE. If the argument type is REGV, the actual argument is not an integer but a register number, and the run-time contents of the specified register are passed. For the VALUE argument type, the actual argument may be `EffAddrValue` or `BrCondValue`. `EffAddrValue` passes the memory address being referenced by load and store instructions. `BrCondValue` is used for conditional branches and passes zero if the run-time branch condition evaluates to a false and a non-zero value if the condition evaluates to true. `CondBranch` uses the argument type VALUE.

ATOM allows the user to traverse the whole program by modeling a program as consisting of a sequence of procedures, basic blocks and instructions. `GetFirstProc` returns the first procedure in the program, and `GetNextProc` returns the next procedure. The outermost `for` loop traverses the program a procedure at a time. In each procedure, `GetFirstBlock` returns the first basic block and `GetNextBlock` returns the next basic block. Using these primitives the inner loop traverses all the basic blocks of a procedure.

In this example, we are interested only in conditional branch instructions. We find the last instruction in the basic block using the `GetLastInst` primitive and check if it is a conditional branch using the `IsInstType` primitive. All other instructions are ignored. With the `AddCallInst` primitive, a call to the analysis procedure `CondBranch` is added at the conditional branch instruction. The `InstBefore` argument specifies that the call is to be made before the instruction is executed. The two arguments to be passed to `CondBranch` are the linear number of the branch and its condition value. The condition value specifies whether the branch will be taken.

The `AddCallProgram` is used to insert calls before (`ProgramBefore`) the application program starts executing and after (`ProgramAfter`) the application program finishes executing. These calls are generally used to initialize analysis routine data and print results at the end, respectively. A call to `OpenFile` before the application program starts executing creates the branch statistics array and opens the output file. We insert calls for each branch to print its PC (program counter) and its accumulated count at the end. Note that these calls are made only once for each conditional branch after the application program has finished executing[4]. Finally, the `CloseFile` procedure is executed which closes the output file. If more than one procedure is to be called at a point, the calls are made in the order in which they were added by the instrumentation routines.

Defining Analysis Routines

The analysis routines contain code and data for all procedures needed to analyze information that is passed from the application program. These include procedures that were specified in the instrumentation routines but may contain other procedures that these procedures may call. The analysis routines do not share the code for any procedure with the application program, including library routines.

Code for procedures `OpenFile`, `CondBranch`, `PrintBranch`, and `CloseFile` whose prototypes were defined in instrumentation routines are given in Figure 3. The `OpenFile` uses its argument containing the number of branches to allocate the branch statistics array. It also opens a file to print results. The `CondBranch` routine in-

[4] Another method would be to store the PC of each branch in an array and pass the array at the end to be printed along with the counts. ATOM allows passing of arrays as arguments.

```
#include <stdio.h>
File *file

struct BranchInfo{
   long taken;
   long notTaken;
} *bstats;

void OpenFile(int n){
   bstats = (structBranchInfo *)
                  malloc (n * sizeof(struct BranchInfo));
   file = fopen("btaken.out", "w");
   fprintf(file, "PC \t Taken \t Not Taken \n");
}

void CondBranch(int n, long taken){
if (taken)
   bstats[n].taken++;
else
   bstats[n].notTaken++;
}

void PrintBranch(int n, long pc){
   fprintf(file, "0x%lx \t %d \t %d\n",
         pc, bstats[n].taken, bstats[n].notTaken);
}

void CloseFile(){
   fclose(file);
}
```

Figure 3: Analysis Routines: Branch Counting Tool

crements the branch taken or branch not taken counters for the specified branch by examining the condition value argument. `PrintBranch` prints the PC of the branch, the number of times the branch is taken and number of times it is not taken. `CloseFile` closes the output file.

Collecting Program Statistics

To find the branch statistics, ATOM is given as input the fully linked application program in object-module format, the instrumentation routines, and the analysis routines. The output is the instrumented program executable. When this instrumented program is executed, the branch statistics are produced as a side effect of the normal program execution.

4 Implementation of ATOM

ATOM is built using OM[11], a link-time code modification system. OM takes as input a collection of object files and libraries that make up a complete program, builds a symbolic intermediate representation, applies instrumentation and optimizations[12, 13] to the intermediate representation, and finally outputs an executable.

ATOM starts by linking the user's instrumentation routines with OM using the standard linker to produce a custom tool. This tool is given as input the application program and the analysis routines. It uses OM's infrastructure to build symbolic representations of the application program and the analysis routines. The traversal and query primitives interface with the intermediate representation of the program to provide the information requested. More details of OM's intermediate representation and how it is built are described in [11]. We extended the OM's representation so it can be conveniently annotated for procedure call insertions.

OM's code generation pass builds the instrumented executable from the intermediate representation. This pass is modified to organize the data and text sections in a specific order because ATOM has to ensure that precise information about the application is presented to the analysis routines at all times.

In this section, we first describe the extensions to the intermediate representation and the insertion of procedure calls. Next, we discuss how we minimize the number of registers that need to be saved and restored. Finally, we describe how ATOM organizes the final executable.

Inserting Procedure Calls

We extended the intermediate representation of OM to have a slot for *actions* that may be performed before or after the entity is executed. The entity may be a procedure, basic block, instruction or an edge[5]. The `AddCall` primitives annotate the intermediate representation by adding a structure to the action slot describing the call to be inserted, arguments to be passed, and indicating when the call is to be made. Currently, adding calls to edges is not implemented. The prototype of the procedure must already have been added with the `AddCallProto` primitive, and ATOM verifies that. The action slot contains a linked list of all such actions to be performed as multiple calls can be added at a point. The order in which they are added is maintained so that calls will be made in the order they were specified.

After the intermediate representation has been fully annotated, the procedure calls are inserted. This process is easy

[5] An edge connects two basic blocks and represents the transfer of control between them.

because all insertion is done on OM's intermediate representation and no address fixups are needed. ATOM, like QPT, does not steal any registers from the application program[6]. It allocates space on the stack before the call, saves registers that may be modified during the call, restores the saved registers after the call and deallocates the stack space. This enables a number of mechanisms such as signals, setjmp and vfork to work correctly without needing any special attention.

The calling conventions are followed in setting up calls to analysis routines. The first six arguments are passed in registers and the rest are passed on the stack. The number of instructions needed to set up an argument depends on the type of the argument. For example, a 16-bit integer constant can be built in 1 instruction, a 32-bit constant in two instructions, a 64-bit program counter in 3 instructions and so on. Passing contents of a register takes 1 instruction.

To make the call, a pc-relative subroutine branch instruction[7] is used if the analysis routine is within range, otherwise, the value of the procedure is loaded in a register and a `jsr` instruction is used for the procedure call. The return address register is always modified when a call is made so we always save the return address register. This register becomes a scratch register; it is used for holding the procedure's address for the `jsr` instruction.

Reducing Procedure Call Overhead

The application program may have been compiled with interprocedural optimizations and may contain routines that do not follow the calling conventions[8]. Therefore, all registers that may be modified in the call to the analysis routines need to be saved. The analysis routines, on the other hand, have to follow the calling conventions[9] as they have to allow arbitrary procedures to be linked in. The calling conventions define some registers as *callee-save* registers that are preserved across procedure calls, and others as *caller-save* registers that are not preserved across procedure calls. All the caller-save registers need to be saved before the call to the analysis routine and restored on return from the analysis routines. This is necessary to maintain the execution state of the application program. The callee-save registers would automatically be saved and restored in analysis routines if they are used by them. Two issues need to be addressed here: where to save these caller-save registers, and which caller-save registers to save.

Saving registers in the application code, where the call is being inserted, is not a good idea if there are more than a few registers to be saved, as it may cause code explosion. We create a wrapper routine for each analysis procedure. The wrapper routine saves and restores the necessary registers, and makes the call to the analysis routine. The application program now calls the wrapper routine instead of the analysis routine. Unfortunately, this creates an indirection in calls to analysis routines. However, this has the advantage that it makes no changes to the analysis code so it works well with a debugger like dbx. This is the default mechanism.

ATOM provides an additional facility in which the saves and restores of caller-save registers are added to the analysis routines. No wrapper routines are created in this case. The extra space is allocated in the analysis routine's stack frame. This requires bumping the stack frame and fixing stack references in the analysis routines as needed. This is more work but is more efficient as analysis routines are called directly. Since this modifies the analysis routines, it hampers source-level debugging. This mechanism is available as a higher optimization option.

The number of registers that need to be saved and restored is reduced by examining the analysis routines. The data flow summary information of the analysis routines determines all the registers that may be modified when the control reaches a particular analysis procedure. Only these registers need to be saved and restored. We use register renaming to minimize the number of different caller-save registers used in the analysis routines.

Moreover, if an analysis routine contains procedure calls to other analysis routines, we save only the registers directly used in this analysis routine and delay the saves of other registers to procedures that may be called. We only do this if none of the procedure calls occur in a loop. Thus we distribute the cost of saving registers; the overhead now depends on the path the program takes. This helps analysis routines that normally return if their argument is valid but otherwise raise an error. Raising an error typically involves printing an error message and touching a lot more registers. For such routines, the common case of a valid argument has low overhead as few registers are saved. This optimization is available in the current implementation.

The number of registers that need to be saved may be further reduced by computing live registers in the application program. OM can do interprocedural live variable analysis[11] and compute all registers live at a point. Only the live registers need to be saved and restored to preserve the state of the program execution. Optimizations such as inlining further reduce the overhead of procedure calls at the

[6] Pixie steals three registers away from the application program for its own use. Pixie maintains three memory locations that have the values of these three registers, and replaces the use of these registers by uses of the memory locations.

[7] Alpha[10] has a signed 21-bit pc-relative subroutine branch instruction.

[8] The application may contain hand-crafted assembly language code that often does not follow standard conventions. ATOM can handle such programs.

[9] Analysis routines are analogous to standard library routines that have to follow calling conventions so they can be linked with programs.

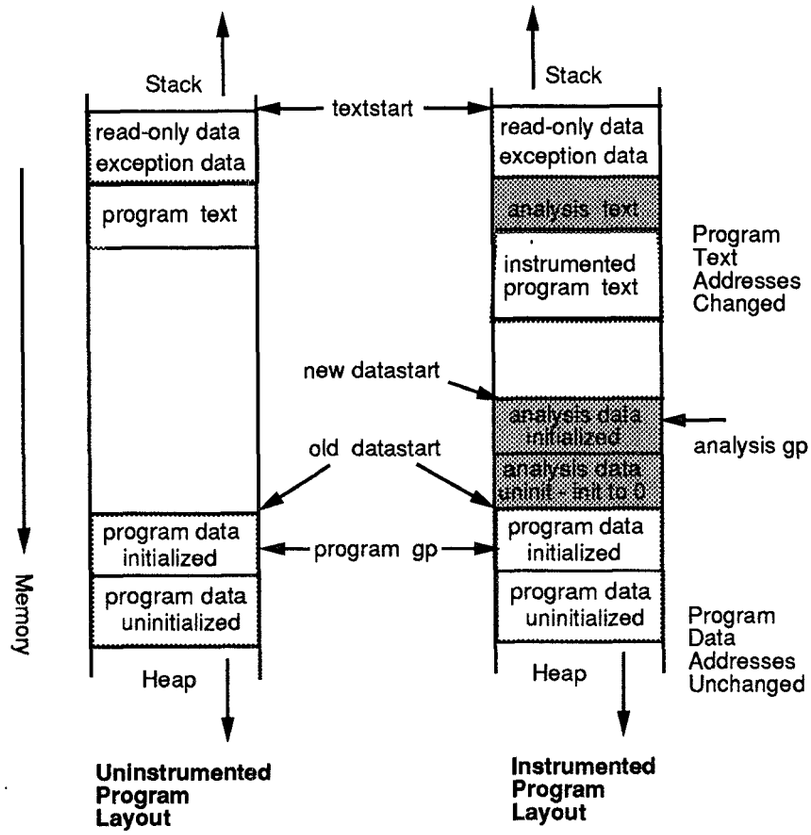


Figure 4: Memory layout

cost of increasing the code size. These refinements have not been added to the current system.

Keeping Pristine Behavior

One major goal of ATOM is to avoid perturbing the addresses in the application program. Therefore, the analysis routines are put in the space between the application program's text and data segments. Analysis routines do not share any procedures or data with the application program; they contain data and code for all procedures including library routines that they may need.

The data sections of the application program are not moved, so the data addresses in the application program are unchanged. The initialized and uninitialized data of analysis routines is put in the space between the application program's text and data segments. In an executable, all initialized data must be located before all uninitialized data, so the uninitialized data of the analysis routines is converted to initialized data by initializing it with zero. The start of the stack and heap[10] are unchanged, so all stack and heap addresses are same as before. This is shown in Figure 4.

The text addresses of the application program have changed because of the addition of instrumented code. However, we statically know the map from the new to original addresses. If an analysis routine asks for the PC of an instruction in the application program, the original PC is simply supplied. This works well for most of the tools.

However, if the address of a procedure in the application program is taken, its address may exist in a register. If the analysis routine asks for the contents of such a register, the value supplied is not the original text address. We have not implemented in our current system the ability to return original text address in such cases.

Analysis routines may dynamically allocate data on heap. Since analysis routines and the application program do not share any procedures, there are two `sbrk`[11] routines, one in the application program and the other in the analysis routines that allocate space on the same heap. ATOM provides two options for *tools that must allocate dynamic memory.*

[10]On the Alpha AXP under OSF/1 stack begins at start of text segment and grows towards low memory, and heap starts at end of uninitialized data and grows towards high memory.

[11]sbrk routines allocate more data space for the program.

The first method links the variables of the two `sbrks`, so both allocate space on the same heap without stepping on each other. Each starts where the other left off. This method is useful for tools that are not concerned with the heap addresses being same as in the uninstrumented version of the program. Such tools include basic block counting, branch analysis, inline analysis and so on. This method is also sufficient for tools such as cache modeling that require precise heap addresses but do not allocate dynamic memory in analysis routines. This is the default behavior.

The second method is for tools that allocate dynamic memory and also require heap addresses to be same as in the uninstrumented version of the application program. To keep the application heap addresses as before, the heap is partitioned between the application and the analysis routines. The application heap starts at the same address but the analysis heap is now made to start at a higher address. The user supplies the offset by which the start of analysis heap is changed. ATOM modifies the `sbrk` in analysis routines to start at the new address; the two `sbrks` are not linked this time. The disadvantage of this method is that there is no runtime check if the application heap grows and enters into the analysis heap.

5 Performance

To find how well ATOM performs, two measurements are of interest: how long ATOM takes to instrument a program, and how the instrumented program's execution time compares to the uninstrumented program's execution time.

We used ATOM to instrument 20 SPEC92 programs with 11 tools. The tools are briefly described in Figure 5. The time taken to instrument a program is the sum of the ATOM's processing time and the time taken by the user's instrumentation routines. The time taken by a tool varies as each tool does different amounts of processing. For example, the *malloc* tool simply asks for the malloc procedure and instruments it; the processing time is very small. The *pipe* tool does static CPU pipeline scheduling for each basic block at instrumentation time and takes more time to instrument an application. The time taken to instrument 20 SPEC92 programs with each tool is also shown in Figure 5.

The execution time of the instrumented program is the sum of the execution time of the uninstrumented application program, the procedure call setup, and the time spent in the analysis routines. This total time represents the time needed by the user to get the final answers. Many systems process the collected data offline and do not include those numbers as part of data collecting statistics. The time spent in analysis routines is analogous to the postprocessing time required by other systems.

We compared each instrumented program's execution time to the uninstrumented program's execution time for each tool. Figure 6 shows the ratios for the SPEC92 programs. The procedure call overhead is dependent on the code in the analysis routines, and the number and type of arguments that are passed. ATOM uses the data flow summary information along with register renaming to find the necessary registers to save. The contribution of procedure call overhead in the instrumented program execution time is also dependent on the number of times the procedure calls take place. The *inline* tool instruments only procedure call sites; the total overhead is much less than the *cache* tool, which instruments each memory reference. The amount of work the analysis routines do when the control reaches them is totally dependent on information the user is trying to compute. Although the communication overhead is small, we expect it to decrease further when we implement live register analysis and inlining.

All measurements were done on Digital Alpha AXP 3000 Model 400 with 128 Mb memory.

6 Status

ATOM is built using OM and currently runs on Alpha AXP under OSF/1. It has been used with programs compiled with Fortran, C++ and two different C compilers. The system currently works on non-shared library modules. Work is in progress for adding support for shared libraries.

ATOM has been used both in hardware and software projects. Besides the SPEC92 benchmarks, it has successfully instrumented real applications of up to 96 Megabytes. The system is being used extensively inside Digital and at a few universities[12].

Our focus until now has mainly been on functionality. Few optimizations have been added to reduce the procedure call overhead. Currently, reduction in register saves has been obtained by computing data flow summary information of analysis routines. We plan to implement live register analysis along with inlining to further improve the performance. We are just starting to instrument the operating system.

[12] ATOM is available to external users. If you would like a copy, please contact the authors.

Analysis Tool	*Tool Description*	*Time to instrument SPEC92 suite*	*Average Time*
branch	prediction using 2-bit history table	110.46 secs	5.52 secs
cache	model direct mapped 8k byte cache	120.58 secs	6.03 secs
dyninst	computes dynamic instruction counts	126.31 secs	6.32 secs
gprof	call graph based profiling tool	113.24 secs	5.66 secs
inline	finds potential inlining call sites	146.50 secs	7.33 secs
io	input/output summary tool	121.60 secs	6.08 secs
malloc	histogram of dynamic memory	97.93 secs	4.90 secs
pipe	pipeline stall tool	257.48 secs	12.87 secs
prof	Instruction profiling tool	122.53 secs	6.13 secs
syscall	system call summary tool	120.53 secs	6.03 secs
unalign	unalign access tool	135.61 secs	6.78 secs

Figure 5: Time taken by ATOM to instrument 20 SPEC92 benchmark programs

Analysis Tool	*Instrumentation points*	Number of Arguments	*Time taken by Instrumented Program*
branch	each conditional branch	3	3.03x
cache	each memory reference	1	11.84x
dyninst	each basic block	3	2.91x
gprof	each procedure/each basic block	2	2.70x
inline	each call site	1	1.03x
io	before/after write procedure	4	1.01x
malloc	before/after malloc procedure	1	1.02x
pipe	each basic block	2	1.80x
prof	each procedure/each basic block	2	2.33x
syscall	before/after each system call	2	1.01x
unalign	each basic block	3	2.93x

Figure 6: Execution time of instrumented SPEC92 Programs as compared to uninstrumented SPEC92 programs

7 Conclusion

By separating object-module modification details from tool details and by presenting a high-level view of the program, ATOM has transferred the power of building tools to hardware and software designers. A tool designer concentrates only on what information is to be collected and how to process it. Tools can be built with few pages of code and they compute only what the user asks for. ATOM's fast communication between application and analysis means that there is no need to record traces as all data is immediately processed, and final results are computed in one execution of the instrumented program. Thus, one can process long-running programs. It has already been used to build a wide variety of tools to solve hardware and software problems. We hope ATOM will continue to be an effective platform for studies in software and architectural design.

Acknowledgements

Great many people have helped us bring ATOM to its current form. Jim Keller, Mike Burrows, Roger Cruz, John Edmondson, Mike McCallig, Dirk Meyer, Richard Swan and Mike Uhler were our first users and they braved through a mine field of bugs and instability during the early development process. Jeremy Dion, Ramsey Haddad, Russel Kao, Greg Lueck and Louis Monier built popular tools with ATOM. Many people, too many to name, gave comments, reported bugs, and provided encouragement. Roger Cruz, Jeremy Dion, Ramsey Haddad, Russell Kao, Jeff Mogul, Louis Monier, David Wall, Linda Wilson and anonymous PLDI reviewers gave useful comments on the earlier drafts of this paper. Our thanks to all.

References

[1] Anant Agarwal, Richard Sites, and Mark Horwitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. *Proceedings of the 13th International Symposium on Computer Architecture*, June 1986.

[2] Robert Bedichek. Some Efficient Architectures Simulation Techniques. *Winter 1990 USENIX Conference*, January 1990.

[3] Anita Borg, R.E. Kessler, Georgia Lazana, and David Wall. Long Address Traces from RISC Machines: Generation and Analysis, *Proceedings of the 17th Annual Symposium on Computer Architecture*, May 1990, also available as WRL Research Report 89/14, Sep 1989.

[4] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. MIT/LCS/TR-516, MIT, 1991.

[5] Robert F. Cmelik and David Keppel, Shade: A Fast Instruction-Set Simulator for Execution Profiling. Technical Report UWCSE 93-06-06, University of Washington.

[6] Susan J. Eggers, David R. Keppel, Eric J. Koldinger, and Henry M. Levy. Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor. *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, vol 8, no 1, May 1990.

[7] Stephen R. Goldschmidt and John L. Hennessy, The Accuracy of Trace-Driven Simulations of Multiprocessors. CSL-TR-92-546, Computer Systems Laboratory, Stanford University, September 1992.

[8] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software, Practice and Experience*, vol 24, no. 2, pp 197-218, February 1994.

[9] MIPS Computer Systems, Inc. *Assembly Language Programmer's Guide*, 1986.

[10] Richard L. Sites, ed. *Alpha Architecture Reference Manual* Digital Press, 1992.

[11] Amitabh Srivastava and David W. Wall. A Practical System for Intermodule Code Optimization at Link-Time. *Journal of Programming Language*, 1(1), pp 1-18, March 1993. Also available as WRL Research Report 92/6, December 1992.

[12] Amitabh Srivastava and David W. Wall. Link-Time Optimization of Address Calculation on a 64-bit Architecture. *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, to appear. Also available as WRL Research Report 94/1, February 1994.

[13] Amitabh Srivastava. Unreachable procedures in object-oriented programming, *ACM LOPLAS*, Vol 1, #4, pp 355-364, December 1992. Also available as WRL Research Report 93/4, August 1993.

[14] David W. Wall. Systems for late code modification. In Robert Giegerich and Susan L. Graham, eds, Code Generation - Concepts, Tools, Techniques, pp. 275-293, Springer-Verlag, 1992. Also available as WRL Research Report 92/3, May 1992.

Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation

Chi-Keung Luk Robert Cohn Robert Muth Harish Patil Artur Klauser Geoff Lowney
Steven Wallace Vijay Janapa Reddi † Kim Hazelwood

Intel Corporation †University of Colorado
Website: http://rogue.colorado.edu/Pin, Email: pin.project@intel.com

Abstract

Robust and powerful software instrumentation tools are essential for program analysis tasks such as profiling, performance evaluation, and bug detection. To meet this need, we have developed a new instrumentation system called *Pin*. Our goals are to provide *easy-to-use*, *portable*, *transparent*, and *efficient* instrumentation. Instrumentation tools (called *Pintools*) are written in C/C++ using Pin's rich API. Pin follows the model of ATOM, allowing the tool writer to analyze an application at the instruction level without the need for detailed knowledge of the underlying instruction set. The API is designed to be *architecture independent* whenever possible, making Pintools source compatible across different architectures. However, a Pintool can access architecture-specific details when necessary. Instrumentation with Pin is mostly *transparent* as the application and Pintool observe the application's original, uninstrumented behavior. Pin uses *dynamic compilation* to instrument executables while they are running. For efficiency, Pin uses several techniques, including inlining, register re-allocation, liveness analysis, and instruction scheduling to optimize instrumentation. This fully automated approach delivers significantly better instrumentation performance than similar tools. For example, Pin is 3.3x faster than Valgrind and 2x faster than DynamoRIO for basic-block counting. To illustrate Pin's versatility, we describe two Pintools in daily use to analyze production software. Pin is publicly available for Linux platforms on four architectures: IA32 (32-bit x86), EM64T (64-bit x86), Itanium®, and ARM. In the ten months since Pin 2 was released in July 2004, there have been over 3000 downloads from its website.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging-code inspections and walk-throughs, debugging aids, tracing; D.3.4 [*Programming Languages*]: Processors-compilers, incremental compilers

General Terms Languages, Performance, Experimentation

Keywords Instrumentation, program analysis tools, dynamic compilation

PLDI'05 June 12–15, 2005, Chicago, Illinois, USA.

1. Introduction

As software complexity increases, *instrumentation*—a technique for inserting extra code into an application to observe its behavior—is becoming more important. Instrumentation can be performed at various stages: in the source code, at compile time, post link time, or at run time. *Pin* is a software system that performs run-time binary instrumentation of Linux applications.

The goal of Pin is to provide an instrumentation platform for building a wide variety of program analysis tools for multiple architectures. As a result, the design emphasizes *ease-of-use*, *portability*, *transparency*, *efficiency*, and *robustness*. This paper describes the design of Pin and shows how it provides these features.

Pin's instrumentation is *easy to use*. Its user model is similar to the popular ATOM [30] API, which allows a tool to insert calls to instrumentation at arbitrary locations in the executable. Users do not need to manually inline instructions or save and restore state. Pin provides a rich API that abstracts away the underlying instruction set idiosyncrasies, making it possible to write *portable* instrumentation tools. The Pin distribution includes many sample architecture-independent *Pintools* including profilers, cache simulators, trace analyzers, and memory bug checkers. The API also allows access to architecture-specific information.

Pin provides *efficient* instrumentation by using a just-in-time (JIT) compiler to insert and optimize code. In addition to some standard techniques for dynamic instrumentation systems including code caching and trace linking, Pin implements *register reallocation*, *inlining*, *liveness analysis*, and *instruction scheduling* to optimize jitted code. This fully automated approach distinguishes Pin from most other instrumentation tools which require the user's assistance to boost performance. For example, Valgrind [22] relies on the tool writer to insert special operations in their intermediate representation in order to perform inlining; similarly DynamoRIO [6] requires the tool writer to manually inline and save/restore application registers.

Another feature that makes Pin efficient is *process attaching and detaching*. Like a debugger, Pin can attach to a process, instrument it, collect profiles, and eventually detach. The application only incurs instrumentation overhead during the period that Pin is attached. The ability to attach and detach is a necessity for the instrumentation of large, long-running applications.

Pin's JIT-based instrumentation defers code discovery until run time, allowing Pin to be more *robust* than systems that use static instrumentation or code patching. Pin can seamlessly handle mixed code and data, variable-length instructions, statically unknown indirect jump targets, dynamically loaded libraries, and dynamically generated code.

Pin preserves the original application behavior by providing instrumentation *transparency*. The application observes the same ad-

dresses (both instruction and data) and same values (both register and memory) as it would in an uninstrumented execution. Transparency makes the information collected by instrumentation more relevant and is also necessary for correctness. For example, some applications unintentionally access data beyond the top of stack, so Pin and the instrumentation do not modify the application stack.

Pin's first generation, Pin 0, supports Itanium®. The recently-released second generation, Pin 2, extends the support to four[1] architectures: IA32 (32-bit x86) [14], EM64T (64-bit x86) [15], Itanium® [13], and ARM [16]. Pin 2 for Itanium® is still under development.

Pin has been gaining popularity both inside and outside of Intel, with more than 3000 downloads since Pin 2 was first released in July 2004. This paper presents an in-depth description of Pin, and is organized as follows. We first give an overview of Pin's instrumentation capability in Section 2. We follow by discussing design and implementation issues in Section 3. We then evaluate in Section 4 the performance of Pin's instrumentation and compare it against other tools. In Section 5, we discuss two sample Pintools used in practice. Finally, we relate Pin to other work in Section 6 and conclude in Section 7.

2. Instrumentation with Pin

The Pin API makes it possible to observe all the architectural state of a process, such as the contents of registers, memory, and control flow. It uses a model similar to ATOM [30], where the user adds procedures (as known as *analysis routines* in ATOM's notion) to the application process, and writes *instrumentation routines* to determine where to place calls to analysis routines. The arguments to analysis routines can be architectural state or constants. Pin also provides a limited ability to alter the program behavior by allowing an analysis routine to overwrite application registers and application memory.

Instrumentation is performed by a just-in-time (JIT) compiler. The input to this compiler is not bytecode, however, but a native executable. Pin intercepts the execution of the first instruction of the executable and generates ("compiles") new code for the straight-line code sequence starting at this instruction. It then transfers control to the generated sequence. The generated code sequence is almost identical to the original one, but Pin ensures that it regains control when a branch exits the sequence. After regaining control, Pin generates more code for the branch target and continues execution. Every time the JIT fetches some code, the Pintool has the opportunity to instrument it before it is translated for execution. The translated code and its instrumentation is saved in a code cache for future execution of the same sequence of instructions to improve performance.

In Figure 1, we list the code that a user would write to create a Pintool that prints a trace of address and size for every memory write in a program. The `main` procedure initializes Pin, registers the procedure called `Instruction`, and tells Pin to start execution of the program. The JIT calls `Instruction` when inserting new instructions into the code cache, passing it a handle to the decoded instruction. If the instruction writes memory, the Pintool inserts a call to `RecordMemWrite` *before* the instruction (specified by the argument `IPOINT_BEFORE` to `INS_InsertPredicatedCall`), passing the instruction pointer (specified by `IARG_INST_PTR`), effective address for the memory operation (specified by `IARG_MEMORYWRITE_EA`), and number of bytes written (specified by `IARG_MEMORYWRITE_SIZE`). Using

[1] Although EM64T is a 64-bit extension of IA32, we classify it as a separate architecture because of its many new features such as 64-bit addressing, a flat address space, twice the number of registers, and new software conventions [15].

```
FILE * trace;

// Print a memory write record
VOID RecordMemWrite(VOID * ip, VOID * addr, UINT32 size) {
    fprintf(trace,"%p: W %p %d\n", ip, addr, size);
}

// Called for every instruction
VOID Instruction(INS ins, VOID *v) {
    // instruments writes using a predicated call,
    // i.e. the call happens iff the store is
    // actually executed
    if (INS_IsMemoryWrite(ins))
        INS_InsertPredicatedCall(
            ins, IPOINT_BEFORE, AFUNPTR(RecordMemWrite),
            IARG_INST_PTR, IARG_MEMORYWRITE_EA,
            IARG_MEMORYWRITE_SIZE, IARG_END);
}

int main(int argc, char *argv[]) {
    PIN_Init(argc, argv);
    trace = fopen("atrace.out", "w");
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_StartProgram(); // Never returns
    return 0;
}
```

Figure 1. A Pintool for tracing memory writes.

`INS_InsertPredicatedCall` ensures that `RecordMemWrite` is invoked only if the memory instruction is predicated `true`.

Note that the same source code works on all architectures. The user does not need to know about the bundling of instructions on Itanium, the various addressing modes on each architecture, the different forms of predication supported by Itanium and ARM, x86 string instructions that can write a variable-size memory area, or x86 instructions like `push` that can implicitly write memory.

Pin provides a comprehensive API for inspection and instrumentation. In this particular example, instrumentation is done one instruction at a time. It is also possible to inspect whole *traces*, *procedures*, and *images* when doing instrumentation. The Pin user manual [12] provides a complete description of the API.

Pin's *call-based* model is simpler than other tools where the user can insert instrumentation by adding and deleting statements in an intermediate language. However, it is equally powerful in its ability to observe architectural state and it frees the user from the need to understand the idiosyncrasies of an instruction set or learn an intermediate language. The inserted code may overwrite scratch registers or condition codes; Pin efficiently saves and restores state around calls so these side effects do not alter the original application behavior. The Pin model makes it possible to write efficient and architecture-independent instrumentation tools, regardless of whether the instruction set is RISC, CISC, or VLIW. A combination of inlining, register re-allocation, and other optimizations makes Pin's procedure call-based model as efficient as lower-level instrumentation models.

3. Design and Implementation

In this section, we begin with a system overview of Pin. We then discuss how Pin initially gains control of the application, followed by a detailed description of how Pin dynamically compiles the application. Finally, we discuss the organization of Pin source code.

3.1 System Overview

Figure 2 illustrates Pin's software architecture. At the highest level, Pin consists of a virtual machine (VM), a code cache, and an instru-

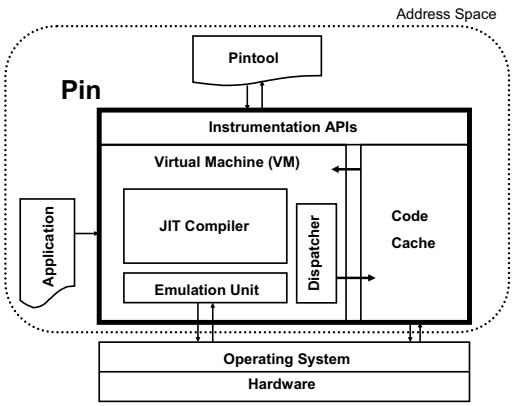


Figure 2. Pin's software architecture

mentation API invoked by Pintools. The VM consists of a just-in-time compiler (JIT), an emulator, and a dispatcher. After Pin gains control of the application, the VM coordinates its components to execute the application. The JIT compiles and instruments application code, which is then launched by the dispatcher. The compiled code is stored in the code cache. Entering/leaving the VM from/to the code cache involves saving and restoring the application register state. The emulator interprets instructions that cannot be executed directly. It is used for system calls which require special handling from the VM. Since Pin sits above the operating system, it can only capture user-level code.

As Figure 2 shows, there are three binary programs present when an instrumented program is running: the application, Pin, and the Pintool. Pin is the engine that jits and instruments the application. The Pintool contains the instrumentation and analysis routines and is linked with a library that allows it to communicate with Pin. While they share the *same* address space, they do *not* share any libraries and so there are typically three copies of `glibc`. By making all of the libraries private, we avoid unwanted interaction between Pin, the Pintool, and the application. One example of a problematic interaction is when the application executes a `glibc` function that is not reentrant. If the application starts executing the function and then tries to execute some code that triggers further compilation, it will enter the JIT. If the JIT executes the same `glibc` function, it will enter the same procedure a second time while the application is still executing it, causing an error. Since we have separate copies of `glibc` for each component, Pin and the application do not share any data and cannot have a re-entrancy problem. The same problem can occur when we jit the analysis code in the Pintool that calls `glibc` (jitting the analysis routine allows us to greatly reduce the overhead of simple instrumentation on Itanium).

3.2 Injecting Pin

The injector loads Pin into the address space of an application. Injection uses the Unix Ptrace API to obtain control of an application and capture the processor context. It loads the Pin binary into the application address space and starts it running. After initializing itself, Pin loads the Pintool into the address space and starts it running. The Pintool initializes itself and then requests that Pin start the application. Pin creates the initial context and starts jitting the application at the entry point (or at the current PC in the case of attach). Using Ptrace as the mechanism for injection allows us to attach to an already running process in the same way as a debugger. It is also possible to detach from an instrumented process and continue executing the original, uninstrumented code.

Other tools like DynamoRIO [6] rely on the `LD_PRELOAD` environment variable to force the dynamic loader to load a shared library in the address space. Pin's method has three advantages. First, `LD_PRELOAD` does not work with *statically-linked* binaries, which many of our users require. Second, loading an extra shared library will shift all of the application shared libraries and some dynamically allocated memory to a higher address when compared to an uninstrumented execution. We attempt to preserve the original behavior as much as possible. Third, the instrumentation tool cannot gain control of the application until after the shared-library loader has partially executed, while our method is able to instrument the very first instruction in the program. This capability actually exposed a bug in the Linux shared-library loader, resulting from a reference to uninitialized data on the stack.

3.3 The JIT Compiler

3.3.1 Basics

Pin compiles from one ISA directly into the same ISA (e.g., IA32 to IA32, ARM to ARM) without going through an intermediate format, and the compiled code is stored in a software-based code cache. Only code residing in the code cache is executed—the original code is never executed. An application is compiled one *trace* at a time. A trace is a straight-line sequence of instructions which terminates at one of the conditions: (i) an *unconditional* control transfer (branch, call, or return), (ii) a pre-defined number of *conditional* control transfers, or (iii) a pre-defined number of instructions have been fetched in the trace. In addition to the last exit, a trace may have multiple side-exits (the conditional control transfers). Each exit initially branches to a *stub*, which re-directs the control to the VM. The VM determines the target address (which is statically unknown for indirect control transfers), generates a new trace for the target if it has not been generated before, and resumes the execution at the target trace.

In the rest of this section, we discuss the following features of our JIT: trace linking, register re-reallocation, and instrumentation optimization. Our current performance effort is focusing on IA32, EM64T, and Itanium, which have all these features implemented. While the ARM version of Pin is fully functional, some of the optimizations are not yet implemented.

3.3.2 Trace Linking

To improve performance, Pin attempts to branch directly from a trace exit to the target trace, bypassing the stub and VM. We call this process *trace linking*. Linking a *direct* control transfer is straightforward as it has a unique target. We simply patch the branch at the end of one trace to jump to the target trace. However, an *indirect* control transfer (a jump, call, or return) has multiple possible targets and therefore needs some sort of target-prediction mechanism.

Figure 3(a) illustrates our indirect linking approach as implemented on the x86 architecture. Pin translates the indirect jump into a move and a direct jump. The move puts the indirect target address into register `%edx` (this register as well as the `%ecx` and `%esi` shown in Figure 3(a) are obtained via register re-allocation, as we will discuss in Section 3.3.3). The direct jump goes to the first predicted target address `0x40001000` (which is mapped to `0x70001000` in the code cache for this example). We compare `%edx` against `0x40001000` using the `lea`/`jecxz` idiom used in DynamoRIO [6], which avoids modifying the conditional flags register `eflags`. If the prediction is correct (i.e. `%ecx`=0), we will branch to `match1` to execute the remaining code of the predicted target. If the prediction is wrong, we will try another predicted target `0x40002000` (mapped to `0x70002000` in the code cache). If the target is not found on the chain, we will branch to `LookupHtab_1`, which searches for the target in a hash table (whose base address is

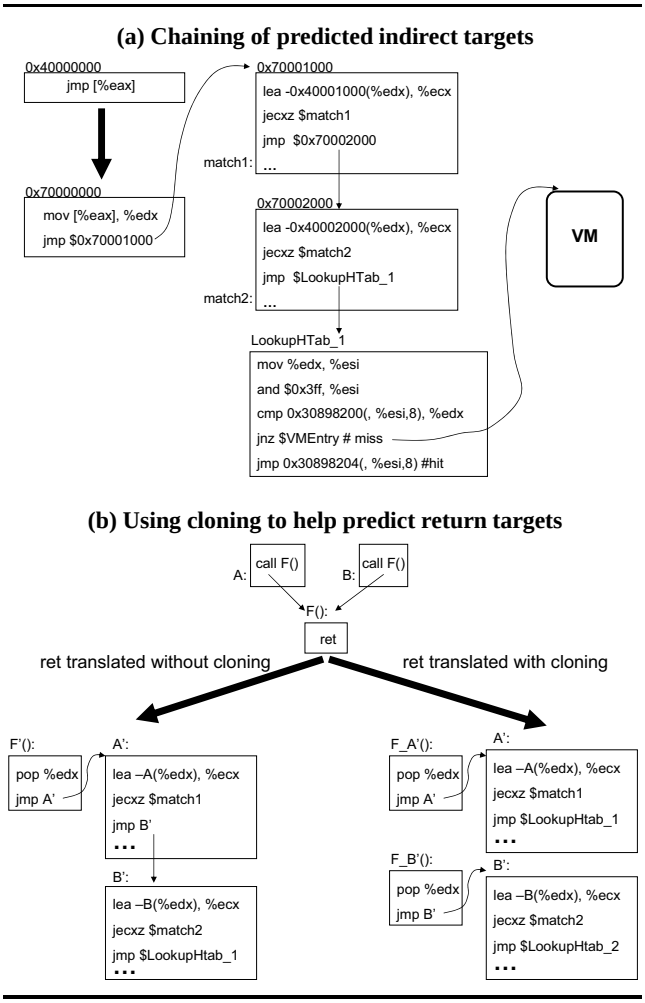


Figure 3. Compiling indirect jumps and returns

`0x30898200` in this example). If the search succeeds, we will jump to the translated address corresponding to the target. If the search fails, we will transfer to the VM for indirect target resolution.

While our indirect linking mechanism is similar to the approach taken in DynamoRIO [6], there are three important differences. First, in DynamoRIO, the entire chain is generated at one time and embedded at the translation of the indirect jump. Therefore no new predicted target can be added onto the chain after it is generated. In contrast, our approach *incrementally* builds the chain while the program is running and thus we can insert newly seen targets onto the chain *in any order* (e.g., Pin can put a new target either at the front or the end of the chain). These new targets can be found in the chain the next time that they occur, without searching the hash table. The second difference is that DynamoRIO uses a *global* hash table for all indirect jumps whereas Pin uses a *local* hash table for each individual indirect jump. A study by Kim and Smith [17] shows that the local hash table approach typically offers higher performance. The third difference is that we apply *function cloning* [10] to accelerate the most common form of indirect control transfers: *returns*. If a function is called from multiple sites, we clone multiple copies of the function, one for each call site. Consequently, a return in each clone will have only one predicted target on the chain in most cases, as illustrated by the example in Figure 3(b). To implement cloning, we associate a *call stack* with each trace (more precisely to the *static context* of each trace, which we will discuss in Section 3.3.3). Each call stack remembers the last four call sites and is compactly represented by hashing the call-site addresses into a single 64-bit integer.

3.3.3 Register Re-allocation

During jitting, we frequently need extra registers. For example, the code for resolving indirect branches in Figure 3(a) needs three free registers. When instrumentation inserts a call into an application, the JIT must ensure that the call does not overwrite any scratch registers that may be in use by the application. Rather than obtaining extra registers in an ad-hoc way, Pin re-allocates registers used in both the application and the Pintool, using linear-scan register allocation [24]. Pin's allocator is unique in that it does interprocedural allocation, but must compile one *trace* at a time while incrementally discovering the flow graph during execution. In contrast, static compilers can compile one *file* at a time and bytecode JITs [5, 8] can compile a whole *method* at one time. We describe two issues that our trace-based register re-allocation scheme must address: *register liveness analysis* and *reconciliation of register bindings*.

Register Liveness Analysis Precise liveness information of registers at trace exits makes register allocation more effective since dead registers can be reused by Pin without introducing spills. Without a complete flow graph, we must incrementally compute liveness. After a trace at address A is compiled, we record the liveness at the beginning of the trace in a hash table using address A as the key. If a trace exit has a statically-known target, we attempt to retrieve the liveness information from the hash table so we can compute more precise liveness for the current trace. This simple method introduces negligible space and time overhead, yet is effective in reducing register spills introduced by Pin's register allocation.

Reconciliation of Register Bindings Trace linking (see Section 3.3.2) tries to make traces branch directly to each other. When registers are reallocated, the JIT must ensure than the register binding at the trace exit of the source trace matches the bindings of the entrance of the destination trace. A straightforward method is to require a standard binding of registers between traces. For example Valgrind [22] requires that all virtual register values be flushed to memory at the end of a basic block. This approach is simple but inefficient. Figure 4(b) shows how Valgrind would re-allocate registers for the original code shown in Figure 4(a). Here, we assume that virtual `%ebx` is bound to physical `%esi` in Trace 1 but to physical `%edi` in Trace 2. Virtual `%eax` and `%ebx` are saved at Trace 1's exit because they have been modified in the trace, and they are reloaded before their uses in Trace 2. `EAX` and `EBX` are the memory locations allocated by the JIT for holding the current values of virtual `%eax` and `%ebx`, respectively.

In contrast, Pin keeps a virtual register in the same physical register across traces whenever possible. At a trace exit e, if the target t has *not* been compiled before, our JIT will compile a new trace for t using the virtual-to-physical register binding at e, say B_e. Therefore, e can branch directly to t. Figure 4(c) shows how Pin would re-allocate registers for the same original code, assuming that target t has not been compiled before. Nevertheless, if target t has been previously compiled with a register binding $B_t \neq B_e$, then our JIT will generate compensation code [19] to reconcile the register binding from B_e to B_t instead of compiling a new trace for B_e. Figure 4(d) shows how Pin would re-allocate registers for the same original code, this time assuming that the target t has been previously compiled with a different binding in the virtual `%ebx`. In practice, these bindings show differences in only one or two virtual registers, and are therefore more efficient than Valgrind's method.

A design choice we encountered was *where* to put the compensation code. It could be placed *before* the branch, which is exactly the situation shown in Figure 4(d) where the two `mov` instructions

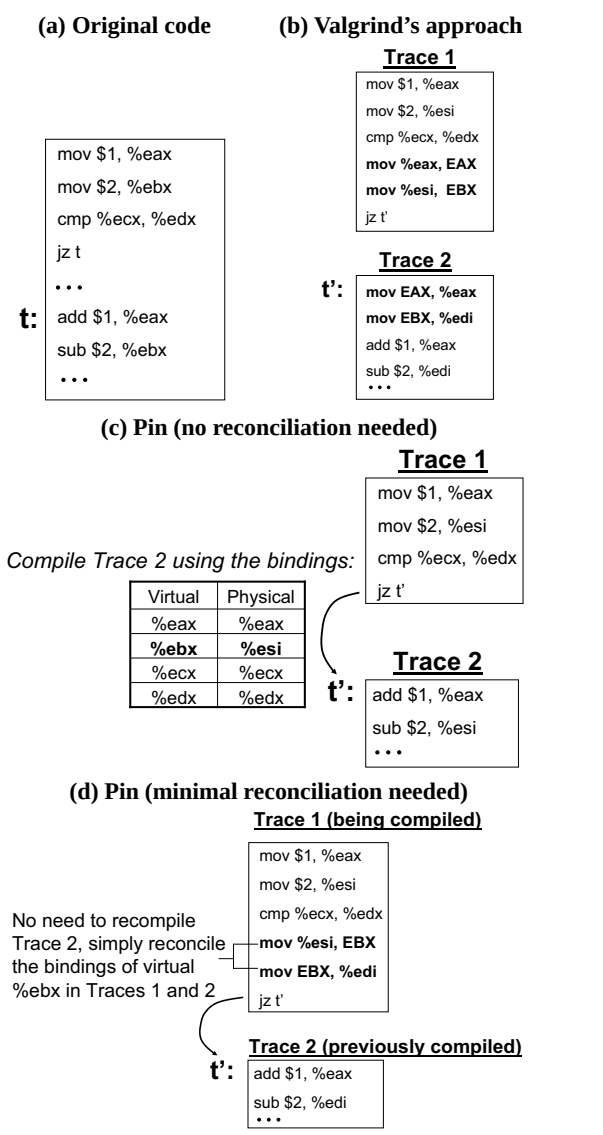


Figure 4. Reconciliation of Register Bindings

that adjust the binding are placed before the `jz`. Or the compensation code could be placed *after* the branch (in that case, the two `mov` instructions in Figure 4(d) would be placed in between the `jz` and t'). We chose the "before" approach because our experimental data showed that it generally resulted in fewer unique bindings, therefore reducing the memory consumed by the compiler. Placing the compensation code before the branch is equivalent to targeting the register allocation to match the binding at the branch target.

To support reconciliation of register bindings, we need to remember the binding at a trace's entry. This is done by associating each trace with a *static context (sct)*, which contains a group of static properties that hold at the trace's entry. Register binding is one such property; another example property is the call stack of the trace, which is used for function cloning (see Section 3.3.2). So, precisely speaking, a trace is defined as a pair $< entryIaddr, entrySct >$, where $entryIaddr$ is the original instruction address of the trace's entry and $entrySct$ is the static context of the trace. Before the JIT compiles a new trace, it will first search for a *compatible* trace in the code cache. Two traces are compatible if they have the same $entryIaddr$ and their $entrySct$'s are either identical or different in only their register bindings (in that case we can reconcile from one register binding to the other, as we have exemplified in Figure 4(d)). If a compatible trace is found, the JIT will simply use it instead of generating a new trace.

3.3.4 Thread-local Register Spilling

Pin reserves an area in memory for spilling virtual registers (e.g., `EAX` and `EBX` shown in Figure 4(b) are two locations in this spilling area). To support multithreading, this area has to be thread local. When Pin starts an application thread, it allocates the spilling area for this thread and steals a physical register (`%ebx` on x86, `%r7` on Itanium) to be the *spill pointer*, which points to the base of this area. From that point on, any access to the spilling area can be made through the spill pointer. When we switch threads, the spill pointer will be set to the spilling area of the new thread. In addition, we exploit an optimization opportunity coming from the *absolute addressing* mode available on the x86 architecture. Pin starts an application assuming that it is single threaded. Accesses to the spilling area are made through absolute addressing and therefore Pin does not need a physical register for the spill pointer. If Pin later discovers that the application is in fact multithreaded, it will invalidate the code cache and recompile the application using the spill pointer to access the spilling area (Pin can detect multithreading because it intercepts all thread-create system calls). Since single-threaded applications are more common than multithreaded ones, this hybrid approach works well in practice.

3.3.5 Optimizing Instrumentation Performance

As we will show in Section 4, most of the slowdown from instrumentation is caused by executing the instrumentation code, rather than the compilation time (which includes inserting the instrumentation code). Therefore, it is beneficial to spend more compilation time in optimizing calls to analysis routines. Of course, the runtime overhead of executing analysis routines highly depends on their invocation frequency and their complexity. If analysis routines are complex, there is not much optimization that our JIT can do. However, there are many Pintools whose frequently-executed analysis routines perform only simple tasks like counting and tracing. Our JIT optimizes those cases by *inlining* the analysis routines, which reduces execution overhead as follows. Without inlining, we call a *bridge routine* that saves all caller-saved registers, sets up analysis routine arguments, and finally calls the analysis routine. Each analysis routine requires two calls and two returns for each invocation. With inlining, we eliminate the bridge and thus save those two calls and returns. Also, we no longer explicitly save caller-saved registers. Instead, we rename the caller-saved registers in the inlined body of the analysis routine and allow the register allocator to manage the spilling. Furthermore, inlining enables other optimizations like constant folding of analysis routine arguments.

We perform an additional optimization for the x86 architecture. Most analysis routines modify the conditional flags register `eflags` (e.g., if an analysis routine increments a counter). Hence, we must preserve the original `eflags` value as seen by the application. However, accessing the `eflags` is fairly expensive because it must be done by pushing it onto the stack[2]. Moreover, we must switch to another stack before pushing/popping the `eflags` to avoid changing the application stack. Pin avoids saving/restoring `eflags` as much as possible by using *liveness analysis* on the `eflags`. The liveness analysis tracks the individual bits in the `eflags` written and read by each x86 instruction. We frequently discover that the

[2] On IA32, we can use `lahf`/`sahf` to access the `eflags` without involving the stack. However, we decided not to use them since these two instructions are not implemented on current EM64T processors.

Architecture	Number of Source Files	Number of Lines (including comments)
Generic	87 (48%)	53595 (47%)
x86 (IA32+EM64T)	34 (19%)	22794 (20%)
Itanium	34 (19%)	20474 (18%)
ARM	27 (14%)	17933 (15%)
TOTAL	182 (100%)	114796 (100%)

Table 1. Distribution of Pin source among different architectures running Linux. Over 99% of code is written in C++ and the remaining is in assembly.

	Hardware	Linux	Compiler	Binary
IA32	1.7GHz Xeon™, 256KB L2 cache, 2GB Memory	2.4.9	gcc 3.3.2, -O3 for SPECint (except in eon where we use icc)	Shared
			icc 8.0 for SPECfp	Static
EM64T	3.4GHz Xeon™, 1MB L2 cache, 4GB Memory	2.4.21	Intel ® compiler (icc 8.0), with interprocedural & profile-guided optimizations	Static
Itanium®	1.3GHz Itanium®2, 6MB L2 cache, 12GB Memory	2.4.18		Static
ARM	400 MHz XScale® 80200, 128 MB Memory	2.4.18	gcc 3.4.1, -O2	Static

Table 2. Experimental setup.

`eflags` are dead at the point where an analysis routine call is inserted, and are able to eliminate saving and restoring of the `eflags`.

Finally, to achieve even better performance, the Pintool writer can specify a hint (`IPOINT_ANYWHERE`) telling Pin that a call to an analysis routine can be inserted *anywhere* inside the scope of instrumentation (e.g., a basic block or a trace). Then Pin can exploit a number of optimization opportunities by *scheduling* the call. For instance, Pin can insert the call immediately before an instruction that overwrites a register (or `eflags`) and thereby the analysis routine can use that register (or `eflags`) without first spilling it.

3.4 Organization of Pin Source Code

Since Pin is a multi-platform system, source code sharing is a key to minimizing the development effort. Our first step was to share the basic data structures and intermediate representations with Ispike [20], a static binary optimizer we previously developed. Then we organized Pin source into generic, architecture dependent, or operating-system dependent modules. Some components like the code cache are purely generic, while other components like the register allocator contain both generic and architecture-dependent parts. Table 1 shows the distribution of Pin source among different architectures, in terms of number of source files and lines. We combine IA32 and EM64T in Table 1 since they are similar enough to share the same source files. The x86 numbers do not include the decoder/encoder while the Itanium numbers do not include the instruction scheduler. The reason is that we borrow these two components from other Intel tools in library form and we do not have their sources. The data reflects that we have done a reasonable job in code sharing among architectures as about 50% of code is generic.

4. Experimental Evaluation

In this section, we first report the performance of Pin without any instrumentation on the four supported architectures. We then report the performance of Pin with a standard instrumentation—basic-block counting. Finally, we compare the performance of Pin with two other tools: DynamoRIO and Valgrind, and show that Pin's instrumentation performance is superior across our benchmarks.

Our experimental setup is described in Table 2. For IA32, we use dynamically-linked SPECint binaries compiled with `gcc -O3`. We compiled `eon` with `icc` because the `gcc -O3` version does not work, even without applying Pin. We could not use the official statically-linked, `icc`-generated binaries for all programs because DynamoRIO cannot execute them. We ran the SPEC2000 suite [11] using reference inputs on IA32, EM64T, and Itanium. On ARM, we are only able to run the training inputs due to limited physical memory (128MB), even when executing uninstrumented binaries. Floating-point benchmarks are not used on ARM as it does not have floating-point hardware.

4.1 Pin Performance without Instrumentation

Figure 5 shows the performance of applying Pin to the benchmarks on the four architectures, *without* any instrumentation. Since Pin 2/Itanium is still under development, we instead use Pin 0 for Itanium experiments. The y-axis is the time normalized to the native run time (i.e. 100%). The slowdown of Pin 2 on IA32 and EM64T is similar. In both cases, the average run-time overhead is around 60% for integer and within 5% for floating point. The higher overhead on the integer side is due to many more indirect branches and returns. The slowdown of Pin 0 on Itanium follows the same trend but is generally larger than on IA32 and EM64T, especially for floating-point benchmarks. This is probably because Itanium is an in-order architecture, so its performance depends more on the quality of the jitted code. In contrast, IA32 and EM64T are out-of-order architectures that can tolerate the overhead introduced in the jitted code. Pin's performance on ARM is worse than the other three architectures because indirect linking (see Section 3.3.2) is not yet implemented and there are fewer computational resources (ILP and memory) available.

One downside of dynamic compilation is that the compilation time is directly reflected in the application's run time. To understand the performance impact of dynamic compilation, we divide the total run time into the components shown in Figures 5(a), (b), and (d) (Pin 0 source code is not instrumented and hence does not have the breakdown). *Code Cache* denotes the time executing the jitted code stored in the code cache. Ideally, we would like this component to approach 100%. We divide the JIT time into three categories: *JIT-Decode*, *JIT-Regalloc*, and *JIT-Other*. *JIT-Decode* is the time spent decoding and encoding instructions, which is a non-trivial task on the x86 architecture. *JIT-Regalloc* is the time spent in register re-allocation. *JIT-Other* denotes the remaining time spent in the JIT. The last component is *VM*, which includes all other time spent in the virtual machine, including instruction emulation and resolving mispredicted indirect control transfers.

As Figures 5 (a) and (b) show, the JIT and VM components on IA32 and EM64T are mostly small except in `gcc` and `perlbmk`. These two benchmarks have the largest instruction footprint in SPEC2000 and their execution times are relatively short. Consequently, there is insufficient code reuse for Pin to amortize its compilation cost. In particular, Pin pays a high cost in re-allocating registers compared to most other tools that do not re-allocate registers. Nevertheless, the advantages provided by register re-allocation outweigh its compilation overhead (e.g., register re-allocation makes it easy to provide Pin and Pintools more virtual registers than the number of physical registers supported by the hardware). In practice, the performance overhead is a serious concern only for long-running applications. In that case, we would have sufficient code reuse to amortize the cost of register re-allocation. Figure 5(d) shows a different trend for ARM, where the VM component is

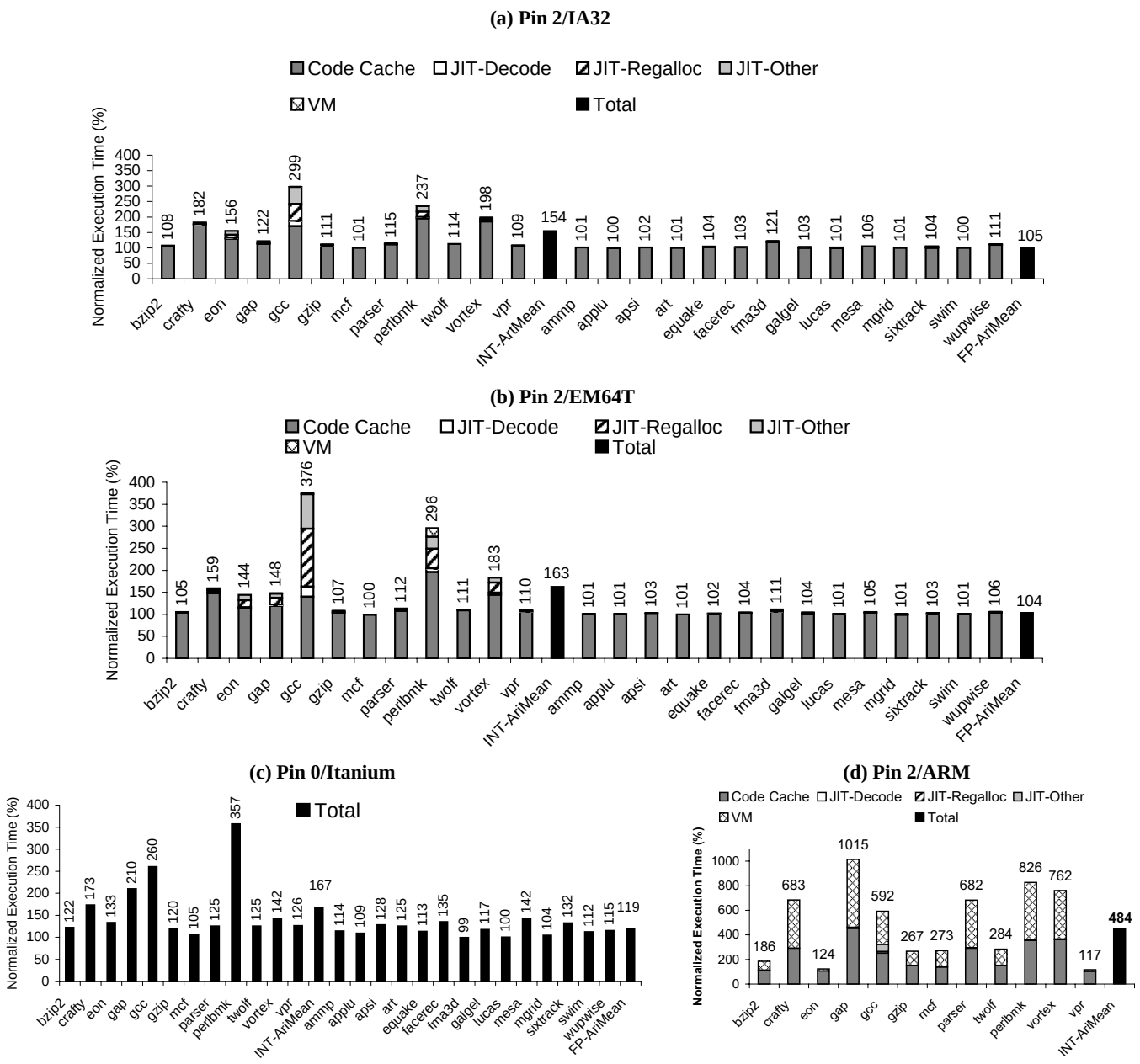


Figure 5. Performance of Pin (*without* any instrumentation) on four architectures. The y-axis is the time normalized to the native run time (i.e. 100%). INT-AriMean and FP-AriMean on the x-axis are the arithmetic means of the integer and floating-point benchmarks, respectively. The legends are explained in Section 4.1.

large but all JIT components are small. This is because register reallocation and indirect linking are not yet implemented on ARM. As a result, all indirect control transfers are resolved by the VM.

4.2 Pin Performance with Instrumentation

We now study the performance of Pin with basic-block counting, which outputs the execution count of every basic block in the application. We chose to measure this tool's performance because basic-block counting is commonly used and can be extended to many other tools such as `Opcodemix`, which we will discuss in Section 5.1. Also, this tool is simple enough that its performance largely depends on how well the JIT integrates it into the application. On the other hand, performance of a complex tool like detailed cache simulation mostly depends on the tool's algorithm. In that case, our JIT has less of an impact on performance.

Figure 6 shows the performance of basic-block counting using Pin on the IA32 architecture. Each benchmark is tested using four different optimization levels. Without any optimization, the overhead is fairly large (as much as 20x slowdown in `crafty`). Adding inlining helps significantly; the average slowdown improves from 10.4x to 7.8x for integer and from 3.9x to 3.5x for floating point. The biggest performance boost comes from the `eflags` liveness

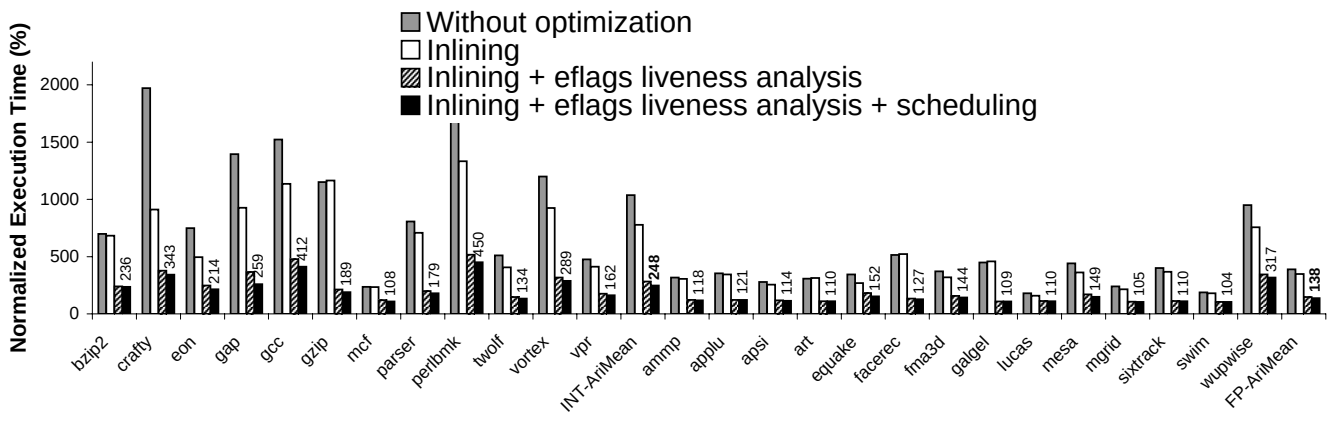


Figure 6. Performance of Pin with basic-block counting instrumentation on the IA32 architecture.

analysis, reducing the average slowdown to 2.8x for integer and 1.5x for floating point. Scheduling of instrumentation code further reduces the slowdown to 2.5x for integer and 1.4x for floating point.

4.3 Performance Comparison with Valgrind and DynamoRIO

We now compare the performance of Pin against Valgrind and DynamoRIO. Valgrind is a popular instrumentation tool on Linux and is the only binary-level JIT other than Pin that re-allocates registers. DynamoRIO is generally regarded as the performance leader in binary-level dynamic optimization. We used the latest release of each tool for this experiment: Valgrind 2.2.0 [22] and DynamoRIO 0.9.3 [6]. We ran two sets of experiments: one without instrumentation and one with basic-block counting instrumentation. We implemented basic-block counting by modifying a tool in the Valgrind package named `lackey` and a tool in the DynamoRIO package named `countcalls`. We show only the integer results in Figure 7 as integer codes are more problematic than floating-point codes in terms of the slowdown caused by instrumentation.

Figure 7(a) shows that without instrumentation both Pin and DynamoRIO significantly outperform Valgrind. DynamoRIO is faster than Pin on `gcc`, `perlbmk` and `vortex`, mainly because Pin spends more jitting time in these three benchmarks (refer back to Figure 5(a) for the breakdown) than DynamoRIO, which does not re-allocate registers. Pin is faster than DynamoRIO on a few benchmarks such as `crafty` and `gap` possibly because of the advantages that Pin has in indirect linking (i.e. incremental linking, cloning, and local hash tables). Overall, DynamoRIO is 12% faster than Pin without instrumentation. Given that DynamoRIO was primarily designed for *optimization*, the fact that Pin can come this close is quite acceptable.

When we consider the performance with instrumentation shown in Figure 7(b), Pin outperforms both DynamoRIO and Valgrind by a significant margin: on average, Valgrind slows the application down by 8.3 times, DynamoRIO by 5.1 times, and Pin by 2.5 times. Valgrind inserts a call before every basic block's entry but it does not automatically inline the call. For DynamoRIO, we use its low-level API to update the counter inline. Nevertheless, DynamoRIO still has to save and restore the `eflags` explicitly around each counter update. In contrast, Pin automatically inlines the call and performs liveness analysis to eliminate unnecessary `eflags` save/restore. This clearly demonstrates a main advantage of Pin: it provides efficient instrumentation without shifting the burden to the Pintool writer.

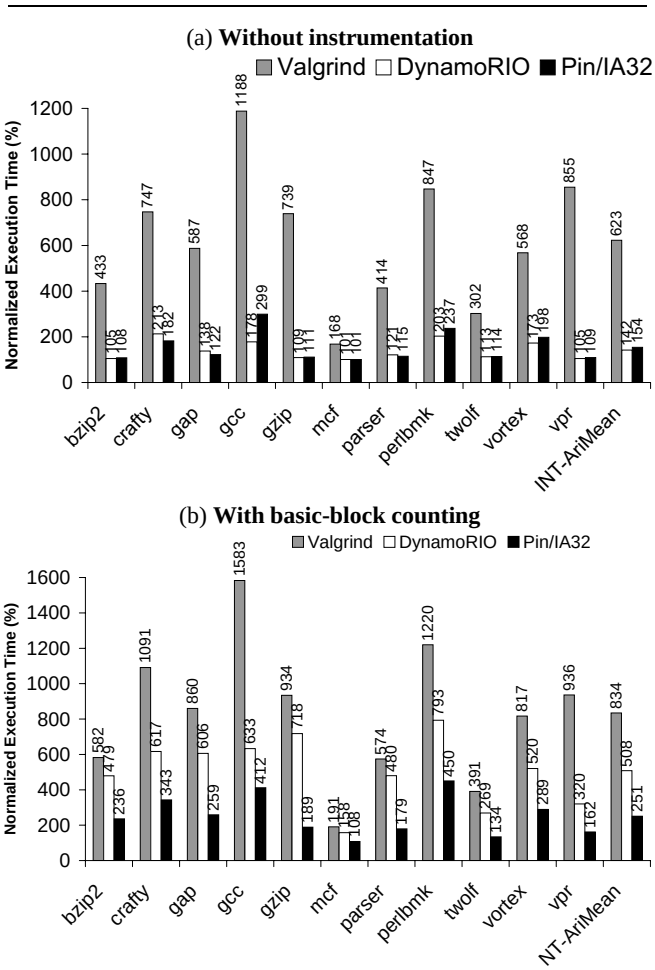


Figure 7. Performance comparison among Valgrind, DynamoRIO, and Pin. `Eon` is excluded because DynamoRIO does not work on the icc-generated binary of this benchmark. Omitting `eon` causes the two arithmetic means of Pin/IA32 slightly different than the ones shown in Figures 5(a) and 6.

5. Two Sample PinTools

To illustrate how Pin is used in practice, we discuss two Pintools that have been used by various groups inside Intel. The first tool, `Opcodemix`, studies the frequency of different instruction types in a program. It is used to compare codes generated by different compilers. The second tool, `PinPoints`, automatically selects representative points in the execution of a program and is used to accelerate processor simulation.

5.1 Opcodemix

`Opcodemix`, whose source code is included in the Pin 2 distribution [12], is a simple Pintool that can determine the dynamic mix of opcodes for a particular execution of a program. The statistics can be broken down on a per basic-block, per routine, or per image basis. Conceptually this tool is implemented as a basic-block profiler. We insert a counter at the beginning of each basic block in a trace. Upon program termination we walk through all the counters. From the associated basic-block's starting address, we can determine the function it belongs to and the instruction mix in that basic block. While the output of `Opcodemix` is ISA dependent (different ISAs have different opcodes), the implementation is generic—the same source code for `Opcodemix` is used on the four architectures.

Though simple, `Opcodemix` has been quite useful both for architectural and compiler comparison studies. As an example, the following analysis revealed a compiler performance problem. We collected `Opcodemix` statistics for the SPEC2000 images produced by two compilers, which we refer to as compilers A and B, for the EM64T architecture. For the benchmark `crafty`, we found that the image produced by compiler A executed 2% more dynamic instructions than the image produced by compiler B. To understand the cause of the extra instructions, we looked at the instruction distribution of frequently-executed routines. The data for the routine `PopCnt()` is shown in Table 3, where opcodes with significantly different frequencies in the two compilers are marked with "←". Examining the `PopCnt()` codes from the two compilers revealed that the deltas in `JE` and `JNZ` were due to different code-layout decisions, and the delta in `MOVL` was due to different register selections. The most surprising finding was the extra `PUSHQ` and `POPQ` generated by compiler A. Figure 8 shows the `PopCnt()` code generated by compiler A. After communicating with compiler A's writers, we learned that the `push` and `pop` are used for stack alignment but are in fact unnecessary in this case. As a result, this performance problem is now fixed in the latest version of compiler A.

In addition to SPEC, we use `Opcodemix` to analyze the Oracle database performance. Typically, more than 10 "Oracle" processes run on the system, but we want to ignore the database startup and only observe a single process performing a transaction. We first run Oracle natively (i.e. without Pin) to startup the database. Next we attach Pin to a single database server process and have it perform a transaction while collecting a profile. Pin's dynamic just-in-time instrumentation allows us to avoid instrumenting the entire 60 MB Oracle binary, and the attach feature allows us to avoid instrumenting the database startup and the other processes.

5.2 PinPoints

The purpose of the `PinPoints` [23] toolkit is to automate the otherwise tedious process of finding regions of programs to simulate, validating that the regions are representative, and generating traces for those regions. There are two major challenges in simulating large commercial programs. First, these programs have long run times, and detailed simulation of their entire execution is too time consuming to be practical. Second, these programs often have large resource requirements, operating system and device-driver dependencies, and elaborate license-checking mechanisms, making it difficult to execute them on simulators. We address the first challenge using SimPoint [28]—a methodology that uses phase analysis for finding representative regions for simulation. For the second challenge, we use Pin to collect SimPoint profiles (which we call *PinPoints*) and instruction traces, eliminating the need to execute the program on a simulator. The ease of running applications with Pintools is a key advantage of the `PinPoints` toolkit. `PinPoints` has been used to collect instruction traces for a wide variety of programs; Table 4 lists some of the Itanium applications (SPEC and commercial), including both single-threaded and multi-threaded applications. As the table shows, some of the commercial applications are an order of magnitude larger and longer-running than SPEC, and fully simulating them would take years. Simulating only the selected PinPoints reduces the simulation time from years to days. We also validate that the regions chosen represent whole-program behavior (e.g., the cycles-per-instruction predicted by PinPoints is typically within 10% of the actual value [23]). Because of its high prediction accuracy, fast simulation time, and ease-of-use, PinPoints is now used to predict performance of large applications on future Intel processors.

Instruction Type	Count		
	Compiler A	Compiler B	Delta
*total	712M	618M	-94M
XORL	94M	94M	0M
TESTQ	94M	94M	0M
RET	94M	94M	0M
PUSHQ	94M	0M	-94M <-
POPQ	94M	0M	-94M <-
JE	94M	0M	-94M <-
LEAQ	37M	37M	0M
JNZ	37M	131M	94M <-
ANDQ	37M	37M	0M
ADDL	37M	37M	0M
MOVL	0M	94M	94M <-

Table 3. Dynamic instruction distribution in `PopCnt()` of `crafty` benchmark.

```
42f538 <PopCnt>:
42f538: push %rsi # unnecessary
42f539: xor %eax,%eax
42f53b: test %rdi,%rdi
42f53e: je 42f54c
42f540: add $0x1,%eax
42f543: lea 0xffffffffffffffff(%rdi),%rdx
42f547: and %rdx,%rdi
42f54a: jne 42f540
42f54c: pop %rcx  # unnecessary
42f54d: retq
```

Figure 8. `PopCnt()` code generated by compiler A.

6. Related Work

There is a large body of related work in the areas of instrumentation and dynamic compilation. To limit our scope of discussion, we concentrate on *binary instrumentation* in this section. At the highest level, instrumentation consists of *static* and *dynamic* approaches.

Static binary instrumentation was pioneered by ATOM [30], followed by others such as EEL [18], Etch [25], and Morph [31]. Static instrumentation has many limitations compared to dynamic instrumentation. The most serious one is that it is possible to mix code and data in an executable and a static tool may not have enough information to distinguish the two. Dynamic tools can rely on execution to discover all the code at run time. Other difficult

Program	Description	Code Size (MB)	Dynamic Count (billions)
SPECINT 2000	SPEC CPU2000 integer suite [11]	1.9 (avg.)	521
SPECFP 2000	SPEC CPU2000 floating -point suite [11]	2.4 (avg.)	724
SPECOMP 2001	SPEC benchmarks for evaluating *multithreaded* OpenMP applications [26]	8.4	4800
Amber	A suite of bio-molecular simulation from UCSF [1]	6.2	3994
Fluent	Computational Fluid Dynamics code from Fluent Inc [2]	19.6	25406
LsDyna	A general-purpose transient dynamic finite element analysis program from Livermore Software Technology [3]	61.9	4932
RenderMan	A photo-realistic rendering application from Pixar [4]	8.5	797

Table 4. Applications analyzed with PinPoints. Column 3 shows the code section size of the application binary and shared libraries reported by the `size` command. Column 4 lists the dynamic instruction count for the longest-running application input.

problems for static systems are indirect branches, shared libraries, and dynamically-generated code.

There are two approaches to dynamic instrumentation: *probe-based* and *jit-based*. The probe-based approach works by dynamically replacing instructions in the original program with trampolines that branch to the instrumentation code. Example probe-based systems include Dyninst [7], Vulcan [29], and DTrace [9]. The drawbacks of probe-based systems are that (i) instrumentation is *not* transparent because original instructions in memory are overwritten by trampolines, (ii) on architectures where instruction sizes vary (i.e. x86), we cannot replace an instruction by a trampoline that occupies more bytes than the instruction itself because it will overwrite the following instruction, and (iii) trampolines are implemented by one or more levels of branches, which can incur a significant performance overhead. These drawbacks make *fine-grained* instrumentation challenging on probe-based systems. In contrast, the jit-based approach is more suitable for fine-grained instrumentation as it works by dynamically compiling the binary and can insert instrumentation code (or calls to it) anywhere in the binary. Examples include Valgrind [22], Strata [27], DynamoRIO [6], Diota [21], and Pin itself. Among these systems, Pin is unique in the way that it supports high-level, easy-to-use instrumentation, which at the same time is portable across four architectures and is efficient due to optimizations applied by our JIT.

7. Conclusions

We have presented Pin, a system that provides easy-to-use, portable, transparent, efficient, and robust instrumentation. It supports the IA32, EM64T, Itanium®, and ARM architectures running Linux. We show that by abstracting away architecture-specific details, many Pintools can work across the four architectures with little porting effort. We also show that the Pin's high-level, call-based instrumentation API does not compromise performance. Automatic optimizations done by our JIT compiler make Pin's instrumentation even more efficient than other tools that use low-level APIs. We also demonstrate the versatility of Pin with two Pintools, `Opcodemix` and `PinPoints`. Future work includes developing novel Pintools, enriching and refining the instrumentation API as more tools are developed, and porting Pin to other operating systems. Pin is freely available at `http://rogue.colorado.edu/Pin`.

Acknowledgments

We thank Prof. Dan Connors for hosting the Pin website at University of Colorado. The Intel Bistro team provided the Falcon decoder/encoder and suggested the instruction scheduling optimization. Mark Charney developed the XED decoder/encoder. Ramesh Peri implemented part of the Pin 2/Itanium instrumentation.

References

[1] *AMBER home page*. http://amber.scripps.edu/.

[2] *Fluent home page*. http://www.fluent.com/.

[3] *LS-DYNA home page*. http://www.lstc.com/.

[4] *RenderMan home page*. http://RenderMan.pixar.com/.

[5] A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal*, 7(1):19–31, Feb 2003.

[6] D. L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, M.I.T. (http://www.cag.lcs.mit.edu/dynamorio/), September 2004.

[7] B. R. Buck and J. Hollingsworth. An api for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[8] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno dynamic optimizing compiler for java. In *ACM Java Grande Conference*, pages 129–141, June 1999.

[9] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[10] K.D. Cooper, M.W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2), April 1993.

[11] J. L. Henning. SPEC CPU2000: measuring cpu performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[12] Intel. *Pin User Manual*. http://rogue.colorado.edu/Pin.

[13] Intel. *Intel Itanium Architecture Software Developer's Manual Vols 1-4*, Oct. 2002.

[14] Intel. *IA-32 Intel Architecture Software Developer's Manual Vols 1-3*, 2003.

[15] Intel. *Intel Extended Memory 64 Technology Software Developer's Guide Vols 1-2*, 2004.

[16] Intel. *Intel PXA27x Processor Family Developer's Manual*, April 2004.

[17] H.-S. Kim and J. Smith. Hardware support for control transfers in code caches. In *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*, Dec 2003.

[18] J. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN 95 Conference on Programming Language Design and Implementation*, pages 291–300, June 1995.

[19] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. The Multiflow Trace Scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, 1993.

[20] Chi-Keung Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. Ispike: A Post-link Optimizer for the Intel Itanium Architecture. In *Proceedings of the 2nd Conference on Code Generation and Optimization*, pages 15–26, 2004.

[21] J. Maebe, M. Ronsse, and K. De Bosschere. Diota: Dynamic instrumentation, optimization and transformation of applications. In *Compendium of Workshops and Tutorials held in conjunction with PACT'02*, 2002.

[22] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Proceedings of the 3rd Workshop on Runtime Verification*. http://valgrind.kde.org/, 2003.

[23] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel itanium progams

with dynamic instrumentation. In *Proceedings of the 37th Annual ACM/IEEE International Symposium on Microarchitecture*, Dec 2004.

[24] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions. on Programming Languages and Systems*, 21(5):895–913, Sept 1999.

[25] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of win32/intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, pages 1–7, August 1997.

[26] H. Saito, G. Gaertner, W. Jones, R. Eigenmann, H. Iwashita, R. Liberman, M. van Waveren, and B. Whitney. Large system performance of spec omp2001 benchmarks. In *Proceedings of the 2002 Workship on OpenMP: Experiences and Implementation*, 2002.

[27] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. L. Soffa. Reconfigurable and retargetable software dynamic translation. In *Proceedings of the 1st Conference on Code Generation and Optimization*, pages 36–47, 2003.

[28] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 2002.

[29] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, April 2001.

[30] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[31] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System support for automatic profiling and optimization. In *Proceedings of the 16th Symposium on Operating System Principles*, October 1997.

Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation

Nicholas Nethercote
National ICT Australia, Melbourne, Australia
njn@csse.unimelb.edu.au

Julian Seward
OpenWorks LLP, Cambridge, UK
julian@open-works.co.uk

Abstract

Dynamic binary instrumentation (DBI) frameworks make it easy to build dynamic binary analysis (DBA) tools such as checkers and profilers. Much of the focus on DBI frameworks has been on performance; little attention has been paid to their capabilities. As a result, we believe the potential of DBI has not been fully exploited.

In this paper we describe Valgrind, a DBI framework designed for building heavyweight DBA tools. We focus on its unique support for *shadow values*—a powerful but previously little-studied and difficult-to-implement DBA technique, which requires a tool to shadow every register and memory value with another value that describes it. This support accounts for several crucial design features that distinguish Valgrind from other DBI frameworks. Because of these features, lightweight tools built with Valgrind run comparatively slowly, but Valgrind can be used to build more interesting, heavyweight tools that are difficult or impossible to build with other DBI frameworks such as Pin and DynamoRIO.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—debugging aids, monitors; D.3.4 [*Programming Languages*]: Processors—incremental compilers

General Terms Design, Performance, Experimentation

Keywords Valgrind, Memcheck, dynamic binary instrumentation, dynamic binary analysis, shadow values

1. Introduction

Valgrind is a dynamic binary instrumentation (DBI) framework that occupies a unique part of the DBI framework design space. This paper describes how it works, and how it differs from other frameworks.

1.1 Dynamic Binary Analysis and Instrumentation

Many programmers use program analysis tools, such as error checkers and profilers, to improve the quality of their software. *Dynamic binary analysis* (DBA) tools are one such class of tools; they analyse programs at run-time at the level of machine code.

DBA tools are often implemented using *dynamic binary instrumentation* (DBI), whereby the *analysis code* is added to the original code of the *client program* at run-time. This is convenient for users, as no preparation (such as recompiling or relinking) is needed. Also, it gives 100% instrumentation coverage of user-mode code, without requiring source code. Several generic *DBI frameworks* exist, such as Pin [11], DynamoRIO [3], and Valgrind [18, 15]. They provide a base system that can instrument and run code, plus an environment for writing tools that plug into the base system.

The performance of DBI frameworks has been studied closely [1, 2, 9]. Less attention has been paid to their instrumentation capabilities, and the tools built with them. This is a shame, as it is the tools that make DBI frameworks useful, and complex tools are more interesting than simple tools. As a result, we believe the potential of DBI has not been fully exploited.

1.2 Shadow Value Tools and Heavyweight DBA

One interesting group of DBA tools are those that use *shadow values*. These tools shadow, purely in software, every register and memory value with another value that says something about it. We call these *shadow value tools*. Consider the following motivating list of shadow value tools; the descriptions are brief but demonstrate that shadow values (a) can be used in a wide variety of ways, and (b) are powerful and interesting.

Memcheck [25] uses shadow values to track which bit values are undefined (i.e. uninitialised, or derived from undefined values) and can thus detect dangerous uses of undefined values. It is used by thousands of C and C++ programmers, and is probably the most widely-used DBA tool in existence.[1]

TaintCheck [20] tracks which byte values are tainted (i.e. from an untrusted source, or derived from tainted values) and can thus detect dangerous uses of tainted values. *TaintTrace* [6] and *LIFT* [23] are similar tools.

McCamant and Ernst's secret-tracking tool [13] tracks which bit values are secret (e.g. passwords), and determines how much information about secret inputs is revealed by public outputs.

Hobbes [4] tracks each value's type (determined from operations performed on the value) and can thus detect subsequent operations inappropriate for a value of that type.

DynCompB [7] similarly determines abstract types of byte values, for program comprehension and invariant detection purposes.

Annelid [16] tracks which word values are array pointers, and from this can detect bounds errors.

Redux [17] creates a *dynamic dataflow graph*, a visualisation of a program's entire computation; from the graph one can see all the prior operations that contributed to the each value's creation.

In these tools each shadow value records a simple approximation of each value's history—e.g. one shadow bit per bit, one

[1] Purify [8] is a memory-checking tool similar to Memcheck. However, Purify is not a shadow value tool as it does not does not track definedness of values through registers. As a result, it detects undefined value errors less accurately than Memcheck.

PLDI'07 June 11–13, 2007, San Diego, California, USA.

shadow byte per byte, or one shadow word per word—which the tool uses in a useful way; in four of the above seven cases, the tool detects operations on values that indicate a likely program defect.

Shadow value tools are a perfect example of what we call "heavyweight" DBA tools. They involve large amounts of analysis data that is accessed and updated in irregular patterns. They instrument many operations (instructions and system calls) in a variety of ways—for example, loads, adds, shifts, integer and FP operations, and allocations and deallocations are all handled differently. For heavyweight tools, *the structure and maintenance of the tool's analysis data is comparably complex to that of the client program's original data*. In other words, a heavyweight tool's execution is as complex as the client program's. In comparison, more lightweight tools such as trace collectors and profilers add a lot of highly uniform analysis code that updates analysis data in much simpler ways (e.g. appending events to a trace, or incrementing counters).

Shadow value tools are powerful, but difficult to implement. Most existing ones have slow-down factors of 10x–100x or even more, which is high but bearable if they are sufficiently useful. Some are faster, but applicable in more limited circumstances, as we will see.

1.3 Contributions

This paper makes the following contributions.

- ***Characterises shadow value tools.*** Tools using shadow values are not new, but the similarities they share have received little attention. This introduction has identified these similarities, and Section 2 formalises them by specifying the requirements of shadow value tools in detail.
- ***Shows how to support shadow values in a DBI framework.*** Section 3 describes how Valgrind works, emphasising its features that support robust heavyweight tools, such as its code representation, its first-class shadow registers, its events system, and its handling of threaded programs. This section does not delve deeply into well-studied topics, such as code cache management and trace formation, that do not relate to shadow values and instrumentation capabilities. Section 4 then shows how Valgrind supports each of the shadow value requirements from Section 2.[2]
- ***Shows that DBI frameworks are not all alike.*** Section 5 evaluates Valgrind's ease-of-tool-writing, robustness, instrumentation capabilities and performance. It involves some detailed comparisons between Valgrind and Pin, and between Memcheck and various other shadow value tools. Section 6 discusses additional related work. These two sections, along with some details from earlier parts of the paper—especially Section 3.5's novel identification of two basic code representations (disassemble-and-resynthesise vs. copy-and-annotate) for DBI—show that different DBI frameworks have different strengths and weaknesses. In particular, lightweight tools built with Valgrind run comparatively slowly, but Valgrind can be used to build more interesting, robust, heavyweight tools that are difficult or impossible to build with other DBI frameworks such as Pin and DynamoRIO.

These contributions show that there is great potential for new DBA tools that help programmers improve their programs, and that Valgrind provides a good platform for building these tools. At the paper's end, Section 7 describes future work and concludes.

[2] Two prior publications [18, 15] described earlier versions of Valgrind. However, they discussed shadow values in much less detail, and most of Valgrind's internals have changed since they were published: the old x86-specific JIT compiler has been replaced, its basic structure and start-up sequence has changed, its handling of threads, system calls, signals, and self-modifying code has improved, and function wrapping has been added.

2. Shadow Value Requirements

This section describes what a tool must do to support shadow values. We start here because (a) it shows that these requirements are generic and not tied to Valgrind, and (b) knowledge of shadow values is crucial to understanding how Valgrind differs from other DBI frameworks. Not until Sections 3 and 4 will we describe Valgrind and show how it supports these requirements. Then in Sections 5 and 6 we will explain in detail how Valgrind's support for these requirements is unique among DBI frameworks.

There are three characteristics of program execution that are relevant to shadow value tools: (a) programs maintain state, S, a finite set of *locations* that can hold values (e.g. registers and the user-mode address space), (b) programs execute operations that read and write S, and (c) programs execute operations (allocations and deallocations) that make memory locations active or inactive. We group the nine shadow value requirements accordingly.

Shadow State. A shadow value tool maintains a shadow state, S', which contains a shadow value for every value in S.

- ***R1: Provide shadow registers.*** A shadow value tool must manipulate shadow register values (integer, FP and SIMD) from S' just like normal register values, in which case it must multiplex two sets of register values—original and shadow—onto the machine's register file, without perturbing execution.
- ***R2: Provide shadow memory.*** S' must hold shadow values for all memory locations in S. To do this a shadow value tool must partition the address space between the original memory state and the shadow memory state. It also must access shadow memory safely in the presence of multiple threads.

Read and write operations. A shadow value tool must instrument some or all operations that read/write S with shadow operations that read/write S'.

- ***R3: Instrument read/write instructions.*** Most instructions access registers and many access memory. A shadow value tool must instrument some or all of them appropriately, and so must know which locations are accessed by every one of the many (hundreds of) distinct instructions, preferably in a way that is portable across different instruction sets.
- ***R4: Instrument read/write system calls.*** All system calls access registers and/or memory: they read their arguments from registers and/or the stack, and they write their return value to a register or memory location. Many system calls also access user-mode memory via pointer arguments. A shadow value tool must instrument some or all of these accesses appropriately, and so must know which locations are accessed by every one of the many (hundreds of) different system calls.

Allocation and deallocation operations. A shadow value tool may instrument some or all allocation and deallocation operations with shadow operations that update S' appropriately.

- ***R5: Instrument start-up allocations.*** At program start-up, all the registers are "allocated", as are statically allocated memory locations. A shadow value tool must create suitable shadow values for these locations. It must also create suitable shadow values for memory locations not allocated at this time (in case they are later accessed erroneously before being allocated).
- ***R6: Instrument system call (de)allocations.*** Some system calls allocate memory (e.g. `brk`, `mmap`), and some deallocate memory

(e.g. `munmap`), and again some shadow value tools must instrument these operations. Also, `mremap` can cause memory values to be copied, in which case the corresponding shadow memory values may have to be copied as well.

- ***R7: Instrument stack (de)allocations.*** Stack pointer updates also allocate and deallocate memory, and some shadow value tools must instrument these operations. This can be expensive because stack pointer updates are so frequent. Also, some programs switch between multiple stacks. Some shadow value tools need to distinguish these from large stack allocations or deallocations, which can be difficult at the binary level.
- ***R8: Instrument heap (de)allocations.*** Most programs use a heap allocator from a library that hands out heap blocks from larger chunks allocated with a system call (`brk` and/or `mmap`). Each heap block typically has book-keeping data attached (e.g. the block size) which the client program should not access (reading it may be safe, but overwriting it may crash the allocator). Thus there is a notion of library-level addressability which overlays the kernel-level addressability.

  Therefore, a shadow value tool may need to also track heap allocations and deallocations, and consider the book-keeping data as not active. It should also treat the heap operations as atomic, ignoring the underlying kernel-level allocations of large chunks, instead waiting until the allocated bytes are handed to the client by the allocator before considering them to be active. Also, `realloc` needs to be handled the same way as `mremap`.

Transparent execution, but with extra output. We assume that shadow value tools do not affect the client's behaviour other than producing auxiliary output. This leads to our final requirement.

- ***R9: Extra output.*** A shadow value tool must use a side-channel for its output, such as a little-used file descriptor (e.g. `stderr`) or a file. No other functional perturbation should occur.

Summary. These nine requirements are difficult to implement correctly. Clearly, tools that do these tasks purely in software will be slow if not implemented carefully.

One thing to note about these requirements: shadow value tools are among the most heavyweight of DBA tools, and most DBA tools involve a subset of these requirements (for example, almost every DBA tool involves R9). Therefore, *a DBI framework that supports shadow values well will also support most conceivable DBA tools*.

Now that we know *what* shadow value tools do, we can describe Valgrind, paying particular attention to its support for the nine shadow value requirements. In Sections 5 and 6, we will see that other DBI frameworks do not support shadow values as well as Valgrind does.

3. How Valgrind Works

Valgrind is a DBI framework designed for building heavyweight DBA tools. It was first released in 2002. The Valgrind distribution [28] contains four tools, the most popular of which is Memcheck. Valgrind has also been used to build several experimental tools. It is available under the GNU General Public License (GPL), and runs on x86/Linux, AMD64/Linux, and PPC{32,64}/{Linux,AIX}.

3.1 Basic Architecture

Valgrind tools are created as plug-ins, written in C, to Valgrind's *core*. The basic view is: ***Valgrind core + tool plug-in = Valgrind tool.*** A tool plug-in's main task is to instrument code fragments that the core passes to it. Writing a new tool plug-in (and thus a new Valgrind tool) is much easier than writing a new DBA tool from scratch. Valgrind's core does most of the work, and also provides services to make common tool tasks such as recording errors easier.

3.2 Execution Overview

Valgrind uses *dynamic binary re-compilation*, similar to many other DBI frameworks. A Valgrind tool is invoked by adding `valgrind --tool=<toolname>` (plus any Valgrind or tool options) before a command. The named tool starts up, loads the client program into the same process, and then (re)compiles the client's machine code, one small code block at a time, in a just-in-time, execution-driven fashion. The core disassembles the code block into an intermediate representation (IR) which is instrumented with analysis code by the tool plug-in, and then converted by the core back into machine code. The resulting *translation* is stored in a code cache to be rerun as necessary. Valgrind's core spends most of its time making, finding, and running translations. None of the client's original code is run.

Code handled correctly includes: normal executable code, dynamically linked libraries, shared libraries, and dynamically generated code. Only self-modifying code can cause problems (see Section 3.16). The only code not under a tool's control is that within system calls, but system call side-effects can be indirectly observed, as Section 3.12 will show.

Many complications arise from squeezing two programs—the client and the tool—into a single process. They must share many resources such as registers and memory. Also, Valgrind must be careful not to relinquish its control over the client in the presence of system calls, signals and threads, as we will see.

3.3 Starting Up

The goal of start-up is to load Valgrind's core, the tool, and the client into a single process, sharing the same address space.

Each tool is a statically-linked executable that contains the tool code plus the core code. Having one copy of the core for every tool wastes a little disk space (the core is about 2.5MB), but makes things simple. The executable is linked to load at a non-standard address that is usually free at program start-up (on x86/Linux it is `0x38000000`). If this address is not free—an exceptionally rare case, in our experience—Valgrind can be recompiled to use a different address.

The `valgrind` executable invoked by the user is a small wrapper program that scans its command-line for a `--tool` option, and then loads the selected tool's static executable using `execve`.

Valgrind's core first initialises some sub-systems, such as the the address space manager and its own internal memory allocator. It then loads the client executable (text and data), which can be an ELF executable or a script (in which case the script interpreter is loaded). It then sets up the client's stack and data segment.

The core then tells the tool to initialise itself. The command-line is parsed and core and tool options are dealt with. Finally, more core sub-systems are initialised: the translation table, the signal-handling machinery, the thread scheduler, and debug information for the client is loaded. At this point, the Valgrind tool is in complete control and everything is in place to begin translating and executing the client from its first instruction.

This is the third structure and start-up approach that has been used for Valgrind, and is by far the most reliable. The first one [18] used the dynamic linker's `LD_PRELOAD` mechanism to inject Valgrind's core and the tool (both built as shared objects) into the client. This did not work with statically compiled executables, allowed some client code to run natively before Valgrind gained control, and was not widely portable. The second one [15] was similar to the current approach, but required the use of large empty memory mappings to force components into the right place, which turned out to be somewhat unreliable.

Most DBI frameworks use injection-style methods rather than having their own program loader. As well as avoiding the problems encountered by the prior two approaches, our third approach has two other advantages. First, it gives Valgrind great control over memory layout. Second, it it avoids dependencies on other tools such as the dynamic linker, which we have found to be an excellent strategy for improving robustness.[3]

3.4 Guest and Host Registers

Valgrind itself runs on the machine's real or *host* CPU, and (conceptually) runs the client program on a simulated or *guest* CPU. We refer to registers in the host CPU as *host registers* and those of the simulated CPU as *guest registers*. Due to the dynamic binary recompilation process, a guest register's value may reside in one of the host's registers, or it may be spilled to memory for a variety of reasons. Shadow registers are shadows of guest registers.

Valgrind provides a block of memory per client thread called the ThreadState. Each one contains space for all the thread's guest and shadow registers and is used to hold them at various times, in particular between each code block. Storing guest registers in memory between code blocks sounds like a bad idea at first, because it means that they must be moved between memory and the host registers frequently, but it is reasonable for heavyweight tools with high host register pressure for which the benefits of a more optimistic strategy are greatly diminished.

3.5 Representation of code: D&R vs. C&A

There are two fundamental ways for a DBI framework to represent code and allow instrumentation.

Valgrind uses *disassemble-and-resynthesise (D&R)*: machine code is converted to an IR in which each instruction becomes one or more IR operations. This IR is instrumented (by adding more IR) and then converted back to machine code. All of the original code's effects on guest state (e.g. condition code setting) must be explicitly represented in the IR because the original client instructions are discarded and the final code is generated purely from the IR. Valgrind's use of D&R is the single feature that most distinguishes it from other DBI frameworks.

Other DBI frameworks use *copy-and-annotate (C&A)*: incoming instructions are copied through verbatim except for necessary control flow changes. Each instruction is annotated with a description of its effects, via data structures (e.g. DynamoRIO) or an instruction-querying API (e.g. Pin). Tools use the annotations to guide their instrumentation. The added analysis code must must be interleaved with the original code without perturbing its effects.

Hybrid approaches are possible. For example, earlier versions of Valgrind used D&R for integer instructions and C&A for FP and SIMD instructions (this was more by accident than design). Variations are also possible; for example, DynamoRIO allows instruction bytes to be modified in-place before being copied through.

Each approach has its pros and cons, depending on the instrumentation requirements. D&R may require more up-front design and implementation effort, because a D&R representation is arguably more complex. Also, generating good code at the end requires more development effort—Valgrind's JIT uses a lot of conventional compiler technology. In contrast, for C&A, good client code stays good with less effort. A D&R JIT compiler will probably also translate code more slowly.

D&R may not suitable for some tools that require low-level information. For example, the exact opcode used by each instruction may be lost. IR annotations can help, however—for example, Valgrind has "marker" statements that indicate the boundaries, addresses and lengths of original instructions. C&A can suffer the same problem if the annotations are not comprehensive.

D&R's strengths emerge when complex analysis code must be added. First, D&R's use of the same IR for both client and analysis code guarantees that analysis code is as expressive and powerful as client code. Making all side-effects explicit (e.g. condition code computations) can make instrumentation easier.

The performance dynamics also change. The JIT compiler can optimise analysis code and client code equally well, and naturally tightly interleaves the two. In contrast, C&A must provide a separate way to describe analysis code (so C&A requires some kind of IR after all). This code must then be fitted around the original instructions, which requires effort (either by the framework or the tool-writer) to do safely and with good performance. For example, Pin analysis code is written as C functions (i.e. the analysis code IR is C), which are compiled with an external C compiler, and Pin then inlines them if possible, or inserts calls to them.

Finally, D&R is more verifiable—any error converting machine code to IR is likely to cause visibly wrong behaviour, whereas a C&A annotation error will result in incorrect analysis of a correctly behaving client.[4] D&R also permits binary translation from one platform to another (although Valgrind does not do this). D&R also allows the original code's behaviour to be arbitrarily changed.

In summary, D&R requires more effort up-front and is overkill for lightweight instrumentation. However, it naturally supports heavyweight instrumentation such as that required by shadow value tools, and so is a natural fit for Valgrind.

3.6 Valgrind's IR

Prior to version 3.0.0 (August 2005), Valgrind had an x86-specific, part D&R, part C&A, assembly-code-like IR in which the units of translation were basic blocks. Since then Valgrind has had an architecture-neutral, D&R, single-static-assignment (SSA) IR that is more similar to what might be used in a compiler. IR blocks are *superblocks*: single-entry, multiple-exit stretches of code.

Each IR block contains a list of statements, which are operations with side-effects, such as register writes, memory stores, and assignments to temporaries. Statements contain expressions, which represent pure (no side effects) values such as constants, register reads, memory loads, and arithmetic operations. For example, a store statement contains one expression for the store address and another for the store value. Expressions can be arbitrarily complicated trees (*tree IR*), but they can also be flattened by introducing statements that write intermediate values to temporaries (*flat IR*).

The IR has some RISC-like features: it is load/store, each primitive operation only does one thing (many CISC instructions are broken up into multiple operations), and when flattened, all operations operate only on temporaries and literals. Nonetheless, supporting all the standard integer, FP and SIMD operations of different sizes requires more than 200 primitive arithmetic/logical operations.

The IR is architecture-independent. Valgrind handles unusual architecture-specific instructions, such as `cpuid` on x86, with a call to a C function that emulates the instruction. These calls have annotations that say which guest registers and memory locations they access, so that a tool can see some of their effects while avoiding the need for Valgrind to represent the instruction explicitly in the IR. This is another case (like the "marker" statements) where Valgrind uses IR annotations to facilitate instrumentation (but it is not C&A, because the instruction is emulated, not copied through).

[3] For example, Valgrind no longer uses the standard C library, but has a small version of its own. This has avoided any potential complications caused by having two copies of the C library in the address space—one for the client, and and for Valgrind and the tool. It also made the AIX port much easier, because AIX's C library is substantially different to Linux's.

[4] This is not just a theoretical concern. Valgrind's old IR used C&A for SIMD instructions; some SIMD loads were mis-annotated as stores, and some SIMD stores as loads, for more than a year before being noticed.

```
0x24F275:  movl -16180(%ebx,%eax,4),%eax
 1: ------ IMark(0x24F275, 7) ------
 2: t0 = Add32(Add32(GET:I32(12),# get %ebx and
      Shl32(GET:I32(0),0x2:I8)), # %eax, and
      0xFFFFC0CC:I32)            # compute addr
 3: PUT(0) = LDle:I32(t0)        # put %eax

0x24F27C:  addl %ebx,%eax
 4: ------ IMark(0x24F27C, 2) ------
 5: PUT(60) = 0x24F27C:I32       # put %eip
 6: t3 = GET:I32(0)              # get %eax
 7: t2 = GET:I32(12)             # get %ebx
 8: t1 = Add32(t3,t2)            # addl
 9: PUT(32) = 0x3:I32            # put eflags val1
10: PUT(36) = t3                 # put eflags val2
11: PUT(40) = t2                 # put eflags val3
12: PUT(44) = 0x0:I32            # put eflags val4
13: PUT(0) = t1                  # put %eax

0x24F27E:  jmp*l %eax
14: ------ IMark(0x24F27E, 2) ------
15: PUT(60) = 0x24F27E:I32       # put %eip
16: t4 = GET:I32(0)              # get %eax
17: goto {Boring} t4
```

Figure 1. Disassembly: machine code → tree IR

3.7 Translating a Single Code Block

Valgrind translates code blocks on demand. To create a translation of a code block, Valgrind follows instructions until one of the following conditions is met: (a) an instruction limit is reached (about 50, depending on the architecture), (b) a conditional branch is hit, (c) a branch to an unknown target is hit, or (d) more than three unconditional branches to known targets have been hit. This policy is less sophisticated than those used by frameworks like Pin and DynamoRIO; in particular, Valgrind does not recompile hot code.

There are eight translation phases. This high number is a consequence of Valgrind using D&R. They are described by the following paragraphs. All phases are performed by the core, except instrumentation, which is performed by the tool. Phases marked with a '*' are architecture-specific.

Phase 1. Disassembly*: machine code → tree IR. The disassembler converts machine code into (unoptimised) tree IR. Each instruction is disassembled independently into one or more statements. These statements fully update the affected guest registers in memory: guest registers are pulled from the ThreadState into temporaries, operated on, and then written back.

Figure 1 gives an example for x86 machine code. Three x86 instructions are disassembled into 17 tree IR statements.

- Statements 1, 4 and 14 are `IMarks`: no-ops that indicate where an instruction started, its address and length in bytes. These are used by profiling tools that need to see instruction boundaries.
- Statement 2 assigns an expression tree to a temporary `t0`; it shows how a CISC instruction can become multiple operations in the IR. `GET:I32` fetches a 32-bit integer guest register from the ThreadState; the offsets 12 and 0 are for guest registers `%ebx` and `%eax`. `Add32` is a 32-bit add, `Shl32` is a 32-bit left-shift. Statement 16 is a simpler assignment.
- Statement 3 writes a guest register (`%eax`) value back to its slot in the ThreadState (the `LDle` is a little-endian load). Statements 5 and 15 update the guest program counter (`%eip`) in the ThreadState.
- Statements 9–12 write four values to the ThreadState. Many x86 instructions affect the condition codes (`%eflags`), and Valgrind computes them from these four values when they are used. Often `%eflags` is clobbered without being used, so most of these `PUT`s can be optimised away later. DBI frameworks that use C&A do not synthesise the condition codes like this, but instead obtain them "for free" as a side-effect of running the code. But when heavyweight analysis code is added they must be saved and restored frequently, which involves expensive instructions on x86. In contrast, Valgrind's approach is more costly to begin with, but does not degrade badly in such cases. Also, knowing precisely the operation and operands most recently used to set the condition codes is helpful for some tools. For example, Memcheck's definedness tracking of condition codes was less accurate with with Valgrind's old IR, which used C&A for `%eflags`.
- Statement 17 is an unconditional jump to the address in `t4`.

Phase 2. Optimisation 1: tree IR → flat IR. The first optimisation phase flattens the IR and does several optimisations: redundant get and put elimination (to remove unnecessary copying of guest registers to/from the ThreadState), copy and constant propagation, constant folding, dead code removal, common sub-expression elimination, and even simple loop unrolling for intra-block loops. It is also possible to pass in callback functions that can partially evaluate certain platform-specific C helper calls. On x86 and AMD64 this is used to optimise the `%eflags` handling.

This phase updates the IR shown in Figure 1 in several ways.

- The complex expression tree in statement 2 is flattened into five assignments to temporaries: two using `GET`, two using `Add32`, one using `Shl32`.
- Statement 3 is changed from a `PUT` to an assignment to a temporary; this is possible because the `PUT` is made redundant by the `PUT` in statement 13.
- Statement 5 is removed. This is possible because statement 15 writes a new value for `%eip` and there are no intervening statements that could cause a memory exception (if there were, it could not be removed because a guest signal handler that inspects the `%eip` value in the ThreadState could be invoked).
- Statements 6, 7 and 16 are removed, because they are made redundant by the `GET` statements introduced by the flattening of statement 2.

Phase 3. Instrumentation: flat IR → flat IR. The code block is then passed to the tool, which can transform it arbitrarily. It is important that the IR is flattened at this point as it makes instrumentation easier, particularly for shadow value tools.

Figure 2 shows IR for the `movl` instruction from Figure 1 after it has been instrumented by Memcheck. Memcheck's shadow values track the definedness of values; its instrumentation has been described previously [25] and the details are beyond the scope of this paper. However, we make the following observations.

- Of the 18 statements, 11 were added by Memcheck—the added analysis code is larger and more complex than the original code.
- Shadow registers are stored in the ThreadState just like guest registers. For example, guest register `%eax` is stored at offset 0 in the ThreadState, and its shadow is stored at offset 320.
- Every operation involving guest values is preceded by a corresponding operation on shadow values.
- In some cases the shadow operation is a single statement, e.g. statements 2, 4 and 6. Even without understanding how Memcheck works it is easy to see what they are doing. For ex-

```
* 1: ------ IMark(0x24F275, 7) ------
  2: t11 = GET:I32(320)          #  get sh(%eax)
* 3: t8  = GET:I32(0)            # *get %eax
  4: t14 = Shl32(t11,0x2:I8)     #  shadow shll
* 5: t7  = Shl32(t8,0x2:I8)      # *shll
  6: t18 = GET:I32(332)          #  get sh(%ebx)
* 7: t9  = GET:I32(12)           # *get %ebx
  8: t19 = Or32(t18,t14)         #  shadow addl 1/3
  9: t20 = Neg32(t19)            #  shadow addl 2/3
 10: t21 = Or32(t19,t20)         #  shadow addl 3/3
*11: t6  = Add32(t9,t7)          # *addl
 12: t24 = Neg32(t21)            #  shadow addl 1/2
 13: t25 = Or32(t21,t24)         #  shadow addl 2/2
*14: t5  = Add32(t6,0xFFFFC0CC:I32) # *addl
 15: t27 = CmpNEZ32(t25)         #  shadow loadl 1/3
 16: DIRTY t27 RdFX-gst(16,4) RdFX-gst(60,4)
       ::: helperc_value_check4_fail{0x380035f4}()
                                 #  shadow loadl 2/3
 17: t29 = DIRTY 1:I1 RdFX-gst(16,4) RdFX-gst(60,4)
       ::: helperc_LOADV32le{0x38006504}(t5)
                                 #  shadow loadl 3/3
*18: t10 = LDle:I32(t5)          # *loadl
```

Figure 2. Instrumented flat IR. The statements that were present before instrumentation took place are prefixed with a '*'.

ample, when the original code GETs %eax from the ThreadState into a temporary, the analysis code GETs the shadow of %eax from the ThreadState into another temporary.

- In some cases the shadow operation is larger than the original operation, as seen in statements 8–10 and 12–13. The shadow load operation in statements 15–17 is larger still. Statement 15 tests the definedness of the pointer value by comparing its shadow value to zero, and statement 16 is a conditional call (conditional on the value in t27) to an error-reporting function that is only called if the test fails, i.e. if the load uses an address value that is not fully defined. (The DIRTY and RdFX annotations indicate that some guest registers are read from the ThreadState by the function, and so these values must be up-to-date. 0x380035f4 is the address of the called function.) Statement 17 calls another C function, helperc_LOADV32le, which does a shadow load to complement the original load in statement 18. The shadow load is implemented using a C function because it is too complex to be written inline [19].

Phase 4. Optimisation 2: flat IR → flat IR. A second, simpler optimisation pass performs constant folding and dead code removal. Figure 2 is a case in point—it actually shows the instrumented code after this second optimisation phase is run (which reduced it from 48 statements to 18). This optimisation makes life easier for tools by allowing them to be somewhat simple-minded, knowing that the code will be subsequently improved.

Phase 5. Tree building: flat IR → tree IR. The tree builder converts flat IR back to tree IR in preparation for instruction selection. Expressions assigned to temporaries which are used only once are usually substituted into the temporary's use point, and the assignment is deleted. The resulting code may perform loads in a different order to the original code, but loads are never moved past stores.

Phase 6. Instruction selection*: tree IR → instruction list. The instruction selector converts the tree IR into a list of instructions which use virtual registers (except for those instructions that are hard-wired to use particular registers; these are common on x86 and AMD64). The instruction selector uses a simple, greedy, top-down tree-matching algorithm.

```
-- t21 = Or32(t19,Neg32(t19))
movl %%vr19,%%vr41          movl %edx,%edi
negl %%vr41                 negl %edi
movl %%vr19,%%vr40
orl %%vr41,%%vr40           orl %edi,%edx
movl %%vr40,%%vr21
```

Figure 3. Register allocation, before and after. Virtual registers are named %%vrNN.

Phase 7. Register allocation: instruction list → instruction list. The linear-scan register allocator [26] replaces virtual registers with host registers, inserting spills as necessary. One general-purpose host register is always reserved to point to the ThreadState.

Although the instructions are platform-specific, the register allocator is platform-independent; it uses some callback functions to find out which registers are read and written by each instruction.

Figure 3 shows an example of register allocation. The statement at the top is created by the tree builder from statements 9 and 10 in Figure 2. The figure shows that the register allocator can remove many register-to-register moves, which makes life easier for the instruction selector.

Phase 8. Assembly*: instruction list → machine code. The final assembly phase simply encodes the selected instructions appropriately and writes them to a block of memory.

3.8 Storing Translations

Valgrind's code storage system is simple and warrants only a brief description. Translations are stored in the *translation table*, a fixed-size, linear-probe hash table. The translation table is large (about 400,000 entries) so it rarely gets full. If the table gets more than 80% full, translations are evicted in chunks, 1/8th of the table at a time, using a FIFO (first-in, first-out) policy—this was chosen over the more obvious LRU (least recently used) policy because it is simpler and it still does a fairly good job. Translations are also evicted when code in shared objects is unloaded (by munmap), or made obsolete by self-modifying code (see Section 3.16).

3.9 Executing Translations

Once a translation is made it can be executed. What happens between code blocks? Control flows from one translation to the next via one of two routes: the *dispatcher* (fast), or the *scheduler* (slow).

At a translation's end, control falls back to the dispatcher, a hand-crafted assembly code loop. At this point all guest registers are in the ThreadState. Only two host registers are live: one holds the guest program counter, and the other holds a value that is only used for unusual events, explained shortly, when control must fall back into the scheduler. The dispatcher looks for the appropriate translation in a small direct-mapped cache which holds addresses of recently-used translations. If that look-up succeeds (the hit-rate is around 98%), the translation is executed immediately. This fast case takes only fourteen instructions on x86.

When the fast look-up fails, control falls back to the scheduler, which is written in C. It searches the full translation table. If a translation is not found, a new translation is made. In either case, the direct-mapped cache is updated to store the translation address for the code block. The dispatcher is re-entered, and the fast direct-mapped look-up will this time definitely succeed.

There are certain unusual events upon which control falls back to the scheduler. For example, the core periodically checks whether a thread switch is due (see Section 3.14) or whether there are any outstanding signals to be handled (see Section 3.15). To support this, the dispatcher causes control to fall out to the scheduler every few thousand translation executions. Control is similarly returned

to the scheduler when system calls (see Section 3.10) and client requests (see Section 3.11) occur.

Valgrind does not perform *chaining* (also known as *linking*)—a technique that patches branch instructions in order to link translations directly, which avoids many visits to the dispatcher. Earlier versions did, but it has not yet been implemented in the new JIT compiler. The lack of chaining hurts Valgrind's speed less than for other DBI frameworks; we believe this is because Valgrind's dispatcher is fast,[5] and Valgrind chases across many unconditional branches.

3.10 System Calls

Valgrind cannot trace into the kernel. When a system call happens, control falls back into the scheduler, which: (a) saves the tool's stack pointer; (b) copies the guest registers into the host registers, except the program counter; (c) calls the system call; (d) copies the guest registers back out to memory, except the program counter; (e) restores the tool's stack pointer. Note that the system call is run on the client's stack, as it should be (the host stack pointer normally points to the tool's stack).

System calls involving partitioned resources such as memory (e.g. `mmap`) and file descriptors (e.g. `open`) are pre-checked to ensure they do not cause conflicts with the tool. For example, if the client tries to `mmap` memory currently used by the tool, Valgrind will make it fail without even consulting the kernel.

3.11 Client Requests

Valgrind's core has a simple trap-door mechanism that allows a client program to pass messages and queries, called *client requests*, to the core or a tool plug-in. Client requests are embedded in client programs using pre-defined macros from a header file provided by Valgrind. The mechanism is described in previous publications about Valgrind [18, 15] and so we omit the details here. We will see in Sections 3.12 and 3.16 examples of the use of client requests.

3.12 The Events System

Valgrind's IR is expressive, but fails to describe to tools certain changes to guest register and memory state done by clients. It also does not convey any details of memory allocations and deallocations. Valgrind provides an *events system* to describe such changes.

Let us first consider the accesses done by system calls. All system calls access registers: they read their arguments from registers and/or memory, and they write their return value to a register. Many system calls also access user-mode memory via pointer arguments, e.g. `settimeofday` is passed pointers to two structs which it reads from, and `gettimeofday` fills in two structs with data. Knowing which registers and memory locations are accessed by every system call is difficult because there are many system calls (around 300 for Linux), some of which have tens or hundreds of sub-cases, and there are many differences across platforms. Several things must be known for each system call: how many arguments it takes, each argument's size, which ones are pointers (and which of those can be `NULL`), which ones indicate buffer lengths, which ones are null-terminated strings, which ones are not read in certain cases (e.g. the third argument of `open` is only read if the second argument has certain values), and the sizes of various types (e.g. `struct timeval` used by `gettimeofday` and `settimeofday`).

Valgrind does not encode this information about system calls in its IR, because there are too many system calls and too much variation across platforms to do so cleanly. Instead it provides the events system to inform tools about register and memory accesses that are not directly visible from the IR. For each event, a tool can register a callback function to be called each time the event occurs. The events list is given in Table 1. A tool can use the `pre_*` events to know when system calls are about to read registers and memory locations, and the `post_*` events to know when to update the shadow state after system calls have written new values. The register events pass to their callbacks the size of the accessed register and its offset in the ThreadState; the memory events pass in the address and size of the accessed memory region.

How are these six events triggered? Valgrind provides a wrapper for every system call, which invokes these callbacks as needed. Every system call has different arguments and thus a different wrapper. Because there are so many cases, Valgrind's wrappers are almost 15,000 lines of tedious C code (in Valgrind 3.2.1), partly generic, partly platform-specific, aggregated over several years of development. In comparison, Memcheck is 10,509 lines of code. The wrappers save a great deal of work for tools that need to know about system call accesses, and also make the system call handling platform-independent for tools. No other DBI framework has such system call wrappers.

This mechanism is crucial for many shadow value tools. For example, Memcheck critically relies on it for its bit-precise definedness tracking. Indeed, several bugs in Valgrind's wrappers were found because they caused Memcheck to give false positives or false negatives.

A similar case involves stack allocations and deallocations. A tool could detect them just by detecting changes to the stack pointer from the IR. However, because it is a common requirement, Valgrind provides events (`new_mem_stack` and `die_mem_stack`) for these cases. The core instruments the code with calls to the event callbacks on the tool's behalf. This makes things easier for tools. It also provides a canned solution to a tricky part of the problem—as Section 2 noted, it is hard to distinguish large stack allocations and deallocations from stack-switches, but doing so is vital for some shadow value tools. Valgrind (and hence tools using the stack events) uses a heuristic: if the stack pointer changes by more than 2MB, a stack switch has occurred. The 2MB value is changeable with a command line option. Sometimes this heuristic is too crude, so Valgrind also provides three client requests which let the client register, de-register and resize stacks with Valgrind. So even in tricky cases, with a small amount of help from the programmer all stack switches can be detected.

The remaining events in Table 1 inform tools about allocations done at program start-up and via system calls.

3.13 Function Replacement and Function Wrapping

Valgrind supports *function replacement*, i.e. it allows a tool to replace any function in a program with an alternative function. A replacement function can also call the function it has replaced. This allows *function wrapping*, which is particularly useful for inspecting the arguments and return value of a function.

3.14 Threads

Threads pose a particular challenge for shadow value tools. The reason is that loads and stores become non-atomic: each load/store translates into the original load/store plus a shadow load/store. On a uni-processor machine, a thread switch might occur between these two operations. On a multi-processor machine, concurrent memory accesses to the same memory location may complete in a different order to their corresponding shadow memory accesses. It is unclear how to best deal with this, as a fine-grained locking approach would likely be slow.

To sidestep this problem, Valgrind serialises thread execution with a thread locking mechanism. Only the thread holding the lock can run, and threads drop the lock before they call a blocking

[5] In comparison, chaining improved Strata's basic slow-down factor from 22.1x to 4.1x, because dispatching takes about 250 cycles [24]. Valgrind's slow-down even without chaining is 4.3x.

Req.	Valgrind events	Called from	Memcheck callbacks
R4	`pre_reg_read`, `post_reg_write`	Every system call wrapper	`check_reg_is_defined`, `make_reg_defined`
	`pre_mem_read{,_asciiz}`	Many system call wrappers	`check_mem_is_defined{,_asciiz}`
	`pre_mem_write`, `post_mem_write`	Many system call wrappers	`check_mem_is_addressable`, `make_mem_defined`
R5	`new_mem_startup`	Valgrind's code loader	`make_mem_defined`
R6	`new_mem_mmap`, `die_mem_munmap`	`mmap` wrapper, `munmap` wrapper	`make_mem_defined`, `make_mem_noaccess`
	`new_mem_brk`, `die_mem_brk`	`brk` wrapper	`make_mem_undefined`, `make_mem_noaccess`
	`copy_mem_mremap`	`mremap` wrapper	`copy_range`
R7	`new_mem_stack`, `die_mem_stack`	Instrumentation of SP changes	`make_mem_undefined`, `make_mem_noaccess`

Table 1. Valgrind events, their trigger locations, and Memcheck's callbacks for handling them.

system call,[6] or after they have been running for a while (100,000 code blocks). The lock is implemented using a pipe which holds a single character; each thread tries to read the pipe, only one thread will be successful, and the others will block until the running thread relinquishes the lock by putting a character back in the pipe. Thus the kernel still chooses which thread is to run next, but Valgrind dictates when thread-switches occur and prevents more than one thread from running at a time.

This is the third thread serialisation mechanism that has been used in Valgrind, and is by far the most robust. The first one [18, 15] involved Valgrind providing a serialised version of the `libpthread` library. This only worked with programs using pthreads. It also caused many problems because on Linux systems, glibc and the pthreads library are tightly bound and interact in various ways "under the covers" that are difficult to replicate.[7] The second one was more like the current one, but was more complex, requiring extra kernel threads to cope with blocking I/O.

This serialisation is a unique Valgrind feature not shared by other DBI frameworks. It has both pros and cons: it means that Valgrind tools using shadow memory can ignore the atomicity issue. However, as multi-processor machines become more popular, the resulting performance shortcomings for multi-threaded programs will worsen. How to best overcome this problem remains an open research question.

3.15 Signals

Unix signal handling presents a problem for all DBI frameworks—when an application sets a signal handler, it is giving the kernel a callback (code) address in the application's space which will be used to deliver the signal. This would allow the client's original handler code to be executed natively. Even worse, if the handler did not return but instead did a `longjmp`, the tool would permanently lose control. Therefore, Valgrind intercepts all system calls that register signal handlers. It also catches all signals and delivers them appropriately to the client. This standard technique is tedious but unavoidable. Also, Valgrind takes advantage of it to ensure that asynchronous signals are delivered only between code blocks, and can thus never separate loads/stores from shadow loads/stores.

3.16 Self-modifying Code

Self-modifying code is always a challenge for DBI frameworks. On architectures such as PowerPC it is easy to detect because an explicit "flush" instruction must be used when code is modified, but the x86 and AMD64 architectures do not have this feature.

Therefore, Valgrind has a mechanism to handle self-modifying code. A code block using this mechanism records a hash of the original code it was derived from. Each time the block executes, the hash is recomputed and checked, and if it does not match, the block is discarded and the code retranslated.

This has a high run-time cost. Therefore, by default Valgrind only uses this mechanism for code that is on the stack. This is enough to handle the trampolines that some compilers (e.g. GCC) put on the stack when running nested functions, which we have found to be the main cause of self-modifying code.[8] This minimises the cost, as only code on the stack is slowed down. The mechanism can also be turned off altogether or turned on for every block.

Valgrind also provides another mechanism for handling self-modifying code—a client request which tells it to discard any translations of instructions in a certain address range. It is most useful for dynamic code generators such as JIT compilers.

4. Valgrind's Shadow Value Support

This section describes how the features described in the previous section support all nine shadow value requirements. Because these requirements are a superset of most DBA tools' requirements, Valgrind supports most conceivable DBA tools.

R1: Provide shadow registers. Valgrind has three noteworthy features that make shadow registers easy to use. First, ***shadow registers are first-class entities:*** (a) space is provided for them in the ThreadState, (b) they can be accessed just as easily as guest registers, (c) they can be manipulated and operated on in the same ways. This makes complex shadow operations code natural and easy to write, even those involving FP and SIMD registers.

Second, the IR provides an ***unlimited supply of temporaries*** in which guest registers, shadow registers, and intermediate values can be manipulated. This is invaluable for ease-of-use because shadow operations can introduce many extra intermediate values.

Third, the IR's RISC-ness ***exposes all implicit intermediate values***, such as those computed by complex addressing modes, which can make instrumentation easier, particularly on a CISC architecture like x86.

Fourth, ***all code is treated equally.*** Shadow operations benefit fully from Valgrind's post-instrumentation IR optimiser and instruction selector. This makes them easy to write, because one can rely on obvious redundancies being optimised away. This is a consequence of using D&R.

This third feature is also crucial for performance, because it means that client code and analysis code can be interleaved arbitrarily by the back-end. For example, Valgrind's register allocator works with guest and shadow registers equally to minimise spilling. Also, no special tricks are required to prevent analysis code from perturbing condition codes, because they are already computed explicitly rather than as a side-effect of client code.

R2: Provide shadow memory. Valgrind provides no overt support for shadow memory, such as built-in data structures, because

[6] Thus kernel code can run in parallel with user code. This is allowable because the kernel code does not affect shadow memory.

[7] This is another example where avoiding dependencies on other software improved robustness.

[8] Ada programs use them particularly often, and Valgrind was more or less unusable with Ada programs until this was implemented.

shadow memory varies enough from tool to tool [19] that it is difficult to factor out any common supporting operations. However, Valgrind does provide two crucial features to avoid problems with the non-atomicity of loads/stores and shadow loads/stores: its serialisation of threads, and its guaranteed delivery of asynchronous signals only between code blocks. Together they allow shadow value tools to run any multi-threaded program correctly and efficiently on uni-processors, and correctly on multi-processors, without any need for shadow memory locking.

R3: Instrument read/write instructions. Valgrind supports this requirement—all reads and writes of registers and memory are visible in the IR and instrumentable. The IR's load/store nature makes instrumentation of memory accesses particularly easy. Also, the splitting of complex CISC instructions into multiple distinct operations helps some tools, e.g. by exposing intermediate values such as addresses computed with complex addressing modes, and making condition code computations explicit. Again, this is a consequence of using D&R.

As for the added analysis code: the ability to write it as inline IR helps with efficiency and ensures that analysis code is as expressive (e.g. can use FP and SIMD operations) as client code; the ability to write it in separate C functions also allows more complex analysis code to be written easily.

R4–R7. These requirements (instrument read/write system calls, instrument start-up allocations, instrument system call (de)allocations, and instrument stack (de)allocations) are all supported by Valgrind's events system. The left-most column of Table 1 shows which events are used for each requirement.

R8: Instrument heap (de)allocations. Valgrind does not track heap allocations and deallocations with its events system. (It could, this is due to historical reasons.) Instead, tools that need to track these events can use function wrappers or function replacements for the relevant functions (e.g. `malloc`, `free`).

R9: Extra Output. Valgrind allows a shadow value tool to print error messages during execution and at termination using its I/O routines, which send output to a file descriptor (`stderr` by default), file, or socket, as specified by a command line option. Tools can also write additional data to files. Valgrind provides other useful output-related services: error recording, the ability to suppress (ignore) uninteresting/unfixable errors via suppressions listed in files, stack tracing, and debug information reading.

5. Evaluation

We now quantify how easy it is to write Valgrind tools, discuss their robustness and capabilities, and measure their performance.

5.1 Tool-writing Ease

We can use code sizes to roughly measure the amount of effort that went into Valgrind's core and various tools. In Valgrind 3.2.1, the core contains 170,280 lines of C and 3,207 lines of assembly code (including comments and blank lines). In comparison, Memcheck contains 10,509 lines of C, Cachegrind (a cache profiler) is 2,431 lines of C, Massif (a heap profiler) is 1,764, and Nulgrind (the "null" tool that adds no analysis code) is 39. Even though lines of code is not a good measure of coding effort, the benefit of using Valgrind is clear, compared to writing a new tool from scratch. Having said that, heavyweight tools like Memcheck are still not trivial to write, and require a reasonable amount of code.

Valgrind's use of D&R can make simple tools more difficult to write than in C&A frameworks. For example, a tool that traces memory accesses would be about 30 lines of code in Pin, and about 100 in Valgrind. However, in our experience, for the most interesting tools most of the development effort goes not into extracting basic data (such as run-time addresses and values), but into analysing and presenting that data in useful ways to the user—it makes little difference whether it takes 30 lines or 100 lines of code to extract a memory access trace if a tool contains 2,000 lines devoted to analysing it.

In contrast, for heavyweight tools D&R makes instrumentation easier for tools like Memcheck because of the reasons explained in Sections 3.5 and 4.

5.2 Tool Robustness

By "robustness", we mean how many different programs a tool can correctly run. For tools built with DBI frameworks, this covers both the framework and the tool—it is possible to build a non-robust tool on top of a robust framework.

Robustness is not easy to quantify. We provide anecdotal evidence for the robustness of Valgrind and Memcheck: their large number of users; and the range of programs with which they have been successfully used; the range of platforms they support; and some design decisions we have made to improve robustness.

Valgrind has become a standard C and C++ development tool on Linux. Memcheck is the most popular Valgrind tool, accounting for about 80% of all Valgrind tool use [27]. The Valgrind website [28] averages more than 1,000 unique visitors per day. Valgrind tools are used by the developers of many large projects, such as Firefox, OpenOffice, KDE, GNOME, Qt, libstdc++, MySQL, Perl, Python, PHP, Samba, RenderMan, and Unreal Tournament.[9] They have successfully been used on a wide range of different software types, implemented using many different languages and compilers, on programs containing up to 25 million lines of code. They also successfully handle multi-threaded programs.

Valgrind and Memcheck run on multiple platforms, 32-bit and 64-bit: x86/Linux, AMD64/Linux, and PPC{32,64}/{Linux,AIX}. There are also experimental ports to x86/MacOS X, x86/FreeBSD, and x86/Solaris. We believe Valgrind is suitable for porting to any typical RISC or CISC architecture, such as ARM or SPARC. VLIW architectures such as IA64 would be possible but Valgrind's use of D&R would make reasonable performance harder to attain, as VLIW code generation is more difficult. We also believe it can be ported to any Unix-style OS; a port to Windows may be possible but would be much more challenging. Porting to a new architecture requires writing new code for the JIT compiler, such as an instruction encoder and decoder, and code to describe the new machine state (i.e. register layout). Porting to a new OS requires some new code for handling details such as signals and address space management. Porting to a new architecture and/or OS requires some new system call wrappers to be written. Memcheck (and other shadow value tools) usually do not need to be changed if Valgrind is ported to new platforms.

The robustness of Valgrind and Memcheck has slowly improved over time. Earlier sections of this paper showed that several Valgrind sub-systems have been re-implemented once or twice in an effort to make them more robust. Also, we have gradually removed all dependencies on external libraries, even the C library. Indeed, since mid-2005 Valgrind has been able to run itself, which is no mean feat considering how many strange things it does.

[9] The SPEC benchmarks are sometimes used as a measure of robustness. They are actually not particularly difficult to run—they stress a DBA tool's code generation well, but they are all single-threaded, compute-bound, not particularly large, do not use many system calls, and do not do tricky things with memory layout or signals. The "large projects" listed above stress a DBA tool much more than the SPEC benchmarks.

5.3 Tool Instrumentation Capabilities

In this section, we compare Valgrind's support for all nine shadow value requirements against Pin [11], because Pin is the best known of the currently available DBI frameworks, and the one that has the most support for shadow values (after Valgrind). The following comparison is based on discussions with two Pin developers [10].

Pin supports R5 (instrument start-up allocations), R8 (instrument heap (de)allocations) and R9 (extra output) directly. It does not support R6 (instrument system call (de)allocations) and R7 (instrument stack (de)allocations) directly, but provides features that allow a Pin tool to manually support them fairly easily.

For R1 (provide shadow registers) Pin provides "virtual registers" which are register-allocated along with guest registers and saved in memory when a thread is not running. Shadow registers could be stored in them. However, virtual registers are not fully first-class citizens. For example, there are no 128-bit virtual registers, so 128-bit SIMD registers cannot be fully shadowed, which would prevent some tools (e.g. Memcheck) from working fully.

Pin provides no built-in support for R2 (provide shadow memory), so tools must cope with the non-atomicity of loads/stores and shadow loads/stores themselves.[10] For example, the Pin tool called *pinSEL* [14], which uses shadow memory but not full shadow values, sets and checks an extra *interference bit* on every shadow load. This lets it handle any thread switches or asynchronous signals that occur between a load/store and a shadow load/store (both of which can occur even on uni-processors under Pin). Multi-threaded programs running on multi-processors are even trickier, and pinSEL does not handle them. In comparison, Valgrind's thread serialisation and asynchronous signal treatment frees shadow value tools from having to deal with this issue.

For R3 (instrument read/write instructions) Pin allows all register and memory accesses to be seen. However, analysis code in Pin is written as C functions, which can be inlined if they contain no control flow. This means that SIMD instructions are again a problem; if a tool needs to use SIMD instructions in its analysis code (as Memcheck does), these would have to be written in Pin using (platform-specific) inline assembly code. This is caused by Pin using C&A and its method for writing analysis code (C code) having less expressivity than client code (machine code).

R4 (instrument read/write system calls) is another stumbling block; it can be done manually within a tool via Pin's system call instrumentation, but would require a large effort—each shadow value tool would essentially need to reimplement Valgrind's system call wrappers.

5.4 Tool Performance

We performed experiments on 25 of the 26 SPEC CPU2000 benchmarks (we could not run `galgel` as gfortran failed to compile it). We ran them with the "reference" inputs in 32-bit mode on a 2.4 GHz Intel Core 2 Duo with 1GB RAM and a 4MB L2 cache running SUSE Linux 10.2, kernel 2.6.18.2. We compared several tools built with Valgrind 3.2.1: (a) Nulgrind, the "no instrumentation" tool; (b) ICntI, an instruction counter which uses inline code to increment a counter for every instruction executed; (c) ICntC, like ICntI but uses a C function call to increment the counter; and (d) Memcheck (with leak-checking off, because it runs at program termination and so would cloud the comparison). Table 2 shows the slow-down factors of these tools.

Lightweight tools. The mean slow-down of 4.3x for the no-instrumentation case (Nulgrind) is high compared to other frameworks. This is consistent with other researchers' findings—a previous comparison [11] showed that Valgrind is 4.0x slower than Pin and 4.4x slower than DynamoRIO on the SPEC CPU2000 integer benchmarks in the no-instrumentation case, and 3.3x and 2.0x slower for a lightweight basic block counting tool.[11]

Program	Nat. (s)	Nulg.	ICntI	ICntC	Memc.
bzip2	192.7	3.5	7.2	10.5	16.1
crafty	92.4	6.9	12.3	22.5	36.0
eon	408.5	7.5	11.8	21.0	51.4
gap	131.3	4.0	9.1	13.5	25.5
gcc	90.0	5.3	9.0	14.1	39.0
gzip	212.1	3.2	5.9	9.0	14.7
mcf	87.0	2.0	3.5	5.4	7.0
parser	218.9	3.6	7.0	10.4	17.8
perlbmk	179.6	4.8	9.6	14.6	27.1
twolf	262.5	3.1	6.5	10.7	16.0
vortex	86.7	6.5	11.4	17.8	38.7
vpr	149.4	4.1	7.7	11.3	16.4
ammp	345.2	3.4	6.5	9.1	32.7
applu	583.0	5.2	14.1	28.1	19.7
apsi	469.0	3.4	8.2	12.5	16.4
art	100.4	4.7	9.4	13.7	24.0
equake	118.2	3.8	8.4	12.4	17.1
facerec	280.9	4.7	8.2	12.2	17.4
fma3d	284.7	4.1	9.4	16.2	26.0
lucas	183.5	3.7	7.1	10.8	24.8
mesa	148.9	5.9	10.3	15.9	57.9
mgrid	809.1	3.5	9.8	14.4	16.9
sixtrack	355.7	5.6	13.4	18.3	20.2
swim	388.2	3.2	11.9	15.3	10.7
wupwise	192.1	7.4	11.8	17.3	26.7
geo. mean		4.3	8.8	13.5	22.1

Table 2. Performance of four Valgrind tools on SPEC CPU2000. Column 1 gives the program name; integer programs are listed before floating-point programs. Column 2 gives the native execution time in seconds. Columns 3–6 give the slow-down factors for each tool. The final row shows each column's geometric mean.

Re-implementing chaining in Valgrind would improve these cases somewhat. However, these lightweight tools are exactly the kinds of tools that Valgrind is *not* targeted at, and Valgrind will never be as fast as Pin or DynamoRIO for these cases. For example, consider Valgrind's use of a D&R representation. For a simple tool like a basic block counter, D&R makes no sense. Rather, the use of D&R is targeted towards heavyweight tools. For this reason, we do not repeat such comparisons with lightweight tools.

The difference between ICntI and ICntC shows the advantage of inline code over C calls. ICntI could be further improved by batching counter increments together.

Heavyweight tools built with Valgrind. Memcheck's mean slow-down factor is 22.2x. Other shadow value tools built with Valgrind have similar or worse slow-downs. TaintCheck ran 37x slower on an invocation of bzip2 [20], but had better performance on an I/O-bound invocation of the Apache web server. Annelid ran a subset of the SPEC CPU2000 benchmarks ("train" inputs) 35.2x slower than native [16]. McCamant and Ernst's secret tracker has slow-downs "similar to Memcheck... 10–100x for CPU-bound programs" [13]. Redux did much more expensive analysis and was not practical for anything more than toy programs [17]. Slow-down figures are not available for DynCompB [7].

[10] It does have thread-locking primitives, but they would be too coarse-grained to be practical for use with shadow memory.

[11] But the measured Valgrind tool used a C function to increment the counter; the use of inline code would have narrowed the gap.

None of these tools are as optimised as Memcheck, particularly their handling of shadow memory; more aggressive implementations would have slow-downs closer to Memcheck's.

Other heavyweight tools. Hobbes' slow-down factors for SPEC CPU2000 integer programs were in the range 30–187x. However, Hobbes used a built-from-scratch binary interpreter rather than a JIT compiler, so this is a poor comparison point.

TaintTrace [6] is built with DynamoRIO, implements shadow registers within the tool itself, and has an mean slow-down factor of 5.5x for a subset of the SPEC CPU2000 benchmarks. LIFT [23] is built with StarDBT, a dynamic binary translation/instrumentation framework developed by Intel. It has a mean slow-down factor of 3.5x for a similar subset of the SPEC CPU2000 integer benchmarks. These two tools are much faster than Memcheck and TaintCheck. This is partly because they are doing a simpler analysis—they track one taintedness bit per byte, whereas Memcheck tracks one definedness bit per bit and does various other kinds of checking, and TaintCheck records four bytes per byte in order to record origins of tainted values.

More importantly, they are faster because they are less robust and have more limited instrumentation capabilities, in several ways.

- TaintTrace reserves the entire upper half of the address space for shadow memory, which makes shadow memory accesses trivial and inlinable, but: (a) it wastes 7/8 of that space (7/16 of the total address space) because each shadow byte holds only a single taintedness bit, and (b) reserving large areas of address space works most of the time on Linux, but is untenable on many other OSes—e.g. Mac OS X, AIX, and many embedded OSes put a lot of code and data in the top half of the address space [19]. In comparison, Memcheck instead uses a shadow memory layout that is slower—largely because it requires calls to C functions for shadow loads and stores—but more flexible and thus more robust, and shadow memory operations account for close to half of Memcheck's overhead [19].
- LIFT translates 32-bit x86 code to run on x86-64 machines. x86-64 machines have eight extra integer registers which are not used by x86 programs which make shadow registers very easy to implement. The translation also avoids the problems of fitting shadow memory into the 32-bit address space, as LIFT has a 64-bit address space to work in. In one way, this is the ideal approach—having twice the registers and (more than) twice as much memory is perfect for shadow values. However, it is only narrowly applicable.

  If LIFT was implemented without binary translation the extra register pressure would not be great—its shadow values are compact (one bit per byte) and so eight shadow registers can be squeezed into a single host register—and so the slow-down might be moderate, particularly on a platform with lots of registers such as PowerPC. But for Memcheck, TaintCheck, or any other tool that has larger shadow register values, the slow-down would be greater.
- Neither TaintTrace nor LIFT handle programs that use FP or SIMD code [5, 22]. We have found that handling these cases by adding them later is more difficult than it might seem. The hybrid IR used by Valgrind (mentioned in Sections 3.5 and 3.6) had FP/SIMD handling added (via C&A) only once the integer D&R part was working. This meant that the Valgrind and Memcheck's performance on FP/SIMD code was much worse than on integer code because the x86 FP/SIMD state had to be frequently saved and restored (even though we optimised away redundant ones whenever possible). Also, the instrumentation capabilities were worse for FP/SIMD code, and Memcheck handled such code less accurately [25]. The rotating x87 FP register stack is particularly difficult to handle well with C&A code representation.
- Neither TaintTrace nor LIFT handle multi-threaded programs.

TaintTrace and LIFT show that shadow value tools can be implemented in frameworks other than Valgrind, and have better performance than Memcheck, if they use techniques that are applicable to a narrower range of programs. We believe that the robustness and instrumentation capabilities of TaintTrace and LIFT could be improved somewhat, and that such changes would reduce their performance. But in general, we believe that making these tools as robust and accurate as Memcheck would be very difficult given that they are built with DBI frameworks that do not support all nine shadow value requirements.

Nonetheless, research prototypes with a narrower focus can identify new techniques that are applicable in real-world tools. For example, LIFT uses clever techniques to avoid performing some shadow operations; these might be adaptable for use in Memcheck.

Although there is some scope for improving Memcheck's performance (by adding chaining to Valgrind's core and using LIFT's techniques for skipping shadow operations), given its other characteristics, we believe that its performance is reasonable considering how much analysis it does [25, 19]. Memcheck's popularity shows that programmers are willing to use a tool with a large slow-down if its benefits are high enough, and it is easily the fastest shadow value tool we know of that is also robust and general. We also believe and that Valgrind's design features—such as its unique D&R IR with first-class shadow registers—are crucial in achieving this reasonable performance despite the challenging requirements of shadow values.

5.5 Summary

Every DBI framework has a number of important characteristics: ease of tool-writing, robustness, instrumentation capabilities, and performance. Robustness and performance are also important for DBA tools built with DBI frameworks, and tool designs crucially affect these characteristics. Performance has traditionally received the most attention, but the other characteristics are equally important. Trade-offs must be made in any framework or tool, and all relevant characteristics should be considered in any comparisons between frameworks and/or tools.

For lightweight DBA, Valgrind is less suitable than more performance-oriented frameworks such as Pin and DynamoRIO. For heavyweight DBA, Valgrind has a uniquely suitable combination of characteristics: it makes tools relatively easy to write, allows them to be robust, provides powerful instrumentation capabilities, and allows reasonable performance. These characteristics are exemplified by Memcheck, which is highly accurate, widely used, and reasonably fast.

6. Related Work

There are many DBI frameworks; Nethercote [15] compares eleven in detail (that publication also discusses shadow values, but in less detail than this paper). They vary in numerous ways: platforms supported, instrumentation mechanisms, kinds of analysis code supported, robustness, speed, and availability. Judging by recent literature, those that are both widely-used and actively maintained are Pin [11], DynamoRIO [3], DIOTA [12], and Valgrind.

We compared Valgrind to Pin in Section 5. As for other DBI frameworks, they all provide less shadow value support than Pin; in particular, they provide no support for R1 (provide shadow registers), such as virtual registers or register re-allocation. We believe R1 is the hardest requirement for a tool to fulfil without help from its framework; without such support, tools have to find ways to "steal" extra registers for themselves. This is possible to some

extent, but very difficult to do on the scale required for shadow values in a manner that is robust and gives reasonable performance.

The nine shadow value tools we know of were discussed in Section 1.2 and 5.4. Six of them were built with Valgrind.

Shadow value tools are not only applicable at the binary level. For example, Perl's "taint mode" [29] and Patil and Fischer's bounds checker for C [21] implement analyses similar to those of TaintCheck and Annelid (see Section 1) at the level of source code. The underlying tool ideas are very similar, but the implementation details are completely different.

7. Future Work and Conclusion

Valgrind is a widely-used DBI framework. It is designed to support DBA heavyweight tools, such as shadow value tools, and therefore can be used to build most conceivable DBA tools. This paper has identified the requirements of shadow value tools and how Valgrind supports them, and shown that Valgrind inhabits a unique part of the DBI framework design space. We have focused more on Valgrind's instrumentation capabilities than its performance, because (a) they are an equally important but less-studied topic, and (b) they distinguish Valgrind from other related frameworks.

We think there are two main areas of future research for Valgrind. First, we want to find a way to avoid forcing serial thread execution in a way that does not compromise the correctness of shadow value tools. This will become increasingly important as multi-core machines proliferate. Second, Memcheck has already shown that heavyweight DBA tools can help programmers greatly improve their programs. We think there is plenty of scope for new heavyweight DBA tools, particularly shadow value tools, and we hope Valgrind will be used to build some of these tools.

Acknowledgments

Thanks to: Greg Lueck for his Pin expertise; Mike Bond, Kim Hazelwood and the anonymous reviewers for reviewing this paper; and everyone who has contributed to Valgrind over the years, particularly Jeremy Fitzhardinge, Tom Hughes and Donna Robinson.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of PLDI 2000*, pages 1–12, Vancouver, Canada, June 2000.

[2] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, Cambridge, Mass., USA, September 2004.

[3] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of CGO'03*, pages 265–276, San Francisco, California, USA, March 2003.

[4] M. Burrows, S. N. Freund, and J. L. Wiener. Run-time type checking for binary programs. In *Proceedings of CC 2003*, pages 90–105, Warsaw, Poland, April 2003.

[5] W. Cheng. Personal communication, November 2006.

[6] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *Proceedings of ISCC 2006*, pages 749–754, Cagliari, Sardinia, Italy, June 2006.

[7] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. In *Proceedings of ISSTA 2006*, pages 255–265, Portland, Maine, USA, July 2006.

[8] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136, San Francisco, California, USA, January 1992.

[9] K. Hazelwood. *Code Cache Management in Dynamic Optimization Systems*. PhD thesis, Harvard University, Cambridge, Mass., USA, May 2004.

[10] G. Lueck and R. Cohn. Personal communication, September–November 2006.

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of PLDI 2005*, pages 191–200, Chicago, Illinois, USA, June 2005.

[12] J. Maebe, M. Ronsse, and K. De Bosschere. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Proceedings of WBT-2002*, Charlottesville, Virginia, USA, September 2002.

[13] S. McCamant and M. D. Ernst. Quantitative information-flow tracking for C and related languages. Technical Report MIT-CSAIL-TR-2006-076, MIT, Cambridge, Mass., USA, 2006.

[14] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operation system effects to guide application-level architecture simulation. In *Proceedings of SIGMetrics/Performance 2006*, pages 216–227, St. Malo, France, June 2006.

[15] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, United Kingdom, November 2004.

[16] N. Nethercote and J. Fitzhardinge. Bounds-checking entire programs without recompiling. In *Informal Proceedings of SPACE 2004*, Venice, Italy, January 2004.

[17] N. Nethercote and A. Mycroft. Redux: A dynamic dataflow tracer. *ENTCS*, 89(2), 2003.

[18] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *ENTCS*, 89(2), 2003.

[19] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Proceedings of VEE 2007*, San Diego, California, USA, June 2007.

[20] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of NDSS '05*, San Diego, California, USA, February 2005.

[21] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software—Practice and Experience*, 27(1):87–110, January 1997.

[22] F. Qin. Personal communication, March 2007.

[23] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. Lift: A low-oeverhead practical information flow tracking system for detecting security attacks. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (Micro'06)*, Orlando, Florida, USA, December 2006.

[24] K. Scott, J. W. Davidson, and K. Skadron. Low-overhead software dynamic translation. Technical Report CS-2001-18, University of Virginia, Charlottesville, Virginia, USA, 2001.

[25] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, Anaheim, California, USA, April 2005.

[26] O. Traub, G. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *Proceedings of PLDI '98*, pages 142–151, Montreal, Canada, June 1998.

[27] The Valgrind Developers. 2nd official Valgrind survey, September 2005: full report. `http://www.valgrind.org/gallery/survey_05/report.txt`.

[28] The Valgrind Developers. Valgrind. `http://www.valgrind.org/`.

[29] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly, 3rd edition, 2000.

Dynamo: A Transparent Dynamic Optimization System

Vasanth Bala
vas@hpl.hp.com

Evelyn Duesterwald
duester@hpl.hp.com

Sanjeev Banerjia*
sbanerjia@incert.com

Hewlett-Packard Labs 1 Main Street,
Cambridge, MA 02142

www.hpl.hp.com/cambridge/projects/Dynamo

Abstract

We describe the design and implementation of Dynamo, a software dynamic optimization system that is capable of transparently improving the performance of a native instruction stream as it executes on the processor. The input native instruction stream to Dynamo can be dynamically generated (by a JIT for example), or it can come from the execution of a statically compiled native binary. This paper evaluates the Dynamo system in the latter, more challenging situation, in order to emphasize the limits, rather than the potential, of the system. Our experiments demonstrate that even statically optimized native binaries can be accelerated Dynamo, and often by a significant degree. For example, the average performance of –O optimized SpecInt95 benchmark binaries created by the HP product C compiler is improved to a level comparable to their –O4 optimized version running without Dynamo. Dynamo achieves this by focusing its efforts on optimization opportunities that tend to manifest only at runtime, and hence opportunities that might be difficult for a static compiler to exploit. Dynamo's operation is transparent in the sense that it does not depend on any user annotations or binary instrumentation, and does not require multiple runs, or any special compiler, operating system or hardware support. The Dynamo prototype presented here is a realistic implementation running on an HP PA-8000 workstation under the HPUX 10.20 operating system.

1. Introduction

Recent trends in software and hardware technologies appear to be moving in directions that are making traditional performance delivery mechanisms less effective. The use of object-oriented languages and techniques in modern software development has resulted in a greater degree of delayed binding, limiting the size of the scope available for static compiler analysis. Shrink-wrapped software is being shipped as a collection of DLLs rather than a single monolithic executable, making whole-program optimization at static compile-time virtually impossible. Even in cases where powerful static compiler optimizations can be applied, computer system vendors have to rely on the ISV (independent software vendor) to enable them. This puts computer system vendors in the uncomfortable position of not being able to control the very keys that unlock the performance potential of their own machines. More recently, the use of dynamic code generation environments (like Java JITs and dynamic binary translators) makes the applicability of heavyweight static compiler optimization techniques impractical. Meanwhile, on the hardware side, technology is moving toward offloading more complexity from the hardware logic to the software compiler, as evidenced by the CISC to RISC to VLIW progression.

The problem with this trend is that the static compiler is taking on an increasingly greater performance burden while the obstacles to traditional static compiler analysis are continuing to increase. This will inevitably lead to either very complex compiler software that provides only modest performance gains on general-purpose applications, or highly customized compilers that are tailored for very narrow classes of applications.

The Dynamo project was started in 1996 to investigate a technology that can complement the static compiler's traditional strength as a static performance improvement tool with a novel dynamic performance improvement capability [3]. In contrast to the static compiler, Dynamo offers a client-side performance delivery mechanism that allows computer system vendors to provide some degree of machine-specific performance without the ISV's involvement.

Dynamo is a *dynamic optimization* system (i.e., the input is an executing *native* instruction stream), implemented entirely in software. Its operation is transparent: no preparatory compiler phase or programmer assistance is required, and even legacy native binaries can be dynamically optimized by Dynamo. Because Dynamo operates at runtime, it has to focus its optimization effort very carefully. Its optimizations have to not only improve the executing native program, but also recoup the overhead of Dynamo's own operation.

The input native instruction stream to Dynamo can come from a statically prepared binary created by a traditional optimizing compiler, or it can be dynamically generated by an application such as a JIT. Clearly, the runtime performance opportunities available for Dynamo can vary significantly depending on the source of this input native instruction stream. The experiments reported in this paper only discuss the operation of Dynamo in the more challenging situation of accelerating the execution of a statically optimized native binary. The performance data presented here thus serve as an indicator of the limits of the Dynamo system, rather than its potential. The data demonstrates that even in this extreme test case, Dynamo manages to speedup many applications, and comes close to breaking even in the worst case.

Section 1 gives an overview of how Dynamo works. The following sections highlight several key innovations of the Dynamo system. Section 2 describes Dynamo's startup mechanism, Section 4 gives an overview of the hot code selection, optimization and code generation process, Section 5 describes how different optimized code snippets are linked together, Section 6 describes how the storage containing the dynamically optimized

*The author is presently with InCert Corporation, Cambridge, MA.

PLDI 2000, Vancouver, British Columbia, Canada.

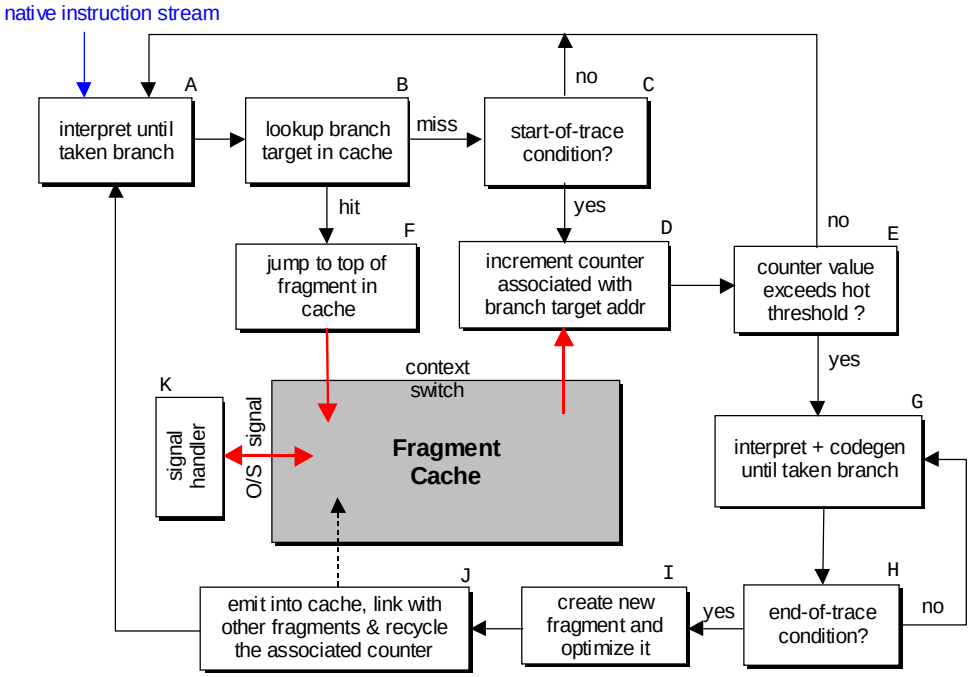


Figure 1. How Dynamo works

code is managed, and Section 7 describes signal handling. Finally, Section 8 summarizes the experimental data to evaluate Dynamo's performance. Dynamo is a complex system that took several years to engineer. This paper only provides an overview of the whole system. Further details are available in [2] and on the Dynamo project website (www.hpl.hp.com/cambridge/projects/Dynamo).

2. Overview

From a user's perspective, Dynamo looks like a PA-8000 software interpreter that itself runs on a PA-8000 processor (the hardware interpreter). Interpretation allows Dynamo to observe execution behavior without having to instrument the application binary. Since software interpretation is much slower than direct execution on the processor, Dynamo only interprets the instruction stream until a "hot" instruction sequence (or *trace*) is identified. At that point, Dynamo generates an optimized version of the trace (called a *fragment*) into a software code cache (called the *fragment cache*). Subsequent encounters of the hot trace's entry address during interpretation will cause control to jump to the top of the corresponding cached fragment. This effectively suspends the interpreter and allows the cached code to execute directly on the processor without incurring any further interpretive overhead. When control eventually exits the fragment cache, Dynamo resumes interpreting the instruction stream, and the process repeats itself.

Figure 1 illustrates this flow of control in more detail. Dynamo starts out by interpreting the input native instruction stream until a taken branch is encountered (A). If the branch target address corresponds to the entry point of a fragment already in the fragment cache (B), control jumps to the top of that fragment, effectively suspending Dynamo, and causing execution of the cached fragments to occur directly on the underlying processor (F). Otherwise, if the branch target satisfies a "start-of-trace" condition (C), a counter associated with the target address is incremented (D). Our current prototype defines start-of-trace as targets of backward-taken branches (likely loop headers) and fragment cache exit branches (exits from previously identified hot traces). If the counter value exceeds a preset hot threshold (E), the interpreter toggles state and goes into "code generation mode" (G). When interpreting in this mode, the native instruction sequence being interpreted is recorded in a hot trace buffer, until an "end-of-trace" condition is reached (H). At that point the hot trace buffer is processed by a fast, lightweight optimizer (I) to create an optimized single-entry, multi-exit, contiguous sequence of instructions called the *fragment*[1]. Our current prototype defines end-of-trace as backward taken branches or taken branches whose targets correspond to fragment entry points in the fragment cache (i.e., fragment cache hits). A trace may also be truncated if its length exceeds a certain number of instructions. The fragment generated by the optimizer is emitted into the fragment cache by a linker (J), which also connects fragment exit branches to other fragments in the fragment cache if possible. Connecting fragments together in this manner minimizes expensive fragment cache exits to the Dynamo interpretive loop. The new fragment is tagged with the application binary address of the start-of-trace instruction.

As execution proceeds, the application's working set gradually materializes in the fragment cache, and the Dynamo *overhead* (time spent in the Dynamo interpretive loop / time spent executing in the fragment cache) begins to drop. Assuming that the majority of an application's execution time is typically spent in a small portion of its code, the performance benefits from repeated reuse of the optimized fragments can be sufficient to offset the overhead of Dynamo's operation. On the SpecInt95 benchmarks,

[1] A fragment is similar to a superblock, except for the fact that it is a dynamic instruction sequence, and can cross static program boundaries like procedure calls and returns.

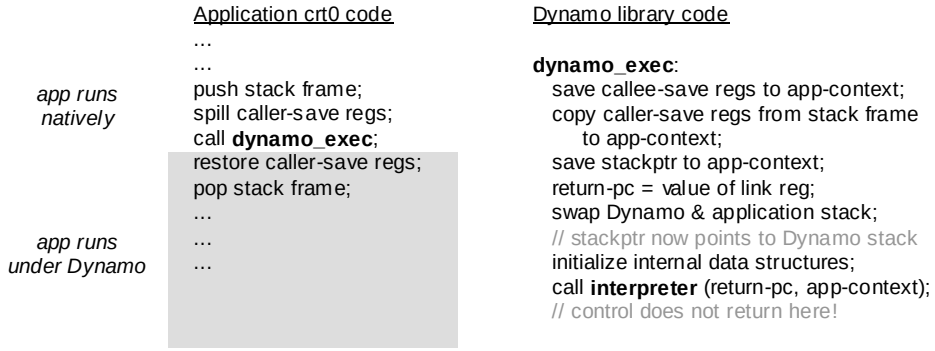


Figure 2. How Dynamo gains control over the application

the average Dynamo overhead is less than 1.5% of execution time. Dynamo's interpreter-based hot trace selection process (A-H) dominates this overhead, with the optimizer and linker components (I, J) contributing a relatively insignificant amount.

3. Startup and Initialization

Dynamo is provided as a user-mode dynamically linked library (shared library). The entry point into this library is the routine dynamo_exec. When dynamo_exec is invoked by an application, the remainder of the application code after return from the dynamo_exec call will execute under Dynamo control.

As outlined in Figure 2, dynamo_exec first saves a snapshot of the application's context (i.e., the machine registers and stack environment) to an internal app-context data structure. It then swaps the stack environment so that Dynamo's own code uses a custom runtime stack allocated separately for its use. Dynamo's operation thus does not interfere with the runtime stack of the application running on it. The interpreter (box A in Figure 1) is eventually invoked with the return-pc corresponding to the application's dynamo_exec call. The interpreter starts interpreting the application code from this return-pc, using the context saved in app-context. The interpreter never returns to dynamo_exec (unless a special bailout condition occurs, which is discussed later), and Dynamo has gained control over the application. From this point onwards, an application instruction is either interpreted, or a copy of it is executed in the fragment cache. The original instruction is never executed in place the way it would have been if the application were running directly on the processor.

We provide a custom version of the execution startup code crt0.o, that checks to see if the Dynamo library is installed on the system, and if it is, invokes dynamo_start prior to the jump to _start (the application's main entry point). Application binaries that are linked with this version of crt0.o will transparently invoke Dynamo if Dynamo is installed on the system, otherwise they will execute normally. The application binary itself remains unchanged whether or not it is run under Dynamo. This strategy allows Dynamo to preserve the original mapping of the application's text segment, a key requirement for transparent operation.

As part of the initialization done in dynamo_exec prior to actually invoking the interpreter, Dynamo *mmaps* a separate area of memory that it manages itself. All dynamically allocated objects in Dynamo code are created in this area of memory. Access to this area is protected to prevent the application from inadvertently or maliciously corrupting Dynamo's state.

4. Fragment Formation

Due to the significant overheads of operating at runtime, Dynamo has to maximize the impact of any optimization that it performs. Furthermore, since the objective is to complement, not compete, with the compiler that generated the instruction stream, Dynamo primarily looks for performance opportunities that tend to manifest themselves in the runtime context of the application. These are generally redundancies that cross static program boundaries like procedure calls, returns, virtual function calls, indirect branches and dynamically linked function calls. Another performance opportunity is instruction cache utilization, since a dynamically contiguous sequence of frequently executing instructions may often be statically non-contiguous in the application binary.

Dynamo's unit of runtime optimization is a *trace*, defined as a dynamic sequence of consecutively executed instructions. A trace starts at an address that satisfies the start-of-trace condition and ends at an address that satisfies the end-of-trace condition. Traces may extend across statically or dynamically linked procedure calls/returns, indirect branches and virtual function calls. Dynamo first selects a "hot" trace, then optimizes it, and finally emits relocatable code for it into the fragment cache. The emitted relocatable code is contiguous in the fragment cache memory, and branches that exit this code jump to corresponding exit stubs at the bottom of the code. This code is referred to as a *fragment*. The trace is a unit of the application's dynamic instruction stream (i.e., a sequence of application instructions whose addresses are application binary addresses) whereas the fragment is a Dynamo internal unit, addressed by fragment cache addresses. The following subsections outline the trace selection, trace optimization and fragment code generation mechanisms of Dynamo.

4.1 Trace selection

Since Dynamo operates at runtime, it cannot afford to use elaborate profiling mechanisms to identify hot traces (such as [14][4]). Moreover, most profiling techniques in use today have been designed for offline use, where the gathered profile data is collated and analyzed post-mortem. The objective here is not accuracy, but predictability. If a particular trace is very hot over a short period of time, but its overall contribution to the execution time is small, it may still be an important trace to identify. Another concern for Dynamo is the amount of counter updates and counter storage required for identifying hot traces, since this adds to the overhead and memory footprint of the system.

As discussed in Section 2, Dynamo uses software interpretation of the instruction stream to observe runtime execution behavior. Interpretation is expensive but it prevents the

need to instrument the application binary or otherwise perturb it in any way. Interpretation is preferable to statistical PC sampling because it does not interfere with applications that use timer interrupts. Also, as we will elaborate shortly, interpretation allows Dynamo to select hot regions directly without having to collate and analyze point statistics like the kind produced by PC sampling techniques. Another important advantage of interpretation is that it is a deterministic trace selection scheme, which makes the task of engineering the Dynamo system much easier.

It is worth noting that the "interpreter" here is a native instruction interpreter and that the underlying CPU is itself a very fast native instruction interpreter implemented in hardware. This fact can be exploited on machines that provide fast breakpoint traps (e.g., through user-mode accessible breakpoint window registers) to implement the Dynamo interpreter very efficiently [2]. On the PA-8000 however, breakpoint traps are very expensive, and it was more efficient to implement the interpreter by using emulation. The higher the interpretive overhead, the earlier Dynamo has to predict the hot trace in order to keep the overheads low. In general, the more speculative the trace prediction scheme, the larger we need to size the fragment cache, to compensate for the larger number of traces picked as a result. Thus, the interpretive overhead has a ripple effect throughout the rest of the Dynamo system.

Dynamo uses a speculative scheme we refer to as MRET (for *most recently executed tail*) to pick hot traces without doing any path or branch profiling. The MRET strategy works as follows. Dynamo associates a counter with certain selected start-of-trace points such as the target addresses of backward taken branches. The target of a backward taken branch is very likely to be a loop header, and thus the head of several hot traces in the loop body. If the counter associated with a certain start-of-trace address exceeds a preset threshold value, Dynamo switches its interpreter to a mode where the sequence of interpreted instructions is recorded as they are being interpreted. Eventually, when an end-of-trace condition is reached, the recorded sequence of instructions (the most recently executed tail starting from the hot start-of-trace) is selected as a hot trace.

The insight behind MRET is that when an instruction becomes hot, it is statistically likely that the very next sequence of executed instructions that follow it is also hot. Thus, instead of profiling the branches in the rest of the sequence, we simply record the tail of instructions following the hot start-of-trace and optimistically pick this sequence as a hot trace. Besides its simplicity and ease of engineering, MRET has the advantage of requiring much smaller counter storage than traditional branch or path profiling techniques. Counters are only maintained for potential loop headers. Furthermore, once a hot trace has been selected and emitted into the fragment cache, the counter associated with its start-of-trace address can be recycled. This is possible because all future occurrences of this address will cause the cached version of the code to be executed and no further profiling is required.

Subsequent hot traces that also start at the same start-of-trace address will be selected when control exits the first selected trace for that start-of-trace address. Exits from previously selected hot traces are treated as start-of-trace points by Dynamo (see Figure 1). This allows subsequent hot tails that follow the earlier hot start-of-trace to be selected by the MRET scheme in the usual manner.

No profiling is done on the code generated into Dynamo's fragment cache. This allows the cached code to run directly on the processor at full native speed without any Dynamo introduced overheads. The flip side of this is that if the biases of some branches change after a hot trace was selected, Dynamo would be unable to detect it. In order to allow Dynamo to adapt to changing branch biases, the fragment cache is designed to tolerate periodic flushes. Periodically flushing some of the traces in the fragment cache helps remove unused traces, and also forces re-selection of active traces. This is discussed in more detail in Section 6.

4.2 Trace optimization

The selected hot trace is prepared for optimization by converting it into a low-level intermediate representation (IR) that is very close to the underlying machine instruction set.

The first task of trace optimization is to transform the branches on the trace so that their fall-through direction remains on the trace. Loops are only allowed if the loop-back branch targets the start-of-trace. Otherwise the loop-back branch is treated as a trace exit. Unconditional direct branches are redundant on the trace

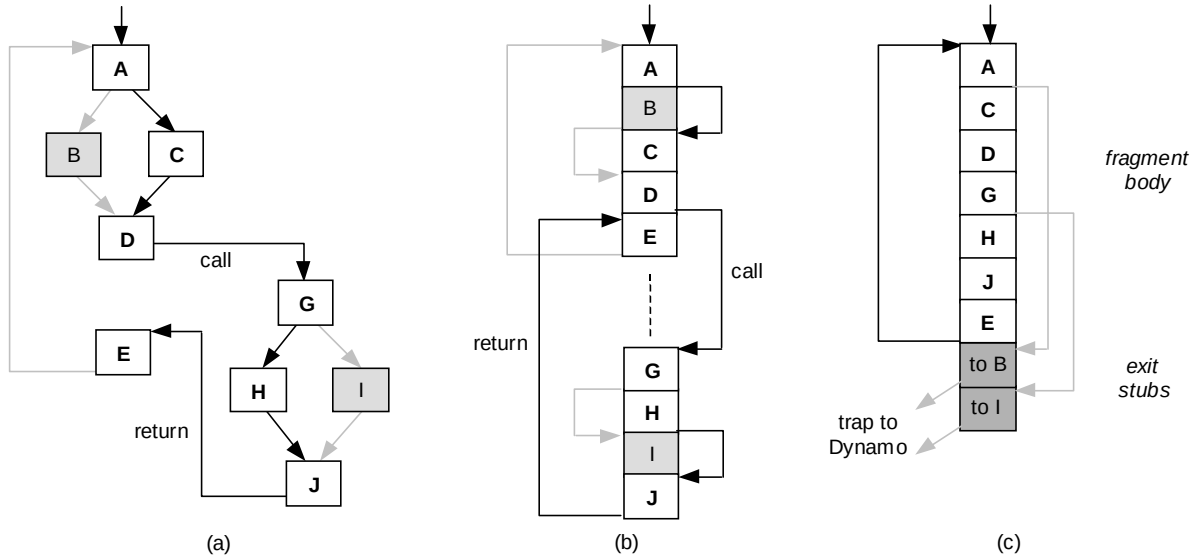


Figure 3. Control flow snippet in the application binary, (b) Layout of this snippet in the application program's memory, and (c) Layout of a trace through this snippet in Dynamo's fragment cache.

and can be removed. In the case of branches with side-effects, such as branch-and-link branches, the side-effect is preserved even if the branch itself is removed. After trace optimization, no branch-and-link type branches remain on the trace.

Even indirect branches may be redundant. For example, a return branch if preceded by the corresponding call on the trace is redundant and will be removed. Other indirect branches are optimistically transformed into direct conditional branches. The transformed conditional branch compares the dynamic branch target with the target contained in the trace at the time the trace was selected (referred to as the *predicted* indirect branch target). If the comparison succeeds, control goes to the predicted (on-trace) target. If the comparison fails, control is directed to a special Dynamo routine that looks up a Dynamo-maintained switch table. The switch table is a hash table indexed by indirect branch target addresses (application binary addresses). The table entries contain the fragment cache address corresponding to the target. If an entry is found for the dynamic indirect branch target, control is directed to the corresponding fragment cache address. Otherwise, control exits the fragment cache to the Dynamo interpreter. If the interpreter then selects a new hot trace starting at that dynamic indirect branch target, Dynamo will add a new entry to the switch table corresponding to the mapping from the start-of-trace application address to its fragment cache address. Assuming execution follows the selected hot trace most of the time, this transformation replaces a potentially expensive indirect branch with a less expensive direct conditional branch. The following outlines the transformed code for an indirect branch instruction:

```
// assuming the indirect branch's dynamic target is in Rx
spill Rscratch to app-context; // free a fixed register
set Rscratch = address of predicted on-trace target;
if (Rx = = Rscratch) goto predicted target;
copy Rx to Rscratch;
goto switch_table_lookup(Rscratch);
```

The actual register that contains the original indirect branch's dynamic target can be different for different indirect branch instructions. The purpose of copying this dynamic target to register Rscratch is to ensure that when control enters the switch table lookup routine at execution time, the same fixed register (Rscratch) will contain the dynamic target that has to be looked up.

Finally, an unconditional trace exit branch is appended to the bottom of the trace so that control reaching the end of the trace can exit it via a taken branch. After fixing up the branches on the trace, the result is a single-entry, multi-exit sequence of instructions with no internal control join points. Figure 3 illustrates the branch adjustments that occur after a trace is selected from the application binary.

Since traces are free of internal join points, new opportunities for optimization may be exposed that were otherwise unsafe in the original program code. The simplicity of control flow allowed within a trace also means traces can be analyzed and optimized very rapidly. In fact, the Dynamo trace optimizer is non-iterative, and optimizes a trace in only two passes: a forward pass and a backward pass. During each pass the necessary data flow information is collected as it proceeds along the fragment. Most of the optimizations performed involve redundancy removal: redundant branch elimination, redundant load removal, and redundant assignment elimination. These opportunities typically result from partial redundancies in the original application binary that become full redundancies in a join-free trace.

The trace optimizer also sinks all partially redundant instructions (i.e., on-trace redundancies) into special off-trace *compensation blocks* that it creates at the bottom of the trace. This ensures that the partially redundant instructions get executed only when control exits the trace along a specific path where the registers defined by those instructions are downward-exposed. Fragment A in Figure 5 illustrates such a case. The assignment to register r5 shown in the compensation block (thick border) could have originally been in the first trace block. This sinking code motion ensures that the overhead of executing this assignment is only incurred when control exits the fragment via the path along which that assignment to r5 is downwards exposed.

Other conventional optimizations performed are copy propagation, constant propagation, strength reduction, loop invariant code motion and loop unrolling. Dynamo also performs runtime disambiguated conditional load removal by inserting instruction guards that conditionally nullify a potentially redundant load.

Note that load removal is only safe if it is known that the respective memory location is not volatile. Information about volatile variables may be communicated to Dynamo through the symbol table. In the absence of any information about volatile variables, load removal transformations are conservatively suppressed.

4.3 Fragment code generation

The fragment code generator emits code for the trace IR into the fragment cache. The emitted code is referred to as a *fragment*. The fragment cache manager (discussed in Section 6) first allocates sufficient room in the fragment cache to generate the code.

A trace IR may be split into multiple fragments when it is emitted into the fragment cache. This is the case, for example, if a direct conditional branch is encountered on the trace, which was converted from the application's original indirect branch instruction by the trace optimizer (see Section 4.2). Such a branch splits the trace into two fragments. The predicted on-trace target of the original indirect branch, which is the instruction immediately following this branch on the trace, starts a separate fragment.

Virtual registers may be used in the IR but the trace optimizer retains their original machine register mappings. The register allocator attempts to preserve the original machine register mappings to the extent possible when the code is finally emitted. The allocator reservers one register to hold the address of the app-context data structure (see Figure 2) when control is within the fragment. The app-context is a Dynamo internal data structure that is used to keep the application's machine state during interpretation, and also to record a snapshot of the application's machine state at the point of the last fragment cache exit to Dynamo. The trace optimizer uses the app-context as a spill area to create temporary scratch registers necessary for its optimizations. It cannot use the application's runtime stack as a spill area because that would interfere with stack operations generated by the static compiler that created the application binary.

Generation of the fragment code from the trace IR involves two steps: emitting the fragment body, and emitting the fragment exit stubs. Emitting the fragment body involves straightforward generation of the code corresponding to the trace IR itself. After that, a unique exit stub is emitted for every fragment exit branch and fragment loop-back branch. The exit stub is a piece of code that transfers control from the fragment cache to the Dynamo interpreter in a canonical way, as outlined below:

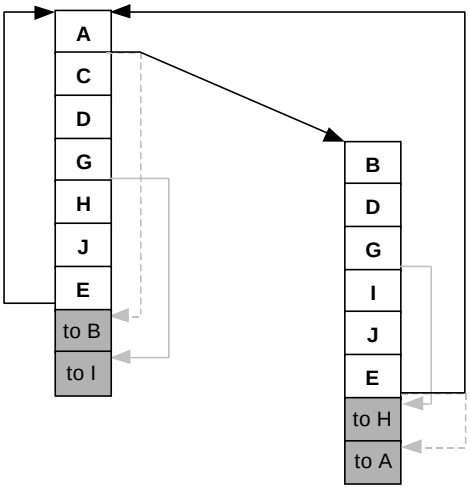


Figure 4. Example of fragment linking

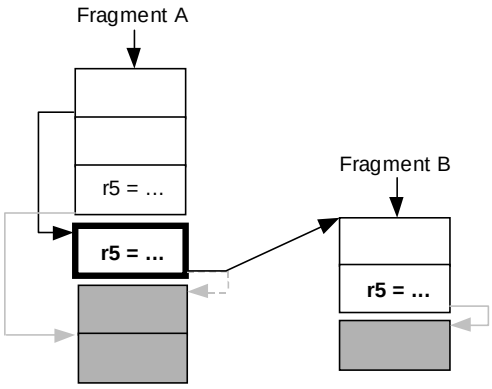


Figure 5. Example of link-time optimization

```
spill Rlink to app-context;
branch & link to interpreter; // sets Rlink to the following PC
<ptr to linkage info for this exit branch>
```

Each stub can be entered by only one fragment exit branch. The stub code first saves the link register (Rlink) to the app-context. It then does a branch and link to the entry point of the Dynamo interpreter, which sets the Rlink register to the fragment cache address following this branch. The Dynamo interpreter will take a snapshot of the application's machine state (with the application's original Rlink value being taken from the app-context data structure) prior to starting interpretation. The end of the exit stub beyond the branch and link instruction contains a pointer to linkage information for the fragment exit branch associated with the stub. When control exits the fragment to the Dynamo interpreter, the interpreter consults this linkage information to figure out the next application address at which it should start interpretation. The value of the Rlink register contains the address of the location containing the pointer to the linkage information for the current fragment exit.

5. Fragment Linking

After the fragment code is emitted into the fragment cache, the new fragment is linked to other fragments already in the fragment cache. Linking involves patching a fragment exit branch so that its taken target is the entry point of another fragment, instead of to its exit stub.

As an example, suppose the trace BDGIJE in Figure 3 (a) now becomes hot (B is a valid start-of-trace by our definition, when it is entered via an exit from the earlier hot trace ACDGHJE). Figure 4 illustrates the linking that occurs after the fragment corresponding to the BDGIJE trace is emitted into the fragment cache. Linked branches are shown as dark arrows, and their original unlinked versions are indicated as dashed light arrows.

Fragment linking is essential for performance, because it prevents expensive exits from the fragment cache back to the Dynamo interpreter. In our prototype implementation on the PA-8000 for example, disabling fragment linking results in an order of magnitude slowdown (by an average factor of 40 for the SpecInt95 benchmarks).

Fragment linking also provides an opportunity for removing redundant compensation code from the source fragment involved in the link. Recall that the trace optimizer sinks on-trace redundancies into compensation blocks, so that these instructions are only executed when control exits the fragment along a particular path (see Section 4.2). Fragment A in Figure 5 illustrates such a case, where the assignment to r5 shown in the compensation block (thick border) was originally in the first block before it was sunk into its compensation block. As part of the linkage information that is kept at each fragment exit stub (the shaded boxes in Figure 5), a mask of on-trace redundant register assignments along that particular fragment exit is maintained. In Figure 5, this mask would be kept in the exit stub corresponding to the compensation block, and bit 5 of the mask would be set. A similar mask of killed register assignments at the top of every fragment is also maintained as part of the Dynamo internal data structure that keeps fragment-related information. At link-time, if a register appears in both masks, the instruction that last defined it in the source fragment's compensation block is dead and can be removed. This is illustrated in Figure 5, where the assignment to r5 in Fragment A's compensation block can be deleted because r5 is defined before being used on entry to Fragment B.

While the advantages of linking are clear, it also has some disadvantages that impact other parts of the Dynamo system. For instance, linking makes the removal of individual fragments from the fragment cache expensive, because all incoming branches into a fragment must first be unlinked first. Linking also makes it difficult to relocate fragments in the fragment cache memory after they have been emitted. This might be useful for instance to do periodic de-fragmentation of the fragment cache memory.

6. Fragment Cache Management

Dynamo cannot afford to do complicated management of the fragment cache storage, because of the overheads this would incur. We could avoid storage management altogether by simply expanding the size of the fragment cache as needed. But this has several undesirable effects. For example, one of the advantages of collecting hot traces in a separate fragment cache is the improved instruction cache locality and TLB utilization that can result from keeping the working set close together in memory. This advantage could go away if over time, the hot traces that make up the current working set are spread out over a large area of fragment cache memory. Clearly, the ideal situation where the fragment cache only contains the traces that make up the current working set is difficult

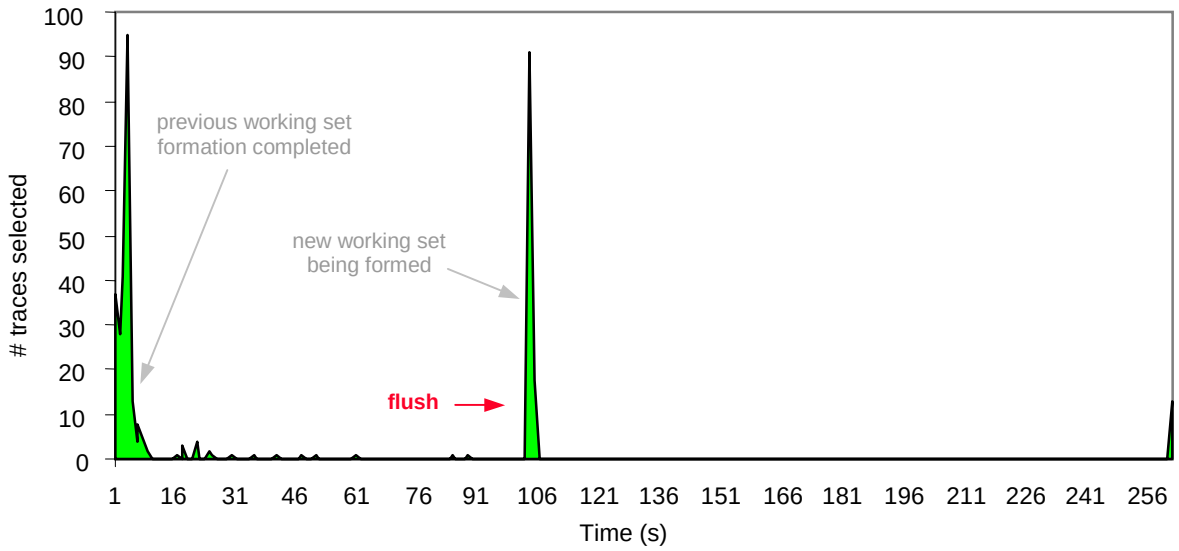


Figure 6. Dynamic trace selection rate for m88ksim, showing a sharp change in the working set ~106 sec into its execution

to achieve. The overhead of implementing an LRU type scheme to identify cold fragments would be too expensive as well. Moreover, as pointed out earlier, any policy that only removes a few fragments would incur the expense of having to unlink every incoming branch into these fragments.

Dynamo instead employs a novel pre-emptive flushing heuristic to periodically remove cold traces from the fragment cache without incurring a high penalty. A complete fragment cache flush is triggered whenever Dynamo recognizes a sharp increase in the fragment creation rate (or hot trace selection rate). The rationale here is that a sharp rise in new fragment creation is very likely indicative of a significant change in the working set of the program that is currently in the fragment cache. Since control is predominantly being spent in Dynamo during this stage, the fragment cache flush is essentially "free". Figure 6 illustrates this scenario for the SpecInt95 m88ksim benchmark. Since all fragments are removed during a fragment cache flush, no unlinking of branches needs to be done.

The pre-emptive flushing mechanism has other useful side effects. All fragment-related data structures maintained for internal bookkeeping by Dynamo are tied to the flush, causing these memory pools to be reset as a side effect of a pre-emptive flush. A pre-emptive flush thus serves as an efficient garbage collection mechanism to free dynamic objects associated with fragments that are likely to have dropped out of the current working set. If some fragments belonging to the new working set are inadvertently flushed as a result, they will be regenerated by Dynamo when those program addresses are encountered later during execution. Regeneration of fragments allows Dynamo to adapt to changes in the application's branch biases. When a trace is re-created, Dynamo may select a different tail of instructions from the same start-of-trace point. This automatic "re-biasing" of fragments is another useful side effect of the pre-emptive cache flushing strategy.

7. Signal Handling

Optimizations that involve code reordering or removal, such as dead code elimination and loop unrolling, can create a problem if a signal arrives while executing the optimized fragment, by making it difficult or impossible for Dynamo to recreate the original signal context prior to the optimization. This can create complications for precise signal delivery. For example, the application might arm a signal with a handler that examines or even modifies the machine context at the instant of the signal. If a signal arrives at a point where a dead register assignment has been removed, the signal context is incomplete.

Dynamo intercepts all signals, and executes the program's signal handler code under its control, in the same manner that it executes the rest of the application code (box K in Figure 1). This gives Dynamo an opportunity to rectify the signal context that would otherwise be passed directly to the application's handler by the operating system. Asynchronous signals (such as keyboard interrupts, etc., where the signal address is irrelevant) are treated differently from synchronous signals (such as segment faults, etc., where the signal address is critical).

If an asynchronous signal arrives when executing a fragment, the Dynamo signal handler will queue it and return control back to the fragment cache. All queued asynchronous signals are processed when the next normal fragment cache exit occurs. This allows Dynamo to provide a proper signal context to the application's handler since control is not in the middle of an optimized fragment at the time the signal context is constructed.

In order to bound asynchronous signal handling latency, the Dynamo signal handler unlinks all linked branches on the current fragment prior to resuming execution of the fragment. To disconnect self-loops in a similar manner, the fragment generator emits an exit stub for each self-loop branch in addition to the exit stubs for the fragment exit branches. Unlinking the current fragment forces the next fragment exit branch to exit the fragment cache via the exit stub, preventing the possibility of control spinning within the fragment cache for an arbitrarily long period of time before the queued signals are processed. This feature allows Dynamo to operate in environments where soft real-time constraints must be met.

Synchronous signals on the other hand are problematic, because they cannot be postponed. A drastic solution is to suppress code removing and reordering transformations altogether. A more acceptable alternative is to use techniques similar to that developed for debugging of optimized code to de-optimize the fragment code before attempting to construct the synchronous signal context.

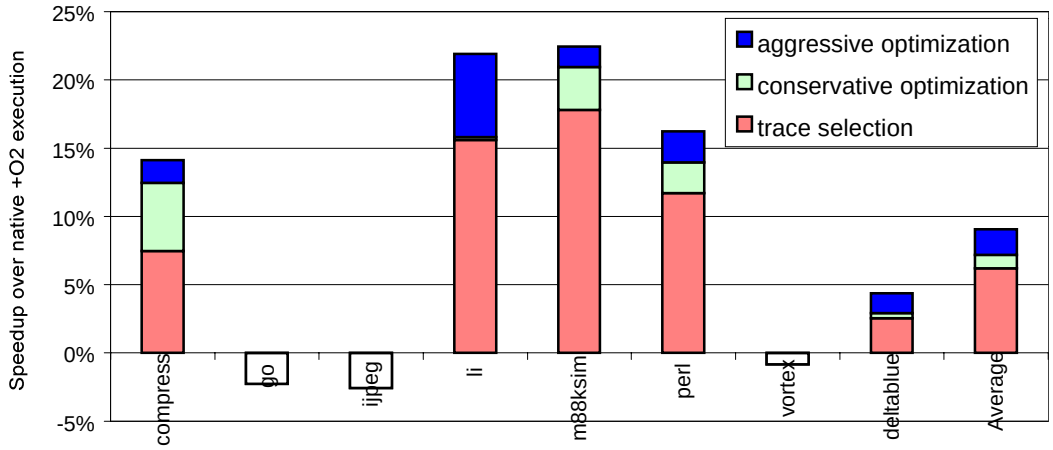


Figure 7. Speedup of +O2 optimized PA-8000 binaries running on Dynamo, relative to the identical binaries running standalone. The contributions from dynamic inlining due to trace selection, conservative trace optimization and aggressive trace optimization are shown. Dynamo bails out to direct native execution on *go* and *vortex*.

Fortunately, the problem of de-optimizing is much simpler in Dynamo since only straight-line fragments are considered during optimization. Optimization logs can be stored along with each fragment that describes compensation actions to be performed upon signal-delivery, such as the execution a previously deleted instruction. This is presently an ongoing effort in the Dynamo project.

Our prototype currently implements a less ambitious solution to this problem, by dividing trace optimizations into two categories, *conservative* and *aggressive*. Conservative optimizations allow the precise signal context to be constructed if a synchronous fault occurs while executing the fragment. Aggressive optimizations on the other hand cannot guarantee this. Examples of conservative optimizations include constant propagation, constant folding, strength reduction, copy propagation and redundant branch removal. The aggressive category includes all of the conservative optimizations plus dead code removal, code sinking and loop invariant code motion. Certain aggressive optimizations, like redundant load removal, can sometimes be incorrect, if the load is from a volatile memory location.

Dynamo's trace optimizer is capable of starting out in its aggressive mode of optimization, and switching to conservative mode followed by a fragment cache flush if any suspicious instruction sequence is encountered. Unfortunately, the PA-RISC binary does not provide information about volatile memory operations or information about program-installed signal handlers. So this capability is currently unused in Dynamo. In a future version of Dynamo, we plan to investigate ways to allow the generator of Dynamo's input native instruction stream to provide hints to Dynamo. Dynamo can use such hints if they are available, but will not rely on them for operation.

8. Performance Data

For performance evaluation we present experiments on several integer benchmarks. Dynamo incurs a fixed startup overhead for allocating and initializing its internal data structures and the fragment cache. The startup overhead could probably be improved through more careful engineering. But for the purposes of this study, we use benchmarks that long enough to allow the startup and initialization overhead to be recouped. This section presents data comparing the performance of running several integer benchmarks on Dynamo to the identical binary executing directly on the processor. Our benchmark set includes the SpecInt95 benchmarks[2] and a commercial C++ code called *deltablue*, which is an incremental constraint solver [28]. The programs were compiled at the +O2 optimization level (equivalent to the default –O option) using the product HP C/C++ compiler. This optimization level includes global intraprocedural optimization. Performance measurements were based on wall clock time on a lightly loaded single-processor HP PA-8000 workstation [21] running the HP-UX 10.20 operating system.

Figure 7 shows the speedup that Dynamo achieves over +O2 optimized native program binaries running without Dynamo. For these runs, Dynamo was configured to use a fixed size 150 Kbyte fragment cache, which is flushed when sharp changes occur to the trace selection rate or there is no room to generate new fragments. Details about the performance impact of varying the fragment cache size are outside the scope of this paper and can be found elsewhere [2]. As the figure indicates, Dynamo achieves considerable speedup in some cases, over 22% in *li* and *m88skim*, about 18% in *perl*, and about 14% in *compress*. These four programs have relatively stable working sets, a fact that dynamic optimization can exploit very well. The average overall speedup is about 9%. A significant portion of the performance gains come from the act of selecting a trace and forming a fragment out of it, that is, from the implied partial procedure inlining and improved

[2] Our experiments do not include the SpecInt95 *gcc* benchmark. This benchmark actually consists of repeated runs of *gcc* on a number of input files, and the individual runs are too short running to qualify for our performance study (less than 60 seconds on the PA-8000). To understand the performance characteristics of *gcc*, we modified the *gcc* program to internally loop over the input files, thus resulting in a single long invocation of *gcc*. We do not show data for the modified *gcc* because it does not represent the original benchmark, but it's performance characteristics are comparable to that of *go* for all of the data shown here.

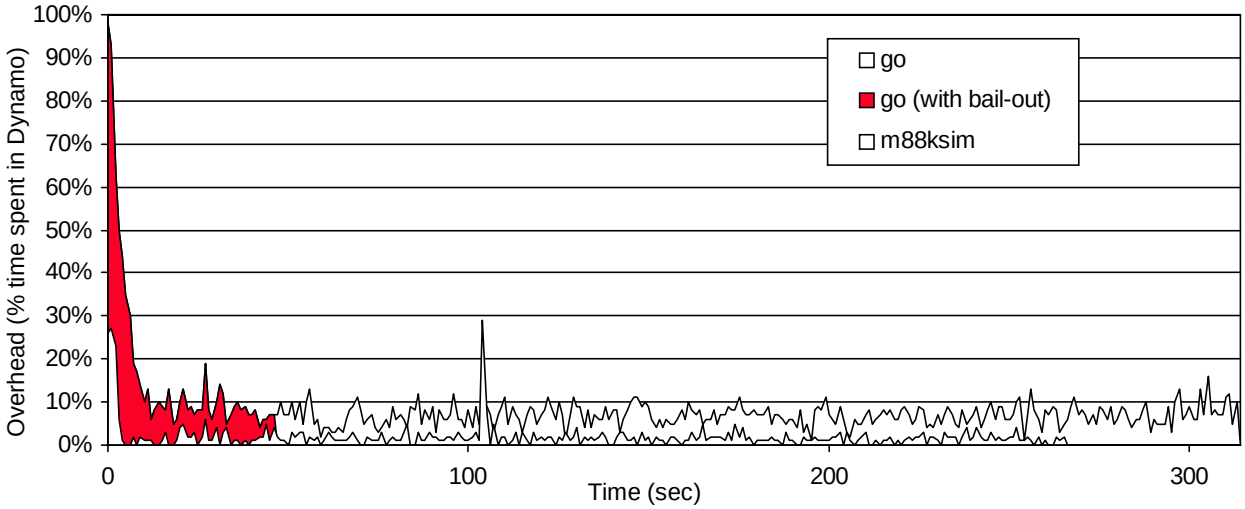


Figure 8. Illustration of bail-out. Dynamo bails out on *go* ~45 sec into its execution, after which *go* runs directly on the processor without incurring any Dynamo overhead. *m88ksim* is shown for comparison as a case where Dynamo does not bail out.

code layout in the fragment cache. Fragment optimization accounts for approximately 3% of the total gains on average, and one-third of this is due to conservative (signal and volatile-memory safe) optimizations. Note however, that if we ignore the inputs on which Dynamo bails out (as discussed shortly), the average contribution due to trace optimization is around 5%.

Dynamo does not achieve performance improvements on programs *go*, *ijpeg* and *vortex*. Dynamo's startup time is a non-negligible fraction of the total runtime of *ijpeg*, as *ijpeg* does not run long enough to recoup Dynamo's startup overhead before starting to provide any performance benefit. In the case of *go* and *vortex* that run for a long time, the problem is the lack of a stable working set. A relatively high number of distinct dynamic execution paths are executed in these benchmarks [4]. Frequently changing dynamic execution paths result in an unstable working set, and Dynamo spends too much time selecting traces without these traces being reused sufficiently in the cache to offset the overhead of its own operation.

Fortunately, since Dynamo is a native-to-native optimizer, it can use the original input program binary as a fallback when its overhead starts to get too high. Dynamo constantly monitors the ratio of time spent in Dynamo over time spent in the fragment cache. If this ratio stays above a tolerable threshold for a prolonged period of time, Dynamo assumes that the application cannot be profitably optimized at runtime. At that point Dynamo *bails-out* by loading the application's app-context to the machine registers and jumping to an application binary address. From that point on the application runs directly on the processor, without any further dynamic optimization. Bail-out allows Dynamo to come close to break-even performance even on "ill-behaved" programs with unstable working sets. This is illustrated in the graph in Figure 8 for the benchmark *go*. The Dynamo overhead for a relatively well-behaved application, *m88ksim*, is also shown for comparison.

Figure 9 shows Dynamo's performance on binaries compiled with higher optimization levels. The figure shows the program runtimes with and without Dynamo, for three optimization levels: +O2 (same as –O), +O4, and profile-based +O4 +P (i.e., +O4 with a prior profile collection run). At level +O4, the HP C compiler performs global interprocedural and link-time optimization. At level +O4 +P the compiler performs +O4 optimizations based on profile information gathered during a prior +O4 run. However, compile-time increases very significantly from +O2 to +O4, and the ability to debug the binary is lost. Because of this, most software vendors are reluctant to enable higher optimization levels, in spite of the performance advantages they offer.

The data in Figure 9 shows that Dynamo finds performance improvement opportunities even in highly optimized binaries. In fact, on this set of benchmarks, Dynamo is able to raise the average performance of +O2 compiled binaries to a level that slightly exceeds the performance of their +O4 compiled versions running without Dynamo! This performance boost comes in a transparent fashion, without the creator of the binary having to do anything special. The fact that Dynamo finds performance improvement opportunities even in +O4 optimized binaries is not as surprising as it first seems, because Dynamo primarily focuses on runtime performance opportunities that a static compiler would find difficult to exploit.

In some programs (such as *li* and *perl*), Dynamo is able to boost the performance of even profile-feedback compiled binaries (+O4 +P). On average however, the benefits of Dynamo disappear once static optimization is enhanced with profile information. This is to be expected, as the most beneficial inlining and other path-sensitive optimizations have been already made at compile-time.

As pointed out in the introduction, the goal of this study is to establish the limits of Dynamo's capabilities in an extreme setting, where the quality of the input program code is good. In compiling these benchmarks, the static compiler had all of the program sources available, and no dynamically linked libraries were used. Using good quality compiled code as input forced the development

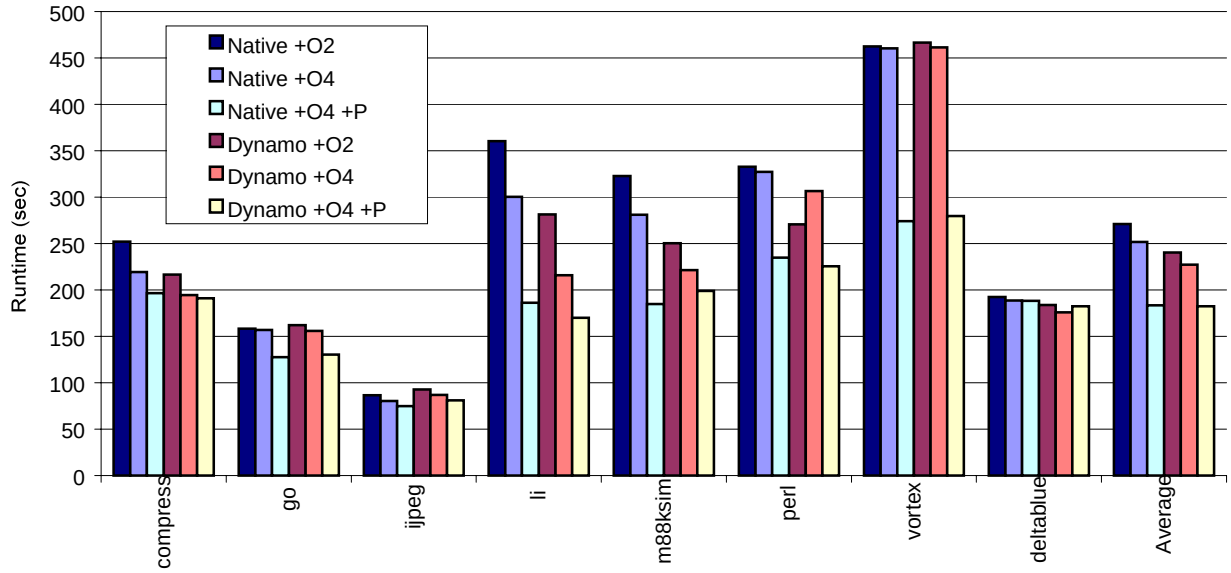


Figure 8. Dynamo performance on native binaries compiled at higher optimization levels (the first 3 bars for each program correspond to the native runs without Dynamo, and the next 3 bars correspond to the runs on Dynamo)

effort to focus on fine-tuning the engineering of the Dynamo system.

It should be emphasized that the performance data shown here is very specific to the quality of the code produced by the PA-8000 compiler, and to the PA-8000 processor implementation. Although the hot trace selection and dynamic optimization can be expected to provide benefits in general, the actual impact in terms of wall-clock performance improvement will vary from target to target. On the deeply pipelined PA-8000 for example, the branch misprediction penalty is 5 cycles, and indirect branches (including returns) are always mispredicted. Indirect branch removal therefore makes a big contribution toward Dynamo's performance gains on the PA-8000. On the other hand, the PA-8000 has a large instruction cache (1 Mbyte), so the gains from improved I-cache locality in the software fragment cache code are unlikely to be significant. However, the processor has a unified instruction and data TLB with only 96 entries, so the reduction in TLB pressure due to better locality of the working set in the fragment cache can contribute to a performance boost.

9. Related Work

In focusing on native-to-native runtime optimization, Dynamo is a fundamentally different approach from past work on dynamic compilation. Just-in-time compilers delay all compilation until runtime [6][11][10]. Selective dynamic compilation [1][9][23][13][22][26][16][24] is a staged form of compilation that restricts dynamic compilation to selected portions of code identified by user annotations or source language extensions. In these cases, the static compiler prepares the dynamic compilation process as much as possible by generating templates that are instantiated at run-time by a specialized dynamic compiler.

In contrast to both just-in-time and selective dynamic compilation, Dynamo separates that task of compilation, which occurs prior to execution, from dynamic optimization, which occurs entirely at runtime and without requiring user assistance. Dynamo's input is an already compiled native instruction stream, that is re-optimized to exploit performance opportunities that manifest themselves at runtime.

A lot of work has been done on dynamic translation as a technique for non-native system emulation [8][30][5][31][12][17]. The idea is to lower emulation overhead by caching native code translations of frequently interpreted regions. Unlike such binary translators, Dynamo is not concerned with translation. The Dynamo approach does however allow one to couple a fast lightweight translator that emits native code to Dynamo, which then becomes a backend optimizer.

There are several implementations of *offline binary translators* that also perform native code optimization [7][29]. These generate profile data during the initial run via emulation, and perform background translation together with optimization of hot spots based on the profile data. The benefit of the profile-based optimization is only available during subsequent runs of the program and the initial profile-collecting run may suffer from worsened performance.

Hardware solutions for a limited form of runtime code optimization are now commonplace in modern superscalar microprocessors [21][25][19]. The optimization unit is a fixed size instruction window, with the optimization logic operating on the critical execution path. The Trace Cache is another hardware alternative that can be extended to do superscalar-like optimization off the critical path [27][15]. Dynamo offers the potential for a purely software alternative, which could allow it to be tailored to specific application domains, and cooperate with the compiler or JIT in ways that hardware dynamic optimizers cannot.

10. Conclusion

Dynamo is a novel performance delivery mechanism. It complements the compiler's traditional strength as a static performance improvement tool by providing a dynamic optimization capability. In contrast to other approaches to dynamic optimization, Dynamo works transparently, requiring no user intervention. This fact allows Dynamo to be bundled with a computer system, and shipped as a client-side performance delivery mechanism, whose activation does not depend on the ISVs (independent software vendors) in the way that traditional compiler optimizations do.

This paper demonstrates that it is possible to engineer a practical software dynamic optimizer that provides a significant performance benefit even on highly optimized executables produced by a static compiler. The key is to focus the optimization effort on opportunities that are likely to manifest themselves only at runtime, and hence those that a static compiler might miss.

We are currently investigating applications of Dynamo's dynamic optimization technology in many different areas. One of the directions we are exploring is to export an API to the application program, so that a "Dynamo-aware" application can use the underlying system in interesting ways. This might be useful for example to implement a very low-overhead profiler, or a JIT compiler. From Dynamo's perspective, user and/or compiler hints provided via this API might allow it to perform more comprehensive optimizations that go beyond the scope of individual traces. Finally, we are also looking at the problem of transparent de-optimization at runtime.

11. Acknowledgements

Since the inception of the Dynamo project, many people have influenced our thinking. We would particularly like to thank Bill Buzbee, Wei Hsu, Lacky Shah, Giuseppe Desoli, Paolo Faraboschi, Geoffrey Brown and Stefan Freudenberger for numerous technical discussions. Finally, we are grateful to Josh Fisher and Dick Lampman for their encouragement and support of this project.

12. References

[1] Auslander, J., Philipose, M., Chambers, C., Eggers, S.J., and Bershad, B.N. 1996. Fast, effective dynamic compilation. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI'96*).

[2] Bala, V., Duesterwald, E., and Banerjia, S. 1999. Transparent dynamic optimization: The design and implementation of Dynamo. *Hewlett Packard Laboratories Technical Report HPL-1999-78. June 1999.*

[3] Bala V., and Freudenberger, S. 1996. Dynamic optimization: the Dynamo project at HP Labs Cambridge (project proposal). HP Labs internal memo, Feb 1996.

[4] Ball, T., and Larus, J.R. 1996. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29), Paris*. 46-57.

[5] Bedichek, R. 1995. Talisman: fast and accurate multicomputer simulation. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.*

[6] Chambers, C., and Ungar, D. 1989. Customization: optimizing compiler technology for Self, a dynamically-typed object-orientied programming language. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*. 146-160.

[7] Chernoff, A., Herdeg, M., Hookway, R., Reeve, C., Rubin, N., Tye, T., Yadavalli, B., and Yates, J. 1998. FX!32: a profile-directed binary translator. *IEEE Micro, Vol 18, No. 2, March/April 1998.*

[8] Cmelik, R.F., and Keppel, D. 1993. Shade: a fast instruction set simulator for execution profiling. *Technical Report UWCSE-93-06-06, Dept. Computer Science and Engineering, University .of Washington*.

[9] Consel, C., and Noel, F. 1996. A general approach for run-time specialization and its application to C. In *Proceedings of the 23th Annual Symposium on Principles of Programming Languages.* 145-156.

[10] Cramer, T., Friedman, R., Miller, T., Seberger, D., Wilson, R., and Wolczko, M. 1997. Compiling Java Just In Time. *IEEE Micro*, May/Jun 1997.

[11] Deutsch, L.P. and Schiffman A.M. 1984. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages*. 297-302.

[12] Ebcioglu K., and Altman, E.R. 1997. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*. 26-37.

[13] Engler, D.R. 1996. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI'96*).

[14] Fisher, J., and Freudenberger, S. 1992. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 5)*. Oct 1992. 85-95.

[15] Friendly, D.H., Patel, S.J., and Patt., Y.N. 1998. Putting the fill unit to work: dynamnic optimizations for trace cache microprocessors. In *Proceedings of the 31st Annual Internation Symposium on Microarchitecture (MICRO-31), Dallas*. 173-181.

[16] Grant, B., Philipose, M., Mock, M., Chambers, C., and Eggers, S.J. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation*. 293-303.

[17] Herold, S.A. 1998. Using complete machine simulation to understand computer system behavior. *Ph.D. thesis, Dept. Computer Science, Stanford University.*

[18] Hwu, W.W., Mahlke, S.A., Chen, W.Y., Chang, P. P., Warter, N.J., Bringmann, R.A., Ouellette, R.Q., Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G., and Lavery, D.M. 1993.The superblock: an effective structure for VLIW and superscalar compilation. *The Journal of Supercomputing 7*, (Jan.). 229-248.

[19] Keller, J. 1996. The 21264: a superscalar Alpha processor with out-of-order execution. Presented at the *9th Annual Microprocessor Forum*, San Jose, CA.

[20] Kelly, E.K., Cmelik, R.F., and Wing, M.J. 1998. Memory controller for a microprocessor for detecting a failure of speculation on the physical nature of a component being addressed. *U.S. Patent 5,832,205, Nov. 1998.*

[21] Kumar, A. 1996. The HP PA-8000 RISC CPU: a high performance out-of-order processor. In *Proceedings of Hot Chips VIII*, Palo Alto, CA.

[22] Leone, M. and Dybvig, R.K. 1997. Dynamo: a staged compiler architecture for dynamic program optimization. *Technical Report #490, Dept. of Computer Science, Indiana University.*

[23] Leone, M. and Lee, P. 1996. Optimizing ML with run-time code generation. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation*. 137-148.

[24] Marlet, R., Consel, C., and Boinot, P. Efficient incremental run-time specialization for free. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*. 281-292.

[25] Papworth, D. 1996. Tuning the Pentium Pro microarchitecture. *IEEE Micro*, (Apr.). 8-15.

[26] Poletta, M., Engler, D.R., and Kaashoek, M.F. 1997. tcc: a system for fast flexible, and high-level dynamic code generation. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*. 109-121.

[27] Rotenberg, E., Bennett, S., and Smith, J.E. 1996. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29), Paris*. 24-35.

[28] Sannella, M., Maloney, J., Freeman-Benson, B., and Borning, A. 1993. Multi-way versus one-way constraints in user interfaces: experiences with the Deltablue algorithm. *Software – Practice and Experience 23*, 5 (May). 529-566.

[29] Sites, R.L., Chernoff, A., Kirk, M.B., Marks, M.P., and Robinson, S.G. Binary Translation. *Digital Technical Journal, Vol 4, No. 4, Special Issue, 1992.*

[30] Stears, P. 1994. Emulating the x86 and DOS/Windows in RISC environments. In *Proceedings of the Microprocessor Forum*, San Jose, CA.

[31] Witchel, E. and Rosenblum R. 1996. Embra: fast and flexible machine simulation. In *Proceedings of the SIGMETRICS '96 Conference on Measurement and Modeling of Computer Systems*. 68-78

The Technology Behind Crusoe™ Processors

Low-power x86-Compatible Processors Implemented with Code Morphing™ Software

Alexander Klaiber
Transmeta Corporation

January 2000

Property of:
Transmeta Corporation
3940 Freedom Circle
Santa Clara, CA 95054 USA
(408) 919-3000
http://www.transmeta.com

Summary

In January of 2000, Transmeta Corporation introduced the Crusoe™ processors, an x86-compatible family of solutions that combines strong performance with remarkably low power consumption. As might be expected, a new technology for designing and implementing microprocessors underlies the development of these products. As might not be expected, the new technology is fundamentally software-based: the power savings come from replacing large numbers of transistors with *software*.

The Crusoe processor solutions consist of a hardware engine logically surrounded by a software layer. The engine is a very long instruction word (VLIW) CPU capable of executing up to four operations in each clock cycle. The VLIW's native instruction set bears no resemblance to the x86 instruction set; it has been designed purely for fast low-power implementation using conventional CMOS fabrication. The surrounding software layer gives x86 programs the impression that they are running on x86 hardware. The software layer is called Code Morphing™ software because it dynamically "morphs" x86 instructions into VLIW instructions. The Code Morphing software includes a number of advanced features to achieve good system-level performance. Code Morphing support facilities are also built into the underlying CPUs. In other words, the Transmeta designers have judiciously rendered some functions in hardware and some in software, according to the product design goals and constraints. Different goals and constraints in future products may result in different hardware-software partitioning.

Transmeta's Code Morphing technology changes the entire approach to designing microprocessors. By demonstrating that practical microprocessors can be implemented as hardware-software hybrids, Transmeta has dramatically expanded the design space that microprocessor designers can explore for optimum solutions. Microprocessor development teams may now enlist software experts and expertise, working largely in parallel with hardware engineers to bring products to market faster. Upgrades to the software portion of a microprocessor can be rolled out independently from the chip. Finally, decoupling the hardware design from the system and application software that use it frees hardware designers to evolve and eventually replace their designs without perturbing legacy software.

Technology Perspective

The Transmeta designers have decoupled the x86 instruction set architecture (ISA) from the underlying processor hardware, which allows this hardware to be very different from a conventional x86 implementation. For the same reason, the underlying hardware can be changed radically without affecting legacy x86 software: each new CPU design only requires a new version of the Code Morphing software to translate x86 instructions to the new CPU's native instruction set.

For the initial Transmeta products, models TM3120 and TM5400, the hardware designers opted for minimal space and power. By eliminating roughly three quarters of the logic transistors that would be

required for an all-hardware design of similar performance, the designers have likewise reduced power requirements and die size. However, future hardware designs can emphasize different factors and accordingly use different implementation techniques.

Finally, the Code Morphing software itself offers opportunities to improve performance without altering the underlying hardware. The current system is a first-generation embodiment of a new technology that can be further optimized with experience and experimentation. Because the Code Morphing software would typically reside in standard Flash ROMs on the motherboard, improved versions can even be downloaded into processors in the field.

Crusoe Processor Fundamentals

With the Code Morphing software handling x86 compatibility, Transmeta hardware designers created a very simple, high-performance, VLIW engine with two integer units, a floating point unit, a memory (load/store) unit, and a branch unit. A Crusoe processor long instruction word, called a *molecule*, can be 64 bits or 128 bits long and contain up to four RISC-like instructions, called *atoms*. All atoms within a molecule are executed in parallel, and the molecule format directly determines how atoms get routed to functional units; this greatly simplifies the decode and dispatch hardware. Figure 1 shows a sample 128-bit molecule and the straightforward mapping from atom slots to functional units. Molecules are executed in order, so there is no complex out-of-order hardware. To keep the processor running at full speed, molecules are packed as fully as possible with atoms. In a later section, we describe how the Code Morphing software accomplishes this.

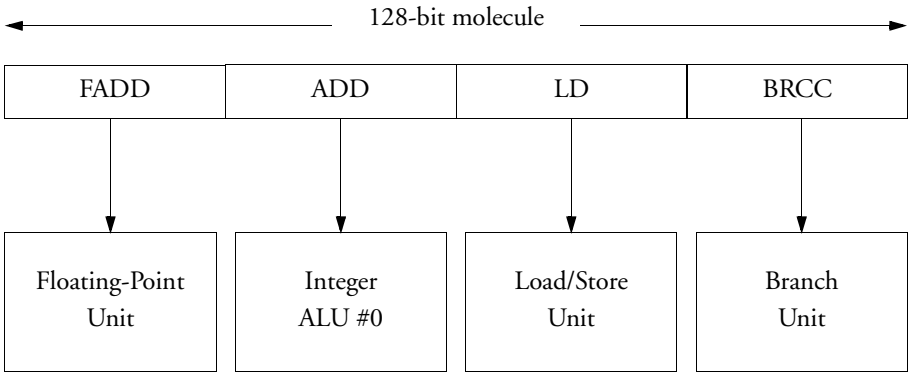


Figure 1. A molecule can contain up to four atoms, which are executed in parallel.

The integer register file has 64 registers, `%r0` through `%r63`. By convention, the Code Morphing software allocates some of these to hold x86 state while others contain state internal to the system, or can be used as temporary registers, e.g., for register renaming in software. In the assembly code examples in this paper,

we write one molecule per line, with atoms separated by semicolons. The destination register of an atom is specified first; a "`.c`" opcode suffix designates an operation that sets the condition codes. Where a register holds x86 state, we use the x86 name for that register (e.g., `%eax` instead of the less descriptive `%r0`).

Superscalar out-of-order x86 processors, such as the Pentium II and Pentium III processors, also have multiple functional units that can execute RISC-like operations (micro-ops) in parallel. Figure 2 depicts the hardware these designs use to translate x86 instructions into micro-ops and schedule (dispatch) the micro-ops to make best use of the functional units. Since the dispatch unit reorders the micro-ops as required to keep the functional units busy, a separate piece of hardware, the in-order retire unit, is needed to effectively reconstruct the order of the original x86 instructions, and ensure that they take effect in proper order. Clearly, this type of processor hardware is much more complex than the Crusoe processor's simple VLIW engine.

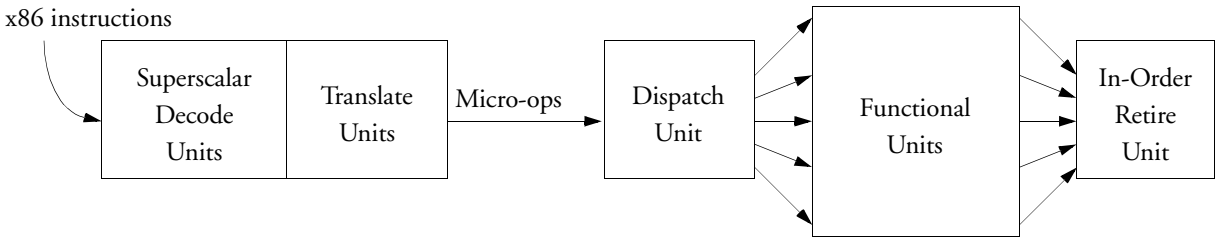


Figure 2. Conventional superscalar out-of-order CPUs use hardware to create and dispatch micro-ops that can execute in parallel.

Because the x86 instruction set is quite complex, the decoding and dispatching hardware requires large quantities of power-hungry logic transistors; the chip dissipates heat in rough proportion to their numbers. Table 1 compares the sizes of Intel mobile and Crusoe processor models.

	Mobile PII	Mobile PII	Mobile PIII	TM3120	TM5400
Process	.25m	.25m shrink	.18m	.22m	.18m
On-chip L1 Cache	32KB	32KB	32KB	96KB	128KB
On-chip L2 Cache	0	256KB	256KB	0	256KB
Die Size	130mm^2	180mm^2	106mm^2	77mm^2	73mm^2

Table 1. The Code Morphing software simplifies chip hardware.

Viewing power dissipation as heat, Figure 3 and Figure 4 contrast the operating temperatures of a Pentium III and a Crusoe processor running a software DVD (Digital Versatile Disk) player. The model TM5400 requires no active cooling, whereas the Pentium III processor can heat to the point of failure if it is not aggressively cooled.

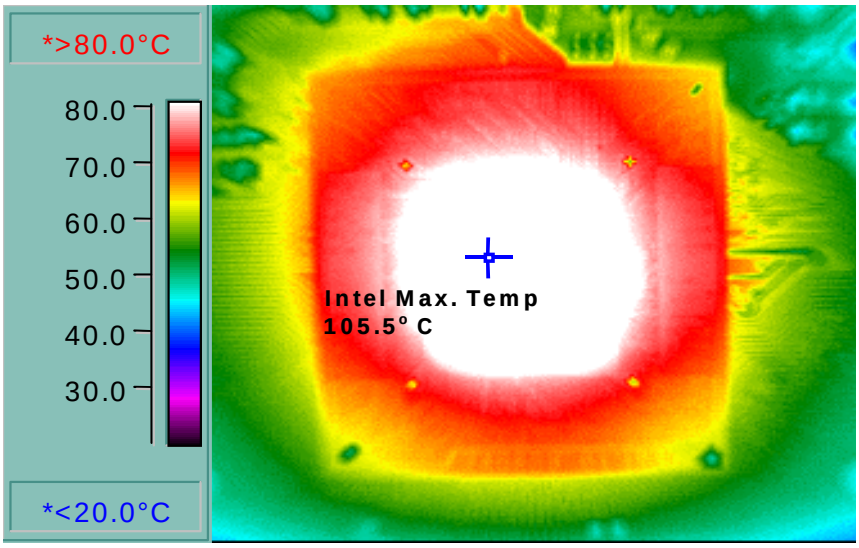


Figure 3. A Pentium III processor plays a DVD at 105° C (221° F).

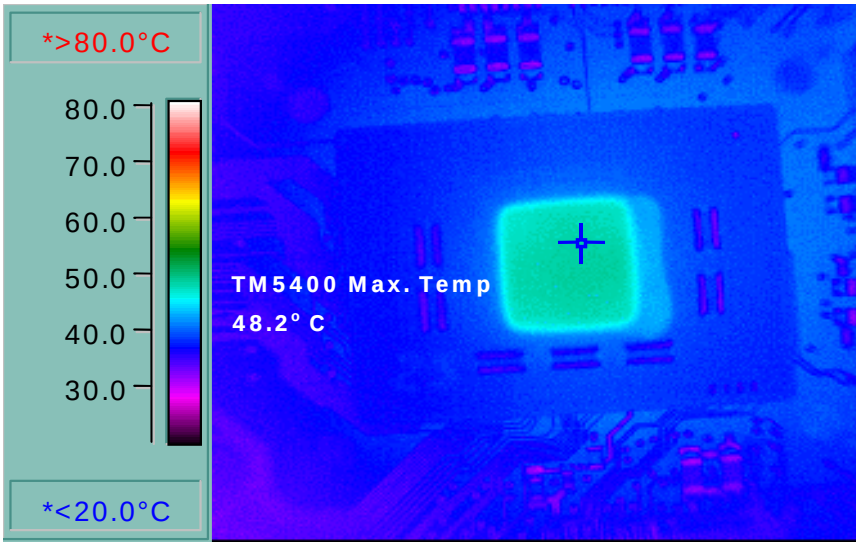


Figure 4. A Crusoe processor model TM5400 plays a DVD at 48° C (118° F).

The Code Morphing Software

The Code Morphing software is fundamentally a dynamic translation system, a program that compiles instructions for one instruction set architecture (in this case, the x86 *target* ISA) into instructions for another ISA (the VLIW *host* ISA). The Code Morphing software resides in a ROM and is the first program to start executing when the processor boots. The Code Morphing Software supports ISA, and is the only thing x86 code sees;; the only program written directly for the VLIW engine is the Code Morphing software itself. Figure 5 shows the relationship between x86 code, the Code Morphing software, and a Crusoe processor.

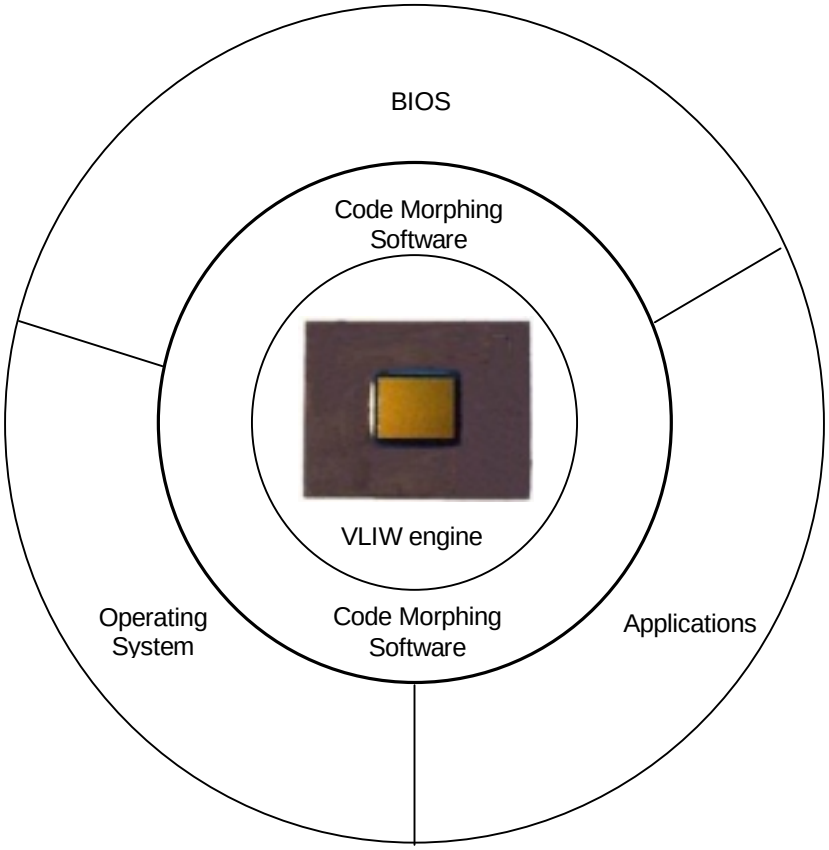


Figure 5. The Code Morphing software mediates between x86 software and the Crusoe processor.

Because the Code Morphing software insulates x86 programs—including a PC's BIOS and operating system—from the hardware engine's native instruction set, that native instruction set can be changed arbitrarily without affecting any x86 software at all. The only program that needs to be ported is the Code Morphing software itself, and that work is done once for each architectural change, by Transmeta. The feasibility of this concept has already been demonstrated: the native ISA of the model TM5400 is an

enhancement (neither forward nor backward compatible) of the model TM3120's ISA and therefore runs a different version of Code Morphing software. The processors are different because they are aimed at different segments of the mobile market: the model TM3120 is aimed at Internet appliances and ultra-light mobile PCs, while the model TM5400 supports high-performance, full-featured 3-4lb. mobile PCs.

Coincidentally, hiding the chip's ISA behind a software layer also avoids a problem that has in the past hampered the acceptance of VLIW machines. A traditional VLIW exposes details of the processor pipeline to the compiler, hence any change to that pipeline would require all existing binaries to be recompiled to make them run on the new hardware. Note that even traditional x86 processors suffer from a related problem: while old applications will run correctly on a new processor, they usually need to be recompiled to take full advantage of the new processor implementation. This is not a problem on Crusoe processors, since in effect, the Code Morphing software always transparently "recompiles" and optimizes the x86 code it is running.

The flexibility of the software-translation approach comes at a price: the processor has to dedicate some of its cycles to running the Code Morphing software, cycles that a conventional x86 processor could use to execute application code. To deliver good practical system performance, Transmeta has carefully designed the Code Morphing software for maximum efficiency and low overhead.

Drawing the Hardware-Software Line

Virtualizing an x86 CPU is a challenging undertaking because of the complexity of the x86 architecture. Choosing which functions to implement in hardware and which in software is a major engineering challenge, involving issues such as cost and complexity, overall performance and power consumption. Clearly, there are many possible choices, influenced by market demands, or the latest hardware technologies available.

For its initial products, Transmeta has drawn the line between hardware and software so that software handles the complex task of decoding x86 instructions and generating explicitly parallel molecules, which the hardware executes using a very simple, high-speed, VLIW engine. A few unique hardware features, described later in this paper, were added to better support dynamic translation. The hardware-software line might be drawn differently for another kind of product, for example, a high-end server processor.

Decoding and Scheduling

Conventional x86 superscalar processors fetch x86 binary instructions from memory and decode them into micro-operations, which are then reordered by out-of-order dispatch hardware and fed to the functional units for parallel execution.

In contrast (besides being a software rather than a hardware solution), Code Morphing can translate an entire group of x86 instructions at once, creating a *translation*, whereas a superscalar x86 translates single instructions in isolation. Moreover, while a traditional x86 translates each x86 instruction every time it is executed, Transmeta's software translates instructions *once*, saving the resulting translation in a *translation cache*. The next time the (now translated) x86 code is executed, the system skips the translation step and directly executes the existing optimized translation.

Implementing the translation step in software as opposed to hardware opens up new opportunities and challenges. Since an out-of-order processor has to translate and schedule instructions every time they execute, it must do so very quickly. This seriously limits the kinds of transformations it can perform. The Code Morphing approach, on the other hand, can *amortize* the cost of translation over many executions, allowing it to use much more sophisticated translation and scheduling algorithms. Likewise, the amount of power consumed for the translation process is amortized, as opposed to having to pay it on every execution. Finally, the translation software can *optimize* the generated code and potentially reduce the number of instructions executed in a translation. In other words, Code Morphing can speed up execution while at the same time reducing power!

Caching

The translation cache, along with the Code Morphing code, resides in a separate memory space that is inaccessible to x86 code. (For better performance, the Code Morphing software copies itself from ROM to DRAM at initialization time.) The size of this memory space can be set at boot time, or the operating system can make the size adjustable.

As with all caching, the Code Morphing software's technique of reusing translations takes advantage of "locality of reference". Specifically, the translation system exploits the high *repeat rates* (the number of times a translated block is executed on average) seen in real-life applications. After a block has been translated once, repeated execution "hits" in the translation cache and the hardware can then execute the optimized translation at full speed.

Some benchmark programs attempt to exercise a large set of features in a small amount of time, with little repetition—a pattern that differs significantly from normal usage. (When was the last time you used every other feature of Microsoft Word exactly once, over a period of a minute?) The overhead of Code Morphing translation is obviously more evident in those benchmarks. Furthermore, as an application executes, Code Morphing "learns" more about the program and improves it so it will execute faster and faster. Today's benchmarks have not been written with a processor in mind that gets faster over time, and may "charge" Code Morphing for the learning phase without waiting for the payback. As a result, some benchmarks do not accurately predict the performance of Crusoe processors.

On typical applications, due to their high repeat rates, Code Morphing has the opportunity to optimize execution and amortize any initial translation overhead. As an example, consider a multimedia

application such as playing a DVD—before the first video frame has been drawn, the DVD decoder will have been fully translated and optimized, incurring no further overhead during the playing time of the DVD. In summary, we find that the Crusoe processor's approach of caching translations delivers excellent performance in real-life situations.

Filtering

It is well known that in typical applications, a very small fraction of the application's code (often less than 10%, sometimes as little as 1%) accounts for more than 95% of execution time. Therefore, the translation system needs to choose carefully how much effort to spend on translating and optimizing a given piece of x86 code. Obviously, we want to lavish the optimizer's full attention on the most frequently executed code but not waste it on code that executes only once.

The Code Morphing software includes in its arsenal a wide choice of execution modes for x86 code, ranging from interpretation (which has no translation overhead at all, but executes x86 code more slowly), through translation using very simple-minded code generation, all the way to highly optimized code (which takes longest to generate, but which runs fastest once translated). A sophisticated set of heuristics helps choose among these execution modes based on dynamic feedback information gathered during actual execution of the code.

Prediction and Path Selection

One of the many ways in which the Code Morphing software can gather feedback about the x86 program is to *instrument* translations: the translator adds code whose sole purpose is to collect information such as block execution frequencies, or branch history. This data can be used later to decide when and what to optimize and translate. For example, if a given conditional x86 branch is highly biased (e.g., usually taken), the system can likewise bias its optimizations to favor the most frequently taken path. Alternatively, for more balanced branches (taken as often as not, for example), the translator can decide to speculatively execute code from both paths and select the correct result later. Analogously, knowing how often a piece of x86 code is executed helps decide how much to try to optimize that code. It would be extremely difficult to make similar decisions in a traditional hardware-only x86 implementation.

Making a Translation

To conclude this section, we illustrate by way of a simple example how the Code Morphing system translates a chunk of x86 code into equivalent code for the Crusoe processor's VLIW engine.[1] Assume that the filtering and path selection algorithms have chosen the following four x86 instructions, (A) through (D), for translation.

```
A. addl %eax,(%esp)          // load data from stack, add to %eax
B. addl %ebx,(%esp)          // ditto, for %ebx
C. movl %esi,(%ebp)          // load %esi from memory
D. subl %ecx,5               // subtract 5 from %ecx register
```

In a first pass, the *frontend* of the translation system decodes the x86 instructions and translates them into a simple sequence of atoms. At this stage, it is still fairly easy to discern the correspondence between the original and generated code. (Registers `%r30` and `%r31` are used as temporaries for the memory-load operations.)

```
ld %r30,[%esp]                // load from stack, into temporary
add.c %eax,%eax,%r30          // add to %eax, set condition codes.
ld %r31,[%esp]
add.c %ebx,%ebx,%r31
ld %esi,[%ebp]
sub.c %ecx,%ecx,5
```

In a second pass, the *optimizer* applies well-known compiler optimizations to the code, such as common subexpression elimination, loop invariant removal, or dead code elimination (including unnecessary settings of the condition codes). This exemplifies optimizations that a hardware-only x86 implementation cannot do: a software-based translation system can actually *eliminate* atoms from the instruction stream, rather than just reorder them. In this example, all but the last setting of the condition code is unnecessary (allowing for greater flexibility in scheduling), and one of the load atoms is redundant, leaving fewer atoms to be executed.

```
ld %r30,[%esp]                // load from stack only once
add %eax,%eax,%r30
add %ebx,%ebx,%r30            // reuse data loaded earlier
ld %esi,[%ebp]
sub.c %ecx,%ecx,5             // only this last condition code needed
```

In a final pass, the *scheduler* reorders the remaining atoms and groups them into individual molecules. This process is similar to what out-of-order processors do in their dispatch hardware. However, by using software to schedule the code, it becomes feasible to use more effective scheduling algorithms and

1. As a reminder, we write VLIW code one molecule per line, with atoms separated by semicolons. The destination register of an atom is specified first; a ".c" opcode suffix designates an operation that sets the condition codes.

consider a larger window of instructions than would be reasonable in hardware. After scheduling, we have reduced the four original x86 instructions down to just two molecules:

```
1. ld %r30,[%esp];    sub.c %ecx,%ecx,5
2. ld %esi,[%ebp];    add %eax,%eax,%r30;    add %ebx,%ebx,%r30
```

There are two important points to observe here:

- Though the molecules are executed in-order by the hardware, they perform the work of the original x86 instructions out of order.

- The molecules explicitly encode the instruction-level parallelism, hence they can be executed by a simple (and hence fast and low-power) VLIW engine; the hardware need not perform any complex instruction reordering itself.

Crusoe Hardware Support for Code Morphing

Dynamic translation on conventional processors would result in unsatisfactory performance. In contrast, the Crusoe hardware can achieve excellent performance because it has been designed specifically with dynamic translation in mind. Below, we discuss three simple hardware features that support exceptions, speculation, optimization of memory operations, and self-modifying code.

Exceptions and Speculation

Without special hardware support, it is in general very difficult for a dynamic translation system to correctly model the exception semantics of the target ISA while at the same time achieving high performance. The reason is that exception semantics impose severe constraints on instruction scheduling. Consider again the example from the previous section, where the following x86 code:

```
A. addl %eax,(%esp)
B. addl %ebx,(%esp)
C. movl %esi,(%ebp)
D. subl %ecx,5
```

was translated into the following two molecules:

```
1. ld %r30,[%esp];    sub.c %ecx,%ecx,5
2. ld %esi,[%ebp];    add %eax,%eax,%r30;    add %ebx,%ebx,%r30
```

In the x86 ISA, exceptions are *precise*: when one instruction causes an exception, all instructions preceding it must complete before the exception is reported, and none of the subsequent instructions may complete. Observe that in the translation above, atoms occur out of order with respect to the original x86 code

order. Now imagine that during execution, the load instruction in molecule 2, corresponding to x86 instruction (C), takes a page fault. However, by that time, the processor has already executed code in molecule 1 corresponding to instruction (D), which violates the rules of precise exceptions.

Solving this problem without special hardware support unduly constrains the scheduling of host instructions, or requires extra host instructions to be issued, either of which reduces performance even in the common case where no exceptions occur.

It is worth noting at this point that out-of-order processors, too, have this problem. They typically employ complex hardware mechanisms to delay or undo the effects of micro-ops that have been executed "too soon".

The Crusoe host processor provides a much simpler hardware solution that works hand-in-hand with the Code Morphing software. All registers holding x86 state are *shadowed*, i.e., there exist two copies of each register, a *working* and a *shadow* copy. Normal atoms only update the working copy of the register. When execution reaches the end of a translation without encountering an exception, a special *commit* operation copies all working registers into their corresponding shadow registers, indeed committing the work done in the translation. On the other hand, if any x86-level exception occurs inside the translation, the runtime system undoes the effects of all molecules executed since the start of the translation. This is done via a *rollback* operation which copies the shadow register values (last committed at the end of the previous translation) back into the working registers. At this point, the Code Morphing software re-executes the x86 instructions conservatively, that is to say in their original program order, to determine the actual location of the exception.

Undoing changes to memory is slightly more complicated. The Crusoe processor handles x86 store operations by holding store data in a "gated store buffer", from which they are only released to the memory system at the time of a commit. On a rollback, stores not yet committed can simply be dropped from the store buffer. To speed the common case (no exceptions), the Crusoe hardware is designed such that commit operations are effectively "free".

Alias Hardware

The more freedom the scheduler has to move atoms around to fill molecules, the better code it can usually generate. One of the biggest limits on this freedom comes from potential dependencies between memory operations. In particular, it is often desirable to be able to reorder load instructions ahead of store instructions. However, doing that is incorrect if the load happens to use data from the preceding store, and since it is generally hard to prove otherwise at translation time, a translator often has to make overly conservative assumptions. (This is also a problem for traditional compilers and microprocessors.)

The Crusoe host provides innovative *alias* hardware that addresses this problem. When the translator moves a load operation ahead of a store operation, it converts the load into a *load-and-protect* (which in addition to loading data also records the address and size of the data loaded) and the store into a

store-under-alias-mask (which checks for protected regions). In the (unlikely) event that the store operation overwrites the previously loaded data, the processor raises an exception and the runtime system can take corrective action. Using this mechanism, it is always safe to reorder memory loads and stores. Again, Crusoe hardware provides a very simple hardware mechanism that in concert with software solves a thorny problem.

The alias hardware can be put to even better use than moving atoms around: it can help to eliminate redundant load/store atoms. Consider the case where a datum is loaded from memory twice, but there is an intervening store operation (a code sequence that is actually fairly common in processors with few registers, like the x86):

```
ld %r30,[%x]                // first load from location X
...
st %data,[%y]               // might overwrite location X
ld %r31,[%x]                // this accesses location X again
use %r31
```

As long as the intervening store operation does not overlap with the first load, the second load is redundant, but all too often a translator or compiler cannot prove that this is the case. Using the alias hardware, it is a simple matter to protect the first load, have the store check pending aliases, and eliminate the second load:

```
ldp %r30,[%x]               // load from X and protect it
...
stam %data,[%y]             // this store traps if it writes X
use %r30                    // can use data from first load
```

Notice that the use of the loaded data can now also be scheduled earlier, further speeding up the generated code. To our knowledge, no out-of-order processor can perform a similar feat!

Coping with Self-modifying Code

At times, x86 instructions in memory get overwritten, either because the operating system is loading a new program, or because an application is using self-modifying code. When this happens to code that has already been translated, the Code Morphing software needs to be notified to keep it from erroneously executing a translation for the old code. To this end, whenever the system translates a block of x86 code, it write-protects the page of x86 memory containing that code. It does so by setting a dedicated "translated" bit in that page's entry in the processor's memory management unit. (As with other details of the VLIW hardware, that bit is invisible to x86 software.) When a protected page is written to, the simplest remedy is to invalidate the affected translation(s). As the runtime system dynamically learns more about the program's behavior, it switches to more sophisticated strategies (beyond the scope of this paper).

Example: A Complex Translation

We close our review of translation technology with a slightly longer example taken from an actual x86 application running on Windows NT, illustrating more of the sophisticated capabilities of Code Morphing. The following twenty x86 instructions (which in a conventional processor would generate more than twenty micro-ops):

```
 1.          movl    %ecx,$0x3
 2.          jmp     lbl1
  ...
 3.   lbl1:  movl    %edx,0x2fc(%ebp)
 4.          movl    %eax,0x304(%ebp)
 5.          movl    %esi,$0x0
 6.          cmpl    %edx,%eax
 7.          movl    0x40(%esp,1),$0x0
 8.          jle     skip1
 9.          movl    %esi,$0x1
10.   skip1: movl    0x6c(%esp,1),%esi
11.          cmpl    %edx,%eax
12.          movl    %eax,$0x1
13.          jl      skip2
14.          xorl    %eax,%eax
15.   skip2: movl    %esi,0x308(%ebp)
16.          movl    %edi,0x300(%ebp)
17.          movl    0x7c(%esp,1),%eax
18.          cmpl    %esi,%edi
19.          movl    %eax,$0x0
20.          jnl     exit1
      exit2:
```

were translated into the following ten VLIW instructions:

```
 1.    addi    %r39,%ebp,0x2fc
 2.    addi    %r38,%ebp,0x304
 3.    ld      %edx,[%r39];      add %r27,%r38,4;     add %r26,%r38,-4
 4.    ld      %r31,[%r38];      add %r35,0,1;        add %r36,%esp,0x40
 5.    ldp     %esi,[%r27];      add %r33,%esp,0x6c;  sub.c %null,%edx,%r31
 6.    ldp     %edi,[%r26];      sel #le,%r32,0,%r35
 7.    stam    0,[%r36];         sel #l,%r24,%r35,0;  add %r25,%esp,0x7c
 8.    stam    %r32,[%r33];      add %ecx,0,3;        sub.c %null,%esi,%edi
 9.    st      %r24,[%r25];      or %eax,0,0;         brcc #lt,<exit2>
10.    br  <exit1>
```

There are several interesting points to note:

- The x86 unconditional `JMP` has no corresponding instruction in the translation: the path selector simply "follows" the branch and continues translation at the target of the `JMP`.
- Registers have been aggressively renamed in software; there is no need for a complex (and power consuming) register renamer in hardware.
- The scheduler has rearranged the instructions to execute out of order relative to the original x86 "source" code.
- The translator has replaced the two internal conditional branches with "select" instructions (which conditionally pick one of two results). In effect, the Code Morphing system is speculatively executing both legs of a branch and picking the correct result later. Reducing the number of branches is highly desirable, since they often cause inefficiencies in pipelined processors. We know of no out-of-order processor that can completely eliminate conditional branches.
- The Crusoe alias hardware has been used in the translation (in molecules 5 through 8) to hoist loads above stores and thus pack the code more effectively.

LongRun™ Power Management

Although the Code Morphing software's primary responsibility is ensuring x86 compatibility, it also provides interfaces to capabilities available only in Crusoe processor models. LongRun power management is one example—a facility in the TM5400 model that can further minimize that processor's already low power consumption.

In a mobile setting, most conventional x86 CPUs regulate their power consumption by rapidly alternating between running the processor at full speed and (in effect) turning the processor off. Different performance levels can be obtained by varying the on/off ratio (the "duty cycle"). However, with this approach, the processor may be shut off just when a time-critical application needs it. The result may be glitches, such as dropped frames during movie playback, that are perceptible (and annoying) to a user.

In contrast, the TM5400 can adjust its power consumption without turning itself off—instead, it can adjust its clock frequency on the fly. It does so extremely quickly, and without requiring an operating system reboot or having to go through a slow sequence of suspending to and restarting from RAM. As a result, software can continuously monitor the demands on the processor and dynamically pick just the right clock speed (and hence power consumption) needed to run the application—no more and no less. Since the switching happens so quickly, it is not noticeable to the user.

Finally, the Code Morphing software can also adjust the Crusoe processor's voltage on the fly (since at a lower operating frequency, a lower voltage can be used). Because power varies linearly with clock speed and by the square of the voltage, adjusting both can produce *cubic* reductions in power consumption whereas a conventional CPUs can adjust power only linearly. For example, assume an application program only requires 90% of the processor's speed. On a conventional processor, throttling back the processor speed by 10% cuts power by 10%, whereas under the same conditions, LongRun power management can reduce power by almost 30%—a noticeable advantage!

Conclusion

In 1995, Transmeta set out to expand the reach of microprocessors into new markets by dramatically changing the way microprocessors are designed. The initial market is mobile computing, in which complex power-hungry processors have forced users to give up either battery running time or performance. The Crusoe processor solutions have been designed for lightweight (two to four pound) mobile computers and Internet access devices such as handhelds and web pads. They can give these devices PC capabilities and unplugged running times of up to a day.

To design the Crusoe processor chips, the Transmeta engineers did not resort to exotic fabrication processes. Instead they rethought the fundamentals of microprocessor design. Rather than "throwing hardware" at design problems, they chose an innovative approach that employs a unique combination of hardware and software. Using software to decompose complex instructions into simple atoms and to schedule and optimize the atoms for parallel execution saves millions of logic transistors and cuts power consumption on the order of 60–70% over conventional approaches—while at the same time enabling aggressive code optimization techniques that are simply not feasible in traditional x86 implementations. Transmeta's Code Morphing software and fast VLIW hardware, working together, achieve low power consumption without sacrificing high performance for real-world applications.

Although the model TM3120 and model TM5400 are impressive first efforts, the significance of the Transmeta approach to microprocessor design is likely to become more apparent over the next several years. The technology is young and offers more freedom to innovate (both hardware and software) than conventional hardware-only designs. Nor is the approach limited to low-power designs or to x86-compatible processors. Freed to render their ideas in a combination of hardware and software, and to evolve hardware without breaking legacy code, Transmeta microprocessor designers may produce one surprise after another in the new millennium.

To learn more about the Transmeta Crusoe processor family, consult http://www.transmeta.com.

Acknowledgements

This paper represents the work of a brilliant team of fellow Transmeta engineers. The author would like to thank everyone who has contributed to this paper.

FX!32

A Profile-Directed Binary Translator

Anton Chernoff

Mark Herdeg

Ray Hookway

Chris Reeve

Norman Rubin

Tony Tye

S. Bharadwaj Yadavalli

Digital Equipment Corporation

John Yates

Chromatic Research

A new innovation from Digital allows most x86 Windows applications to run on Alpha platforms with good performance.

Because Digital's Alpha architecture provides the world's fastest processors, many applications, especially those requiring high processor performance, have been ported to it. However, many other applications are available only under the x86 architecture. We designed Digital FX!32 to make the complete set of applications, both native and x86, available to Alpha. The goal for the software is to provide fast and transparent execution of x86 Win32 applications on Alpha systems. FX!32 achieves its goal by transparently running those applications at speeds comparable to high-performance x86 platforms. Digital FX!32 is a software utility that enables x86 Win32 applications to be run on Windows NT/Alpha platforms. Once FX!32 has been installed, almost all x86 applications can be run on Alpha without special commands and with excellent performance.

Before the introduction of this software, two common techniques for running an application on a different architecture than the one for which it was originally compiled were emulation and binary translation. Each technique has an advantage, but also a drawback. Emulation is transparent and robust, but delivers only modest performance. Binary translation[1] is fast, but not transparent. For the first time, Digital FX!32 combines these technologies to provide both fast and transparent execution.

This software consists of three interoperating components. There is a runtime environment providing transparent execution, a binary translator (the background optimizer) providing high performance, and a server coordinating them. Although FX!32 is transparent and does not require user intervention, it includes a graphical interface for monitoring status and managing system resources.

The first time an x86 application runs, all of the application is emulated. Together with transparently running the application, the emulator generates an execution profile describing the application's execution history. The profile shows which parts of the application are heavily used (for each user) and which parts are unimportant or rarely used. While the first run may be slow, it "primes the pump" for additional processing. Later, after the application exits, the profile data directs the background optimizer to generate native Alpha code to replace all the frequently executed procedures. The next time the application runs, native Alpha code is used and the application executes much faster. This process repeats whenever a sufficiently enlarged profile shows that it is warranted.

Three significant innovations of Digital FX!32 include transparent operation, interface to native APIs, and, most importantly, profile-directed binary translation.

Transparent operation

When we say FX!32 is transparent, we mean two things: applications execute in the expected way (without any special commands), and interoperability with native applications works normally.

Launching x86 applications. Transparent launching of Win32 x86 applications comes from a dynamically linked library (DLL), the transparency agent. Launching an application on Windows NT always results in a call to the CreateProcess function. By intercepting calls to CreateProcess, the transparency agent can examine every image as it is about to be executed. If a call to CreateProcess specifies an x86 image, the transparency agent instead invokes the FX!32 runtime to execute the image. Although special privileges are required to install FX!32, once installed, the transparency agent, and therefore the applications themselves, run without special privileges.

Digital FX!32 inserts the transparency

0272-1732/98/$10.00 © 1998 IEEE

agent into the address space of each process. A process containing the transparency agent is said to be enabled. Once a process is enabled, any attempt to execute a Win32 x86 image causes the runtime to start executing the process. The agent propagates through the system because each attempt to create a process to run an Alpha image results in that created process being enabled. By the time a user is logged on, all top-level processes have been enabled by Digital FX!32, and any attempt to execute a Win32 x86 application invokes FX!32's runtime.

Processes are enabled by injecting a copy of the transparency agent into the process's address space, using a technique described by Richter.[2] The transparency agent's initialization routine then modifies a number of imported entry points by changing the addresses in the image import tables of all loaded modules to point to routines in the agent.

The transparency agent provides a general mechanism to change the behavior of an API routine called from Alpha code. We use this in a number of ways. For example, the behavior of the Win32 API routine LoadLibrary changes so that FX!32 loads x86 images. This is important because an attempt to load an x86 image on an NT Alpha system using the native loader results in an error. As another part of its function, FX!32 jackets the x86 image's exports so that they can be called from native Alpha code (discussed later). Finally, if FX!32's runtime is not already in memory, the transparency agent loads the runtime when it loads an x86 image.

The transparency agent we developed can be used for utilities besides Digital FX!32. For example, the transparency agent supports SPIKE (once known as OM), an Alpha native link-time optimization tool.[3] Users of SPIKE need only mark an application as interesting and every internally used library and image will be translated.

Runtime environment. The Windows NT operating system invokes the FX!32 runtime via the transparency agent whenever the user runs an x86 Win32 application. The runtime provides transparent execution because it contains an emulator that implements the entire x86 user-mode instruction set, and because it supports the complete x86 Win32 environment.

When the application first executes, Digital FX!32 has no knowledge of this application for this user and so runs it completely in the emulator. (As explained later, application calls to the x86 Win32 APIs, in fact, call corresponding native Alpha APIs.) The next execution of the application runs translated code for greater performance. The emulator remains present to interpret those x86 instructions that, for whatever reason, cannot be translated.

The rest of the transparency is provided by full support for the Win32 environment, such as multiple threads, structured exception handling, and the Microsoft component object model (COM) architecture across both the Alpha and x86 architectures. The runtime allows interfaces to all COM objects to be accessed from either x86 or Alpha code.

Runtime operation

The performance of Digital FX!32 comes from executing high-speed, native Alpha code. To secure high performance, the runtime transparently substitutes native Alpha code in place of x86 code whenever possible.

The FX!32 runtime is invoked whenever an enabled process attempts to execute an x86 image. The runtime loads the image into memory, sets up the runtime environment, and then starts emulating the image.

The runtime loader duplicates the functionality of the NT loader. This is necessary since the Alpha NT loader returns an error indicating that the image is of the wrong architecture if the loader is invoked to load an x86 image. This would have been much simpler had we been able to modify NT. Duplicating the functionality of the NT loader requires that the runtime relocate images not loaded at their preferred base address, set up shared sections, and process static thread local storage (TLS) sections.

After the image is loaded, the loader inserts pointers to the image into various lists used internally by NT. Maintaining those lists allows the native Windows NT code to treat both x86 and Alpha images identically. Fortunately, those image lists are in the user's address space, and no modification of NT is required. Unfortunately, the structure of those lists is not part of the documented Win32 interface and using them creates a dependency on the version of NT being run. This is one of a number of places where Digital FX!32 has dependencies on undocumented NT features, making it more dependent on a particular version of NT than a typical layered application would be. On the other hand, it is remarkable that Digital FX!32 implementation required no changes to NT.

Next, the image is entered into FX!32's database. The database provides the name of the translated image to be used with a given x86 image. The database is accessed by using an image ID obtained by hashing the image's header. The ID uniquely identifies the image by its contents, independent of the image's name or location in the file system. Both the runtime and the server use the image ID to access information stored in the database about the image.

If there is a translated image in the database, the runtime

Availability

Digital FX!32 is available free electronically from http://www.service.digital.com/fx32/.

This Web site contains more information on Digital FX!32 and the software itself. Additional information is available in Chernoff and Hookway,[1] Hookway and Herdeg,[2] and Thompson.[3] Digital FX!32 is a trademark of the Digital Equipment Corporation. All other trademarks and registered trademarks are the property of their respective owners.

1. A. Chernoff and R. Hookway, "Running 32-Bit x86 Applications on Alpha NT," *Proc. USENIX Windows NT Workshop*, Usenix Assoc., Sunset Beach, Calif., 1997, pp. 17-23.
2. R. Hookway and M. Herdeg, "Digital FX!32: Combining Emulation and Binary Translation," *Digital Tech. J.*, Vol. 9, No. 1, 1997, pp. 3-12.
3. T. Thompson, "An Alpha in PC Clothing," *BYTE*, Vol. 19, No. 2, Feb. 1996, pp. 195-196.

loads that image along with the original x86 image. Translated images are normal NT DLLs loaded by the native loader. A translated image contains the translated Alpha code, together with the two additional sections that define the correspondence between x86 code and Alpha code:

- A section containing relocation information for references to the x86 image. If the x86 image was not loaded at its preferred base address, those references must be relocated.
- A section containing a map between x86 and translated routine entry points. The runtime processes this map to update a hash table, indexed by x86 addresses, with entries pointing to the corresponding translated code.

Once the images are loaded, the runtime starts emulating the x86 instructions. When the emulator interprets a CALL instruction, it looks for the target x86 address in the hash table. If a corresponding translated address exists, the emulator transfers to the translated code. The emulator also generates profile data for use by the translator containing the following information:

- addresses that are targets of CALL instructions,
- source address/target address pairs for indirect jumps, and
- addresses of instructions that make unaligned references to memory.

The profile data is collected by inserting values into the runtime hash table whenever a relevant instruction is emulated. For example, when emulating the CALL instruction, the emulator records the call's target. When an image is unloaded, or when the application exits, the hash table is processed, and a profile for that image is written. The server processes this profile, merging it with any previous profiles and may invoke the translator.

Cross-architecture interoperability

Win32 applications make calls to routines that are not part of the application, specifically the Win32 API. Because these are x86 applications, they make calls by using the x86 calling conventions. NT Alpha provides the same routines, but with Alpha calling conventions. FX!32 provides a mechanism to connect the two.

Transformations are required to manage a call between a native Alpha routine and a piece of emulated or translated code. For example, x86 routines pass arguments on the stack while Alpha routines expect arguments in registers. Small code fragments called jackets, which manage the transition between the x86 and Alpha environments and calling conventions, perform these transformations.

There are two basic kinds of jackets, static and dynamic, based on how and when they are created. Static jackets are created from a defined interface known at load time. They are included as part of Digital FX!32's runtime. Most static jackets are simple and are generated automatically from documentation and header files. Some static jackets are built by hand because code is required to process the arguments in a special way. Digital FX!32 provides static jackets for the Win32 API interface, NT call-back routines, standard object linking and embedding (OLE) objects, and some selected plug-in extensions.

COM objects whose interfaces are not statically available are dynamically jacketed at runtime. These dynamic jackets are created by using type information obtained from the OLE libraries.

Interface to native APIs

Unlike Unix, in Windows NT most system APIs are part of the operating system. For example, most graphical user interface functions are built into NT system DLLs. We found that some applications, such as Microsoft Excel, spend almost half their execution time in these libraries. We knew that it was very important for Digital FX!32 to call native libraries whenever possible to achieve our performance goals.

When the NT loader loads an image, the loader "snaps" the image's imports by using symbolic information in the image to locate the addresses of the imported routines or data. The runtime duplicates this process. However, the runtime treats imports referring to entries in Alpha images specially, by redirecting them to refer to the correct jacket entry.

Each jacket contains a special illegal x86 instruction that serves as a signal to the emulator to switch into the Alpha environment by calling Alpha code at a fixed offset from the illegal x86 instruction. The basic operation of most jacket routines is to move arguments from the x86 stack to the appropriate Alpha registers, as dictated by the Alpha calling standard. Some jacket routines provide special semantics for the native routine being called. For example, the jacket for GetSystemDirectory returns the path to the x86 system directory rather than to the true system directory, so x86 applications do not overwrite native Alpha DLLs.

Jacketing the Win32 API. Previous translation utilities (for various Unix flavors) created by Digital jacketed the operating system call interface because that was the defined interface between applications and the operating system. This required jacketing an interface to about 100 system calls. Windows NT defines and documents the Win32 API (layered above the system call interface) as the interface between applications and the operating system, and Digital FX!32 jackets the complete Win32 API. Although jacketing the complete Win32 API is a significant task, it is required to guarantee correctness and provides better initial performance because the jacketed routines are native and do not need translation. As a result, Digital FX!32 provides static jackets for entries to over 50 native Alpha DLLs, including jackets for many undocumented routines. About 12,000 routines are currently jacketed.

Jacketing call-back routines. Many Windows NT routines are passed the addresses of routines to call back when an event occurs. If these values were to be passed blindly, the Windows NT Alpha code would make a call to a location containing x86 code and would certainly crash. A jacket is statically created for each procedure-pointer argument, and the address of that jacket is passed to the native Alpha code. When Alpha code calls back to its argument, the jacket enters the FX!32 runtime.

Jacketing COM objects. The most complicated jacketing

problem is associated with COM. A COM object is represented by a table of OLE function pointers. These functions often have arguments that are pointers to functions or structures containing pointers to functions. Digital FX!32 manages these objects in a way that can be used from either native Alpha or x86 code.

Jacketing plug-in extensions. For full interoperability, it is also desirable to support x86 plug-ins (add-ons or extensions defined by an application vendor) with the corresponding native application, when it is available. Each of these introduces another interface (requiring jackets) that is not defined by NT and not available at runtime. Digital FX!32 cannot load such a plug-in unless it is programmed to jacket the interfaces. The current version of Digital FX!32 jackets a few common plug-in interfaces—we are working on ways to describe arbitrary plug-in interfaces for a future release.

Runtime and background optimizer

Commercial applications typically consist of numerous executable files, called images. Some images are unique to the application, and some are shared across different applications on the system. Each time the runtime loads an x86 image, the runtime queries the database as to whether translated code exists for that image to run in place of the slower x86 code. Translated code is high-speed, native Alpha code, produced by the background optimizer after previously emulating the image under Digital FX!32.

After loading the translated code, the runtime sets up tables that correlate addresses between any x86 code and the translated code. The runtime then initiates the emulator, which starts executing the application. From careful design and alignment with the Alpha architecture, the emulator is both small and efficient. The emulator is small enough to reside mostly in the high-speed instruction cache, is optimized for the Alpha processor pipeline, and takes full advantage of the 64-bit Alpha processor registers.

As it emulates untranslated portions of x86 images, the runtime collects and saves execution profiles for subsequent use by the background optimizer. The performance of Digital FX!32 is based on this cooperation between the runtime and the background optimizer.

Coordinating the process: the server

The server manages FX!32's environment by coordinating the runtime and the background optimizer. The server acts according to Digital FX!32 defaults or according to parameters that can be specified by the user. In response to these parameters, the server manages execution profiles and invokes the background optimizer.

After an x86 image is unloaded, the server merges any new profile information with any existing profiles and compares the size of the result with any previous size. A new profile means that a previously unseen x86 image has been executed and may require optimization. An enlarged profile contains new information, indicating that the current optimized image is incomplete. In either case, the server places the image and the corresponding profile on the work list for the background optimizer.

This process is repeated each time the image runs. Figure 1 shows the execution flow among FX!32 components. When the size of the profile stabilizes (typically at two or three iterations), it indicates that virtually all executed routines in the image are translated. The image and corresponding profile are no longer placed on the work list for the background optimizer. Running the image executes high-performance, native Alpha code, rather than the slower x86 code. The image runs at its highest performance.

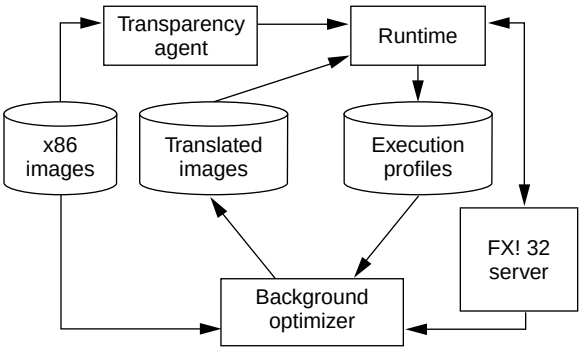


Figure 1. The flow of information among Digital FX!32 components.

Creating the speed: binary translation

The background optimizer, a third-generation, profile-directed binary translator, produces high-speed, native Alpha code from x86 code by using information gathered into profiles by the runtime. A binary translator is a program that, from the original code, produces translated native code that can be executed directly. The native Alpha code is subsequently made available to the runtime and executed the next time the image is run. It is this coordinated process that adds high performance to the transparency of execution.

Design goals. The operation and output of the background optimizer must be as transparent and robust as the runtime environment. The user never sees the operation of the background optimizer; it always presents code to the runtime that runs to correct completion. To ensure transparency, the background optimizer design allows for no assumptions, no manual initiation, and no user intervention in any question/answer cycle.

Coupled with the stringent need for transparent and flawless operation is a requirement for the highest possible performance.

Realization of the goals. The background optimizer guarantees transparent and robust operation by cooperating with the runtime to ensure a faithful representation of the x86 machine state. A coherent x86 machine state means the x86 register assignments, call/return boundaries, and the x86 stack all reflect what would be observed on actual x86 hardware at relevant observation points.

Achieving the performance goals required us to exploit the full range of modern compiler optimization techniques, which are all predicated on global optimization.

Previous binary translators operated with a poor quality

Digital FX!32 is the first system to exploit this combination of emulation, profile generation, and binary translation.

approximation of the application's control flow graph. As a consequence, they were limited to the basic block, or perhaps the extended basic block, as the fundamental unit of translation. (A basic block is a sequence of instructions with a single entry point and a single exit point.) All modern optimizing compilers require global optimization techniques that directly conflict with such a basic-block unit restriction. Therefore, removal of this restriction was the fundamental performance requirement. The background optimizer successfully removes this restriction by using profiles to organize carefully chosen groupings of basic blocks into significantly larger units, called translation units. Conceptually, a translation unit approximates a "routine" in a more traditional compiler and thus allows the full exploitation of global optimization techniques.

Profile-directed binary translators

Digital has used other binary translation techniques in the past,[1] mainly static binary translation. Our development group has extensive experience with previous binary translators. We also looked at other solutions, such as hardware engines and dynamic binary translation.

In Digital FX!32, we have developed a new approach to translation. The emulator captures an execution profile, which the binary translator subsequently uses to translate executed x86 code into native Alpha code. Since the translator runs in the background, it can use complex algorithms to improve the quality of the generated code. To our knowledge, Digital FX!32 is the first system to exploit this combination of emulation, profile generation, and binary translation. We call our approach profile-directed to contrast it with static and dynamic approaches.

Because we have the execution profile, our binary translator was easier to write, runs faster, and produces better code than any previous static binary translators. Our translator was easier to write because the complex search algorithms and heuristics used to find the code and the control flow graph were replaced by much faster and simpler lookups. Digital FX!32 produces better code because the profiles result in more accurate approximations of the control flow graph, allowing optimizations to be more effective.

Translator operation

In many ways our binary translator is a traditional high-performance compiler. However, there is an important difference. Compilers start from source level and proceed to lower the semantic level, while binary translators start with bits and raise the semantic level first to instructions and then to control flow graphs. The challenge for the translator is to produce correct and efficient code in this framework.

Locating code. The search for code begins at all the destinations of call instructions recorded in the profile. As the code is parsed, the destinations of indirect branches are resolved by looking in the profile. As a consequence, no complex and slow iterated data flow is required. The profile-directed approach needs less code in the translator even though this approach makes a more accurate determination of the location of code and control flow edges.

Since the translator builds a good approximation to the control flow graph, basic blocks can be joined into larger units. The translator contains a component called the regionizer that divides the x86 image into routines.[4] Routines are units of translation that approximate real routines in source programs.

The regionizer represents routines as a collection of regions. Each region is a contiguous range of addresses containing instructions that can be reached from an entry address of the routine. Routines end at the return statements identified by the profile. Unlike basic blocks, regions can have multiple entry points. The smallest collection of regions containing all the instructions reachable from the routine entry represents the routine. Most routines have a single region. This representation efficiently describes the division of the source image into units of translation.

Intermediate representation. The remaining translator components process the source image one routine at a time. All control flow is explicitly represented (including all the direct control flow), as well as indirect control flow recorded in the profile information. For every transfer of control that might have additional unknown destinations (such as indirect branches), the translator inserts a call to the emulator. Only the routine's entry points are entered in the x86-to-Alpha correlation table, ensuring that the emulator cannot transfer to an arbitrary block in the routine.

The emulator and translator share a canonical representation of the x86 state. In the translator, all entries into and out of the routine use explicit intermediate representation to represent the canonical x86 state. Other than at these points, the translator is free to use whatever representations for the x86 state it finds convenient. As a result, the transformations and optimizations do not have to be as conservative as in the static translators, which have to allow for the emulator transferring control to almost any basic block. This allows the translator to perform global transformations and optimizations on the whole routine.

A more accurate control flow graph, based on the profile, is vital to our performance. Each time the application executes, an indirect branch to a target not previously executed invokes the emulator. Once in the emulator, translated code is not resumed until another routine is called or the routine returns to a translated caller. The runtime then adds that fact to the profile.

The same intermediate representation has primitives for both x86 and Alpha operations. The processing of a routine starts by building a representation of the x86 code. Then, multiple transformations convert the representation from an x86 semantic model to an Alpha semantic model. Optimiza-

tion phases are interspersed with these transformations. At the end of processing each routine, the final Alpha code is assembled into the translated image.

Translation and optimization

Our goal was to handle a very large percentage of x86 applications, including those that do not follow the NT calling conventions. We knew that Digital FX!32 would need to maintain great fidelity. The translator uses a simple code generator to map x86 instructions into a correct general, but long, sequence of Alpha instructions. Then the translator uses global transformations and optimization to improve the code.

The translator uses many traditional compiler techniques. It includes optimization phases for dead-code elimination, constant propagation, common subexpression elimination, register renaming, global register allocation, instruction scheduling, and numerous peephole optimizations.

Condition code management. Most x86 instructions generate condition codes, but only rarely are they consumed. Initially, the x86 model is represented in the intermediate representation with condition code information for each instruction. Global data flow determines the lifetimes of the x86 condition codes. Explicit Alpha code is then inserted to compute only those condition codes used.

Register management. The x86 architecture uses distinct registers to access different bytes of the same underlying register. The mapping of these overlaid registers to Alpha registers uses data flow to minimize the amount of generated Alpha code. Since the x86 state only has to be canonical at routine boundaries, the various overlays of an x86 register within a routine can be maintained in separate Alpha registers to allow more efficient access. This also allows global renaming to reduce register dependencies, increasing the benefits of instruction scheduling.

Stack management. The x86 architecture has few registers, so x86 code tends to make extensive use of the x86 stack to hold temporary results. The translator analyzes memory accesses to identify storing and loading from the x86 stack. The translator assumes that when the x86 stack is popped, any data stored above the new stack pointer is dead. The translator uses this information to eliminate those unnecessary loads and stores. Any loads and stores that cannot be proved to be unaliased are not eliminated. After eliminating loads and stores, the translator coalesces increments and decrements to the x86 stack pointer to minimize the number of updates, while preserving the runtime convention that the stack is never accessed above the stack pointer.

Routine management. We have found x86 routines that walk up the stack and modify local variables of their callers, including return addresses. To make these routines work, FX!32 needs to make the x86 application see an identical stack image. The translation of a CALL instruction saves the x86 return address on the x86 stack and then calls the translated code for the routine. After the translated call, the x86 return address is on the x86 stack, and the native return address that corresponds to the x86 return address is in an Alpha register. In the usual case, the routine does not change the return address, and the translated code can pop the x86 stack and perform a native return by using the native return address. However, there are two problems to solve. First, it must be possible to determine whether the application modified the x86 return address. Second, there must be a place to save the native return address. Both problems are solved using the shadow stack.

The shadow stack resides at the top of the native Alpha stack and is maintained by the translated code and the emulator. A shadow stack frame holds the x86 and the Alpha return addresses, along with the x86 stack pointer at the time of the call. The translated code for a RET instruction uses these values to determine when it is not legal to make a native return, at which point the emulator is entered to start emulating from the modified x86 return address. The emulator consults the shadow stack when emulating RET instructions to see if translated code can be resumed. In this case, the emulator uses the Alpha return address in the shadow stack.

The emulator uses the x86 stack pointer saved in the shadow stack to remove shadow-stack frames above the current value of the x86 stack pointer. Such frames can occur if the code cuts back the x86 stack to return to an earlier caller (as is done by the longjmp C library routine). This cleanup always finishes before the emulator uses the shadow stack, ensuring the shadow stack does not overflow.

Alternative solutions

As mentioned previously, our primary design goals for Digital FX!32 were transparency and high performance. Before arriving at the coordinated combination of emulation, profile generation, and profile-directed binary translation, we examined a range of alternative solutions.[5-7]

Hardware-based solutions. One approach would have been to design a new chip that supports both the Alpha and the x86 ISAs. Similar techniques exist in a number of designs. The most popular variations on this approach use a hybrid design known as a decoupled microarchitecture. This design combines a high-performance execution core with a sophisticated x86 instruction decoder. The decoder translates x86 instructions into simpler operations that execute more efficiently. This approach can generate quite good performance on applications written for the x86. Some examples of machines that use this approach are the AMD K6, Intel Pentium Pro, and NexGen Nx586. None of these machines expose the alternative instruction set architecture (ISA) to the user, and therefore they pay the penalty of being basically CISC designs (albeit with a RISC core). This limitation could be overcome with an x86/Alpha chip that exposes both ISAs. However, we felt that Digital FX!32 could achieve good performance for x86 applications by using a totally software-based solution, avoiding the complexity of including support for the x86 ISA in future Alpha chip designs.

Software-based solutions. There are two common software alternatives that also allow applications written for one ISA to execute on a different ISA—emulation and binary translation.

Emulators. These programs, at runtime, dynamically execute instructions written in the original ISA. Many systems have successfully used emulators to run applications on platforms for which they were not targeted.[8] The major advantage of emulators is transparency. The major drawback is poor performance. For example, our x86 emulator, which we care-

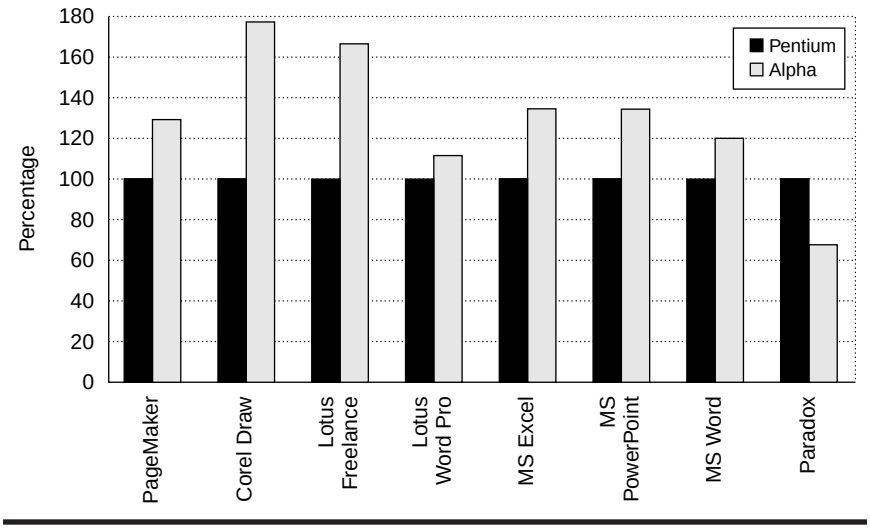


Figure 2. Relative Performance of a 500-MHz Alpha running Digital FX!32 and a 200-MHz Pentium.

fully wrote in Alpha assembler, requires an average of 45 Alpha instructions to emulate one x86 instruction (or 30 Alpha instructions per Pentium Pro micro-operation). While this is acceptable for infrequent use, it is too slow to meet our goals.

Emulators are commonly deployed in one of two ways: within a restricted environment or tightly integrated into the operating system. In a typical restricted environment, the user brings up an emulator window, and the emulator executes any application launched in that window. Our goal of transparent execution made us reject restricted environments.

In an integrated system, a modified operating system loader automatically launches the emulator whenever an emulated application is started. Windows NT has contained an emulator to run 16-bit x86 applications since it was first released on RISC platforms. Since we did not build Windows NT, we developed a scheme that allows us to launch emulated applications without needing source changes to NT.

Binary translators. These programs start with original code and produce translated native code that can be executed directly. The main advantage of binary translation is that the translated applications run at high speed. For example, after translation, Digital FX!32 executes an average of 4.4 Alpha instructions per x86 instruction (2.1 Alpha instructions per Pentium Pro micro-operation). Since the typical clock speed of an Alpha (500 to 600 MHz) is twice the clock speed of an x86 (166 to 233 MHz), it is clear that using a binary translator could achieve our goal.

Two previous types of binary translation already existed when we began to design Digital FX!32: dynamic translators and static translators.

Dynamic binary translators

Several emulators[9] have used dynamic translation, sometimes called just-in-time translation, or JIT, to achieve better performance. This approach translates small segments of an application while it is being executed. Systems using dynamic translation trade off the amount of time spent translating and the resulting benefit of the translation. Too much time spent on the translation and related processing makes the application unresponsive; too little time makes the performance slow.

Therefore, most of these systems limit the optimizations they perform to minimize the translation overhead. Dynamic translators are usually stateless, so that each time an application starts, the translator begins anew. For each application, the start of each execution serves as a training set that is then used to guide the dynamic translator. For code run only once, this is an attractive option. However, important applications are run repeatedly, and the initial training is thus repeated each time.

Static binary translators

The other existing software alternative is static translation. Here, a translator program scans the entire image and translates everything at once. We have built several static binary translators in the past, and developers and sophisticated end users have found them quite useful as a way to quickly port an application. Static binary translation is particularly useful when the source code for the application is not available or is prohibitively complex to recompile, as an interim solution while source code is being ported, or when the best possible performance is not an issue.

Static binary translator operation. The user manually invokes the translation tool to convert code from a non-Alpha ISA to Alpha. This scheme is difficult to use with an application that contains many images, because each image requires the user to manually invoke the tool. It is hard to get users to run tools that have many steps; users expect applications to "just work."

Static translators use a static approach to try to answer the following questions: what part of an image is code, what part is data, and what is the control flow graph?

Static translators separate an image into basic blocks using the following steps:

1. The static translator identifies a set of addresses considered to be the start of a basic block. It looks for addresses that meet the following criteria: they're externally visible in the text section of an image, the addresses serve as either an entry point or as the target of a relocated instruction, and they start a valid sequence of instructions that ends in a branch.
2. The static translator parses the identified basic blocks, finds the ending branch, and tries to determine the destination of the branch. Each such destination is considered the start of another basic block. For some branches, finding the destination is simple, but for others (such as indirect branches via a register), interprocedural global data flow is required. It is possible to identify a sequence of instructions as a single basic block and later find a

branch into the middle of that sequence. Thus the translator needs to iterate both the data flow calculations (to find possible values of registers) and the parsing of blocks (to find indirect branches).

3. Because the data flow calculation misses many possible values, the static translator walks over the text section and scans for missed basic blocks. It looks for any sequence of bits not part of a known basic block, but that could be parsed into a sequence of valid instructions ending in a branch.

At the end of this process, the static translator has a list of addresses in the source image that are likely to be the start of basic blocks, together with some control flow. As this sea of basic blocks is translated, the list of addresses expands into a structure called a correlation table. This table lists pairs that contain source machine addresses and the addresses of corresponding translated code.

At runtime, indirect branches are translated into a call to a library routine. This routine looks up the destination of the branch in the correlation table. If there is an entry, there is an available translation of the corresponding basic block and the library routine branches to the translation. If there is no entry, the library routine emulates up to the next branch and tries the lookup again.

Since the emulator can enter translated code at any block in the list of pairs, optimization is generally limited to single basic blocks. However, optimizations can be done across basic blocks, provided that the block is removed from the correlation table. Of course, any block disconnected from the control flow graph cannot be globally optimized.

Analysis of static binary translators. Although we were willing to use expensive techniques such as repeated full-image data flow, the static translators missed important control flow edges and sometimes saw edges that were never taken. The correlation table could contain entries for addresses that appeared to be the start of a block but were actually data. At the same time, the table could be missing entries for blocks reached by indirect branches.

The performance of static translations tends to depend upon how well destinations of indirect branches can be resolved. When we started to build Digital FX!32, we realized that the style of programming used in many x86 applications would make resolving these destinations very difficult. Static translators also provide no transparent way of executing an application, requiring a full translation was manually done before the application could be executed. This led us to consider a profile-directed translator in conjunction with an emulator that generates profiles.

What does not work?

The most obvious way in which Digital FX!32 is not transparent is that x86 applications are installed by using an add/remove x86 program applet visually and functionally similar to the NT add/remove program applet. Another nontransparency is that the first execution of an application is much slower than the second execution.

There are some things that the initial version of Digital FX!32 was not designed to do. Digital FX!32 only executes application code. It does not execute drivers, so a native driver is required for any peripheral device installed on an Alpha system. Digital FX!32 does not provide complete support for x86 NT services (services from the NT control panel services applet) because such services are enabled only when they are started after FX!32's server. We hope to remove this restriction in future versions of Digital FX!32.

Digital FX!32 does not support the NT debug API. Supporting that interface would require the ability to rematerialize the x86 state after every x86 instruction, severely limiting optimizations that could be performed by the translator. This limitation is similar to the trade-off in optimizing compilers where debugging is restricted when optimizations are turned on. Since Digital FX!32 does not support the debug interface, applications requiring it do not run under Digital FX!32. Those applications are mostly x86 development environments, and it probably makes sense to run them on an x86 anyway.

Performance

Figure 2 shows relative performance on a set of benchmarks for a 200-MHz Pentium and a 500-MHz Alpha with similar configurations. A larger number indicates higher performance. For the Alpha, we took the timings at the second execution of the benchmark using the same input data. For these benchmarks, the Alpha running Digital FX!32 provides roughly the same performance as a 200-MHz Pentium. These benchmarks are the set of applications included in the well-known PC benchmark, BapCo SysMark 32.

Of course, no small set of benchmarks characterizes the performance of a system. Even so, when executing translated x86 applications, we have consistently measured performance on a 500-MHz Alpha in the range between a 200-MHz Pentium and a 200-MHz Pentium Pro.

SINCE IT WAS FIRST RELEASED two years ago, Digital FX!32 has been used by thousands of NT/Alpha users, with over 13,000 copies downloaded from FX!32's Web site alone. At least five commercial redistributors of NT/Alpha systems have made FX!32 available on their own Web sites. FX!32 has also been factory-installed software on all NT/Alpha workstations shipped by Digital. Although, it's become the most widely used of all profile-directed software tools, development work remains. Specifically, FX!32's operation is still not completely transparent to the user. To install an x86 application on NT/Alpha, the user must check a box in the add/remove programs dialog box. Work remains to be done on the background optimizer so that its operation need not be scheduled, and the code produced by the optimizer is still not as close in performance to native Alpha code as we would like.

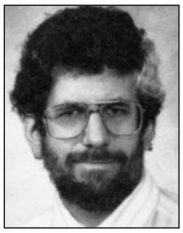
Acknowledgments

Building a product like Digital FX!32 required a lot of hard work by some extremely talented people. Many of these people contributed the ideas described in this article. The following engineers were part of Digital FX!32's development team: Rick Bagley, Jim Cambell, George Darcy, Paul Drongowski, Tom Evans, Charlie Greenman, Alan Krzywicki, Maurice Marks, Srinivasan Murari, Brian Nelson, Scott Robin-

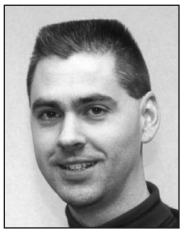
son, Joyce Spencer, John Strange, Monty Vanderbilt, and Andrew Wison.

References

1. R.L. Sites et al., "Binary Translation," *Digital Tech. J.,* Vol. 4, No. 4, 1992, pp. 137-152.
2. J. Richter, *Advanced Windows NT*, Microsoft Press, Redmond, Wash., 1994.
3. R. Cohn et al., "SPIKE: An Optimizer for Alpha/NT Executables," *Proc. USENIX Windows NT Workshop*, Usenix Assoc., Sunset Beach, Calif., 1997, pp. 9-15.
4. R. Hank and B.R. Rau, "Region-Based Compilation: An Introduction and Motivation," *Proc. MICRO-28,* IEEE Computer Society Press, Los Alamitos, Calif., 1995, pp. 158-168.
5. R. Bedichek, "Some Efficient Architecture Simulation Techniques," *Proc. USENIX Conf.,* Usenix Assoc., Sunset Beach, Calif., 1990, pp. 53-63.
6. R.F. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," *Proc. ACM Sigmetrics Conf.*, ACM Press, NY, 1994, pp. 128-137.
7. T.R. Halfhill, "Emulation: RISC's Secret Weapon," *BYTE*, Vol. 19, No. 4, Apr. 1994, pp. 119-130.
8. B. Case, "Rehosting Binary Code for Software Portability," *Microprocessor Report*, Vol. 3, No. 1, Jan. 1989, pp. 4-9.
9. L.P. Deutsch and A.M. Schiffman, "Efficient Implementation of the Smalltalk-80 System," *Proc. 11th Ann. Principles of Programming Languages*, 1983, pp. 297-302.

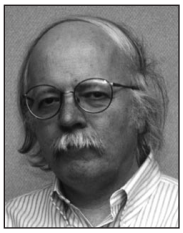


Anton Chernoff is a senior consulting engineer in the Alpha Migration Tools (AMT) group at Digital Equipment Corporation in Littleton, Massachusetts. He has been working on binary translation since 1991. Chernoff is an ACM member.

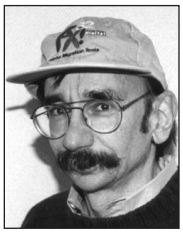


Mark Herdeg is a principal software engineer in the AMT group at Digital Equipment Corporation. He is a member of FX!32's development team, having contributed to its original design, and works primarily on the runtime environment. He has received several patents related to binary translation and emulation environments.

Ray Hookway is a consulting engineer in the AMT group at Digital Equipment Corporation. His interests include binary translation and compilation. He received MS and PhD degrees from Case Western Reserve University.

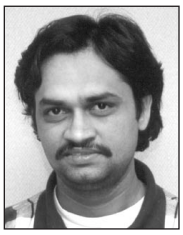


Chris Reeve is a member of the technical staff at Digital Equipment Corporation. He is a member of the FX!32 group working on the binary translator. His research interests include optimizing compilers for high-performance microprocessors and parallel processing. Reeve has BS and MS degrees from MIT.

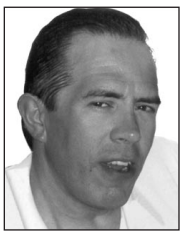


Norman Rubin is a member of the technical staff in the Advanced Products Unit in Digital Equipment Corporation's AMT group. Rubin's technical interests include optimizing compilers, profile-directed compilation, computer architecture, object-oriented programming, and Java. Rubin received his MS and PhD degrees in computer science from the Courant Institute of NYU. He is a member of both the IEEE and ACM.

Tony Tye is a software engineer in Digital Equipment Corporation's AMT group. He has been a member of FX!32's engineering team since its inception, primarily working on the offline optimizer component. Tye received his MS and PhD degrees in computation from the University of Manchester Institute of Science and Technology, England.



S. Bharadwaj Yadavalli is a software engineer in FX!32's AMT group at Digital Equipment Corporation. His current technical interests include binary translation, compiler optimizations, computer architecture, and related high-performance system tool development issues. Yadavalli obtained his MTech in computer science from Jawaharlal Nehru University, India, and is currently completing his PhD at University of Saskatchewan, Canada.



John Yates works on mass-market computing at Chromatic Research. He is happiest working at the hardware/software boundary, lapsing regularly into assembler (such as FX!32's x86 interpreter). He has designed and shipped a computer microarchitecture and instruction set, various runtime models, and numerous compiler components (including an SSA intermediate language).

Direct questions concerning this article to Norm Rubin, Digital Equipment Corporation, rubin@amt.tay1.dec.com.

Efficient Implementation of the Smalltalk-80 System

L. Peter Deutsch
Xerox PARC, Software Concepts Group

Allan M. Schiffman
Fairchild Laboratory for Artificial Intelligence Research

ABSTRACT

The Smalltalk-80* programming language includes dynamic storage allocation, full upward funargs, and universally polymorphic procedures; the Smalltalk-80 programming system features interactive execution with incremental compilation, and implementation portability. These features of modern programming systems are among the most difficult to implement efficiently, even individually. A new implementation of the Smalltalk-80 system, hosted on a small microprocessor-based computer, achieves high performance while retaining complete (object code) compatibility with existing implementations. This paper discusses the most significant optimization techniques developed over the course of the project, many of which are applicable to other languages. The key idea is to represent certain runtime state (both code and data) in more than one form, and to convert between forms when needed.

*Smalltalk-80 is a trademark of the Xerox Corporation.

BACKGROUND

The Smalltalk-80 system is an object-oriented programming language and interactive programming environment. The Smalltalk-80 language includes many of the most difficult-to-implement features of modern programming languages: dynamic storage allocation, full upward funargs, and call-time binding of procedure names to actual procedures based on dynamic type information, sometimes called *message-passing*. The interactive environment includes a full complement of programming tools: compiler, debugger, editor, window system, and so on, all written in the Smalltalk-80 language itself. A detailed overview of the system appears in [SCG 81]. [Goldberg 83] is a technical reference for both the non-interactive programmer and the system implementor; [Goldberg 84] is a reference manual for the interactive system.

Special Difficulties

The standard Smalltalk-80 system implementation is based on an ideal *virtual machine* or v-machine. The compiler generates code for this machine, and the implementor's documentation describes the system as an interpreter for the v-machine instruction set, similar to the Pascal P-system [Ammann 75] [Ammann 77]. One unusual feature of the Smalltalk-80 v-machine is that it makes runtime state such as procedure activations visible to the programmer as data objects. This is similar to the "spaghetti stack" model of Interlisp [XSIS 83], but more straightforward: Interlisp uses a programmer-visible indirection mechanism to reference procedure activations, whereas the Smalltalk-80 programmer treats procedure activations just like any other data objects.

The Smalltalk-80 language approaches programming with generic data types through *message-passing* and dynamic typing. To invoke a procedure (*method* in Smalltalk-80 terminology), a message is sent to a data object (the *receiver*), which selects the method to be executed. This means that a method address must be found at runtime. At a given lexical point in the code, only the message name (*selector*) is known. To perform a *message-send*, the data type (*class*) of the receiver is extracted, and the selector is used as a hash index into a table of the *message dictionary* of the class, which maps selectors to methods. The task of *method-lookup* is complicated by the *inheritance* property of classes -- a class may be defined as a *subclass* to another, inheriting all of the methods of the superclass. If the initial method-lookup fails, the lookup algorithm tries again using the message dictionary of the superclass of the receiver's class, continuing in this way up the class hierarchy until a method corresponding to the selector is found or the top of the inheritance hierarchy is reached.

The Smalltalk-80 language uses the organization of objects into classes to provide strong information hiding. Only the methods associated with a given class (and its subclasses) can access directly the state of an instance of that class. All access from "outside" must be through messages. Because of this, a Smalltalk-80 program must often make procedure calls to access state where languages such as Pascal could compile a direct access to a field of a record. This makes the performance of the method-lookup algorithm even more critical.

Implementation Outline

The purpose of the research described here was to build a Smalltalk-80 system with acceptable performance on a relatively inexpensive, microprocessor-based computer; specifically, to discover how to implement the basic data and code objects of the Smalltalk-80 system in a way that still conformed to the v-machine specification, but were more suitable for conventional hardware. (As of early 1982, the only implementations that ran at acceptable speed were on non-commercial, user-microprogrammable machines, as described in [Krasner 83] [Lampson 81].) The system specification in [Goldberg 83] includes the definition of internal data structures and object code representation for the virtual machine. Indeed, much of the system code depends on these definitions. We chose to take these definitions as given, rather than alter the system code.

This was motivated partly by a desire to retain object-code portability, and partly by a desire not to complicate the description of the Smalltalk-80 machine model.

The single principle that underlies all the results reported here is *dynamic change of representation*. By this we mean that the same information is represented in more than one (structurally different) way during its lifetime, being converted transparently between representations as needed for efficient use at any moment. An important special case of this idea is *caching*: one can think of information in a cache as a different representation of the same information (considering contents and accessing information together) in the backup memory. In the implementation described in this paper, we applied this principle to several different kinds of runtime information in the Smalltalk-80 system.

* We dynamically translate v-code (i.e., code in the instruction set of the v-machine) into code that executes directly on the hardware without interpretation, the native code or *n-code*. Translated code is cached: it is regenerated rather than paged.

* We represent procedure activation records (*contexts* in Smalltalk-80 parlance) in either a machine-oriented form, when they are being used to hold execution state, or in the form of Smalltalk-80 data objects, when they are being treated as such.

* We use several different caches to speed up the polymorphic search required at each procedure invocation. In the best case, which applies over 90% of the time, a Smalltalk-80 procedure invocation requires only one comparison operation in addition to a conventional procedure linkage.

* Using the techniques in [Deutsch&Bobrow 76], we represent reference count information for automatic storage management in a way that eliminates approximately 85% of the reference counting operations required by a standard implementation.

CODE TRANSLATION

Targeting code to a portable v-machine has been used in other language implementations. Usually v-code targeting is used only to avoid having multiple (one per target machine) code-generation phases of the compiler; a secondary benefit is that v-code is usually much more compact than code for any real machine. Since the Smalltalk-80 compiler is just one tool available in the same interactive environment used for execution, and other tools besides the compiler must be able to examine the machine state, the v-machine approach is even more attractive in reducing the cost of rehosting.

PERFORMANCE ISSUES

To rehost the system, an implementor must emulate the v-machine on the target hardware, either in microcode or in software. This normally incurs a severe performance penalty arising from several factors.

* Processors have specialized hardware for fetching, decoding, and dispatching their own native instruction set. This hardware is typically not available to the programmer (although it may be available at the microprogram level), and therefore not useful to the v-machine interpreter in its time-consuming operation of instruction fetching, decoding, and dispatching.

* The v-machine architecture may be substantially different from that of the underlying hardware. For example, many v-machines, including both the P-system and Smalltalk-80 v-machines, use a stack-oriented architecture for convenience in code generation, but most available hardware machines execute register-oriented code much more efficiently than stack-oriented code.

* The basic operations of the v-machine may be relatively expensive to implement, even though the overall algorithm represented by a v-code program may not be much more expensive than if it were implemented in the hardware instruction set. For example, even though a naive interpreter for the Smalltalk-80 v-code must perform reference counting operations every time it pushes a variable value onto the stack, a sequence of several instructions often has no net effect on reference counts.

If the v-code were translated to n-code after normal compilation of a source program to v-code, the interpreter's overhead could be eliminated and some optimizations become possible. One technique for eliminating part of the overhead of interpretation is *threaded code* [Bell 73] [Moore 74]. In this approach, v-code consists of an actual sequence of subroutine calls on runtime routines. This technique does reduce the overhead for fetching and dispatching v-code instructions, although it does not help with operand decoding, or enable optimizations that span more than one v-instruction. We prefer to translate v-code to in-line n-code in a more sophisticated way.

Naive translation from v-code to n-code is a process something like macro-expansion. In fact, [Mitchell 71] observed that a translator can be derived very simply from an interpreter by having the interpreter save its action-routine code in a buffer rather than executing it. If the computation performed by individual action routines is small relative to the computation needed for the interpreter loop, the benefit of even this simple kind of translation will be great.

Translation-time can also be considered an opportunity for peephole optimization or even mapping stack references to registers [Pittman 80]. Translation back-ends for portable compilers have been implemented [Zellweger 79].

DYNAMIC TRANSLATION

Because the Smalltalk-80 v-code is a compact representation that captures the basic semantics of the language, n-code will typically take up much more space than v-code. (In the implementation discussed in this paper, n-code takes about 5 times as much space as v-code.) This would place severe stress on a virtual memory system if the n-code were being paged. However, since n-code is derived algorithmically from v-code, there is no need to keep it permanently: it can be recomputed when needed, if this is more efficient than swapping it in from secondary storage. This leads us to the idea of translating at runtime. (The idea of dynamic translation appears in [Rau 78], where it is applied to translation from v-code to microcode.) When a procedure is about to be executed, it must exist in n-code form. If it does not, the call faults and the translator takes control. The translator finds the corresponding v-code routine, translates it, and completes the call. Since, as mentioned earlier, the translation process is more akin to macro-expansion than compilation, translation time for a v-code byte is comparable to the time taken to interpret it.

We consider the translation approach, and dynamic translation in particular, to be the most interesting part of our research, since it motivated the work on multiple state representations described below. A later section of this paper presents the experimental results that support our contention that dynamic translation is an effective technique in a substantial region of current technological parameters.

MAPPING STATE AT RUNTIME

Since the definition of the Smalltalk-80 v-machine makes runtime state such as procedure activations visible to the programmer as data objects, an implementation based on n-code must find a way to make the state appear to the programmer as though it were the state of a v-machine, regardless of the actual representation. The system must maintain a mapping of n-machine state to v-machine state; in particular, it must keep the v-code available for inspection.

How can we guarantee that all attempts to access a quantity requiring representation mapping are detected? The structure of the Smalltalk-80 language guarantees that the only code that can access an object of a given class directly is the code that implements messages sent to that class. Thus, the only code that can directly access the parts of an object requiring mapping is code associated with that object's class. Recall that all the code in the Smalltalk-80 system is written in the Smalltalk-80 language, hence compiled into v-code. When we translate a procedure from v-code to n-code that is associated with a class whose representation may require mapping, we generate special n-code that calls a subroutine to ensure that the object is represented in a form where accesses to its named parts are meaningful.

The most obvious quantity requiring mapping is the return address (PC) in an activation record, which refers to a location in the n-code procedure rather than in the v-code. Although there is no simple algorithmic correspondence between the v-PC and the n-PC values, the v-PC need only be available when a program attempts to inspect an activation as a data object. At that moment, the system can consult (or compute) a table associated with the procedure that gives the correspondence between n- and v-PC values.

We can greatly reduce the size of the mapping tables for PC values by observing that the PC can only be accessed when an activation is suspended, i.e., at a procedure call or interrupt/process-switch. If we are willing to accept somewhat greater latency in a Smalltalk-80 program's response to interrupts, we can choose a restricted but sufficient set of allowable interrupt points, and only store the mapping tables for those points. This is what our implementation does: interrupts are only allowed at, and PC map entries are only stored for, all procedure calls and backward branches (the latter since interrupts must be allowed inside loops).

MULTIPLE REPRESENTATIONS OF CONTEXTS

As mentioned earlier, the format of procedure activation records are part of the Smalltalk-80 v-machine specification. Contexts are full-fledged data objects; they have identifiable fields which can be accessed and they respond to messages. A context is created for every message-send. There is also syntax in the language for creating contexts whose activation is deferred, called *block contexts* in Smalltalk-80 terminology, which correspond to the *functionals*, *closures*, or *funargs* of other languages. Most control structures in the Smalltalk-80 system are implemented with block contexts.

The fact that contexts are standard data objects implies that they must be created like data objects, i.e., allocated on a heap and reclaimed by garbage collection or reference counting. Unfortunately, conventional machines are adapted for calling sequences that create a new activation record as a stack frame, storing suspended state in predefined slots in the frame. Actually implementing contexts as heap objects results in a serious performance penalty.

Measurements show that even in Smalltalk-80 programs, more than 85% of all contexts behave like procedure activations in conventional languages: they are created by a call, never referenced as a data object, and can be freed as soon as control returns from them. (Note that any context in which a block context is created does not satisfy this criterion.) Such contexts are candidates for stack-frame representation. (An unpublished experimental implementation of an earlier Smalltalk system used linear stacks, but did not deal properly with contexts that outlived their callers.)

Stack allocation of contexts solves one of the two major efficiency problems associated with treating contexts like other objects, namely the overhead of allocating the contexts themselves. [Deutsch&Bobrow 76] shows how to solve the other problem, of reference counting operations apparently being required on every store into a local variable. With these two problems solved, we can use the hardware subroutine call, return, and store instructions directly.

Our system has several types of context representations. A message-send creates a new context in a representation optimized for execution: a frame is allocated on the machine's stack (with some spare slots) by the usual machine instructions. In the simple case, where no reference is ever made to the context as a data object, the machine's return instruction simply pops the frame off the stack when control returns from the context. This kind of context, which lives its life as a stack frame, we call *volatile.*

At the other extreme, we store contexts in a format compliant with the virtual machine specification, which can be manipulated as data items. We call this representation *stable.*

The third representation of a context, called *hybrid,* is a stack frame that incorporates header information to make it look partly like an ordinary data object. A volatile context is converted to hybrid when a pointer is generated to it. Since this makes it possible for programs to refer to the context as an object, we fill in slots in the frame corresponding to the header fields in an ordinary object. This pseudo-object is tagged as being of a class we name "DummyContext." A block of memory is allocated, and its address is stored in the context in case the context must be stabilized in the future. Since there may be pointers to this context, it cannot be returned from in a normal way, so the return address is copied to another slot in the frame and replaced with the address of a clean-up routine that stabilizes the context on return.

When a message is sent to a hybrid context, the send fails (there are no procedures defined for the DummyContext class), and a routine is called to convert the hybrid context to the stabilized form. At this point PC mapping comes into play; the n-PC in the activation is converted to a v-PC for the stabilized representation. Pointers to the hybrid context are switched to refer to the stable context (this is simple in our system, which uses an indirection table for all objects). After the context has been stabilized, the failed message is re-sent to the stable form.

A stable context is not suitable for execution. Before a stabilized context can be resumed, it is reconstituted on the stack as hybrid. Again, this means that the n-PC must be reconstructed from the v-PC. Usually the v-PC does not change during the stable period, so our system includes a one-element cache in each n-code procedure for the most recent v-PC/n-PC pair, to avoid having to run the mapping algorithm.

Block contexts are "born" in stable form, since the whole purpose of closures is to provide a representation for an execution context which can be invoked later.

IN-LINE CACHING OF METHOD ADDRESSES

Message-passing is applied down to the simplest operations in Smalltalk. The system provides a variety of predefined classes: the most basic operations on elementary data types (such as addition of integers) are performed by *primitives* implemented

by the kernel of the system, rather than by Smalltalk routines, but there is no distinction drawn at the language level. Since message-sends are so ubiquitous, they must be fast; the operation of method-lookup is both expensive and critical.

All existing Smalltalk-80 implementations accelerate method-lookup by using a *method cache*, a hash table of popular method addresses indexed by the pair (receiver class, message selector). This simple technique typically improves system performance by 20-30%. More extensive measurements of this improvement appear in [Krasner 83].

Further performance improvements are suggested by the observation of dynamic locality of type usage. That is, at a given point in code, the receiver is often the same class as the receiver at the same point when the code was last executed. If we cache the looked-up method address at the point of send, subsequent execution of the send code has the method address at hand, and method-lookup can be avoided if the class of the receiver is the same as it was at the previous execution of this particular send. Of course, the class of the receiver may have changed, and must be checked against the class corresponding to the cached method address.

In the implementation described here, the translator generates n-code for sends *unlinked* -- as a call to the method-lookup routine, with the selector as an in-line argument. The method-lookup routine *links* the call by finding the receiver class, storing it in-line at the call point, and doing the method-lookup (like other implementations, it uses a selector/class-method cache). When the n-code method address is found, it is placed in-line with a call instruction, overwriting the former call to the lookup routine. The call is then re-executed. (Of course, there may be no corresponding n-code method, in which case the translator is called first.) Note that this is a kind of dynamic code modification, which is generally condemned in modern practice. The n-method address can just as well be placed out-of-line and accessed indirectly; code modification is more efficient, and we are using it in a well-confined way.

The entry code of an n-code method checks the stored receiver class from the point of call against the actual receiver class. If they do not match, relinking must occur, just as if the call had not yet been linked.

Since linked sends have n-code method addresses bound in-line, this address must be invalidated if the called n-code method is being discarded from memory. The idea of scanning all n-code routines to invalidated linked addresses was initially so daunting that we almost rejected the scheme. However, since n-code only exists in main memory, invalidation cannot produce time-consuming page faults. Furthermore, since the PC mapping tables described earlier contain precisely the addresses of calls in the n-code, no searching of the n-code is required: it is only necessary to go through the mapping tables and overwrite the call instructions to which the entries point. (A scheme similar to this may be found in [Moon 73].)

For a few special selectors like +, the translator generates in-line code for the common case along with the standard send code. For example, + generates a class check to verify that both arguments are small integers, native code for integer addition, and an overflow check on the result. If any of the checks fail, the send code is executed. This is a space-time tradeoff justified by measurements that indicate that the overwhelming majority of arithmetic operations involve only small integers, even though they are (in principle) polymorphic like all other operations in the language.

EXPERIMENTAL RESULTS

Three aspects of our results deserve experimental validation: the use of stable and volatile context representations, the use of the one-element in-line cache and linked sends for accelerating method-lookup, and the technique of v-code to n-code translation (specifically, dynamic translation).

CONTEXT REPRESENTATIONS

The dramatic drop in reference counting overhead obtained by treating contexts specially has been documented elsewhere (e.g., [Krasner 83], section 19). We also obtain a striking efficiency improvement by allocating contexts on a stack, and by keeping their contents in execution-oriented form. Offsetting these advantages, in our implementation there is an added overhead of converting contexts between volatile/hybrid and stable forms, and of ensuring that a context accessed as a data object (either by sending it a message or directly while running a method implemented in a context class) is in stable form.

To evaluate the performance advantage of linear context allocation and volatile representation, we compared our code for allocating and deallocating contexts against code based on a hypothetical design that used the standard object representation for contexts, but did not reference-count their contents. This code appears to take about 8 times as long to execute, which would make it consume 12% of total execution time compared to 1.5% for our present code.

Less than 10% of all contexts ever exist in other than volatile form. Block contexts, which are created in stable form, and their enclosing context, which must be made hybrid so the block context can refer to it, account for two-thirds of these; nearly all of the remainder arise from an implementation detail regarding linking together fixed-size stack segments. In all of our measured examples, the time required for the conversion between the stable and volatile form was under 3% of total execution time.

If the receiver of a message is not a hybrid context, there is no overhead for making the check because it happens as part of the normal method-lookup (recall that hybrid contexts appear to be objects of a special class DummyContext with no associated methods). Only when method-lookup fails is a check made whether the receiver was actually a DummyContext. In the normal operation of the system, messages are only sent to contexts by the debugger and for cleanup during destruction of a process, so the overall impact is negligible.

As discussed above, methods associated with context classes must be translated specially, so that each reference to an instance variable checks to make sure the receiver is in stable form. The time required for this check is negligible.

IN-LINE CACHE AND LINKED SENDS

Independent measurements by us and by a group at U.C. Berkeley confirm that the one-element in-line cache is effective about 95% of the time. Measurements reported in [Krasner 83] indicate that a more conventional global cache of a reasonable size is effective about 85-90% of the time. It may be that an in-line cache tends to lower the effectiveness of the global cache, since most of the lookups that would succeed in the global cache are now handled by the in-line cache, but we have no direct evidence on this point.

Adding an in-line cache to the simple translator described below improved overall performance by only 9%. On a benchmark consisting almost entirely of message sends where the in-line cache is guaranteed valid, the in-line cache only improved performance by 11%. The improvement obtained by adding an in-line cache to the optimizing translator was also about 10%. Our original hand-analysis indicated that the overall improvement should be closer to 20%, and we cannot yet account for the discrepancy. The code produced by the optimizing

translator for the activate-and-return benchmark is a remarkable 47% faster than the code from the simple translator with the in-line cache, suggesting that operations other than the overhead eliminated by the in-line cache still dominates overall execution time.

Dynamic Code Translation

Our implementation of the Smalltalk-80 v-machine is designed to be easily switchable between different execution strategies. We have implemented a straightforward interpreter, a simple translator with almost no optimization, and a more sophisticated translator. Both translators exist in two variants, with and without the in-line cache described above. Switching between strategies simply requires relinking the implementation with a different set of modules; the price in execution speed paid for this flexibility is negligible.

Our first experiment in code translation was a simple translator that does little peephole optimization and always generates exactly 4 n-bytes per v-byte. (The latter restriction eliminated the need for the PC mapping tables described earlier.)

Our second experiment was a translator that does significant peephole optimization. The code it generates keeps the top element of the v-machine stack in a machine register whenever possible, and implements all v-instructions in-line except sends and a few rare instructions like load current context. Even arithmetic and relational operations are implemented in-line, with a call on an out-of-line routine if the operands are not small integers. The resulting code is bulky but fast.

To estimate the space required by translated methods, we have observed that the average v-method consists of 55% pointers (literal constants, message selectors, and references to global variables) and 45% v-instructions. Since our simple translator expands each v-code byte to 4 n-code bytes, the expansion factor for the method as a whole is .55+(.45*4)=2.35. The version of the simple translator that uses an in-line cache simply triples the size of the pointer area, leaving room for a cached class and n-method pointer regardless of whether the pointer is a selector or something else. This expands the total size of methods by a factor of (3*.55)+(4*.45)=3.45. The observed expansion factors for the optimizing translators appear in the table below.

We ran the standard set of Smalltalk-80 benchmarks described in [Krasner 83], section 9, using each of our five execution strategies. The normalized results are summarized in the following table:

Strategy	Space	Time
Interpreter	1.00	1.000
Simple translator, no in-line cache	2.35	0.686
Simple translator with in-line cache	3.45	0.625
Optimizing translator, no in-line cache	5.0	0.564
Optimizing translator with in-line cache	5.03	0.515

The space figure for the optimizing translator without the in-line cache could be reduced at the expense of further slowing the code down.

With respect to paging behavior in a virtual memory environment, we would like to compare the following three execution strategies:

* Pure interpretation: only v-code exists; it is brought into main memory as needed.

* Static translation: n-code is generated simultaneously with v-code. Only n-code is needed at execution time. N-code is brought into memory as needed.

* Dynamic translation: n-code is kept in a cache in main memory; v-code is brought into memory for translation as needed.

Note that space taken by n-code in main memory trades off against space for data. When main memory space is needed (either for n-code or for data), we have the option of replacing data pages or discarding n-code. Unfortunately, since the work described here has been carried out in a non-virtual memory environment, we have no experimental results on this topic.

CONCLUSIONS AND RELATED WORK

Perhaps the most important observation from our research is that we have demonstrated that it is possible to implement an interactive system based on a demanding high-level language, with only a modest increase in memory requirements and without the use of any of the special hardware (special-purpose microcode, tagged memory architecture, garbage collection co-processor) often advocated for such systems, and with resulting performance that users judge excellent. We have achieved this by careful optimization of the observed common cases and by the plentiful use of caches and other changes of representation.

A related research project [Patterson 83] is investigating a Smalltalk-80 implementation that uses only n-code, on a specially designed VLSI processor called SOAR. As discussed above, this implementation requires rewriting the compiler, debugger, and other tools that manipulate compiled code and contexts. We expect some interesting comparisons between the two approaches sometime in 1984, when the SOAR implementation becomes operational.

We believe the techniques described in this paper are applicable in varying degrees to other late-bound languages such as Lisp, and to portable V-code-based language implementations such as the Pascal P-system, but we have no current plans to investigate these other languages.

ACKNOWLEDGMENTS

Thanks are due to Mike Braca, who programmed the I/O kernel of our implementation; Bob Hagmann, who programmed the optimizing code translator and made many contributions to the design of the system; and Mark Roberts, who implemented the disk file system and virtual memory capabilities. Bob Hagmann, Dan Ingalls, and Paul McCullough contributed helpful comments on this paper. The Smalltalk-80 system itself is owed to PARC SCG. Butler Lampson gave helpful suggestions during the early project design phase.

REFERENCES

[Ammann 75] Ammann, U., Nori, Jensen, K., Nageli, H., "The Pascal (P) Compiler Implementation Notes." Institut Fur Informatik, Eidgenossische Technische Hochschule, Zurich, 1975.

[Ammann 77] Ammann, U., "On code generation in a Pascal compiler." Software Practice and Experience v7 #3, June/July 1977, pp. 391-423.

[Bell 73] Bell, J. R., "Threaded Code." Communications of the ACM, v16 (1973) pp. 370-372.

[Deutsch & Bobrow 76] Deutsch, L. P., Bobrow, D. G., "An efficient, incremental, real-time garbage collector." Communications of the ACM, October 1976.

[Goldberg 83] Goldberg, A., Robson, D., "Smalltalk-80: The Language and its Implementation." Addison-Wesley, Reading, MA, 1983.

[Goldberg 84] Goldberg, A., "Smalltalk-80: The Interactive Programming Environment." Addison-Wesley, Reading, MA, 1984.

[Krasner 83] Krasner, Glenn, Ed., "Smalltalk-80: Bits of History, Words of Advice." Addison-Wesley, Reading, MA, 1983.

[Lampson 81] Lampson, B. W., Ed., "The Dorado: A High-Performance Personal Computer." Xerox PARC Report CSL-81-1, Palo Alto, CA, January 1981.

[Mitchell 71] Mitchell, J. G., "The Design and Construction of Flexible and Efficient Interactive Programming Systems." Ph.D. dissertation, 1971, NTIS AD 712-721, in Outstanding Dissertations in the Computer Sciences, Garland Publishing, New York (1978).

[Moon 73] Moon D., Ed., Maclisp Manual pp. 3-75 to 3-77, MIT AI Laboratory Technical Report (1973).

[Moore 74] Moore, C. H., "FORTH: a New Way to Program a Computer." Astronomy and Astrophysics Supplement, # 15 (1974) pp 497-511.

[Patterson 83] Patterson, D., Ed., "Smalltalk on a RISC: Architectural Investigations (Proceedings of CS 292R)." University of California, Berkeley, April 1983.

[Perkins 79] Perkins, D. R., Sites, R. L., "Machine independent Pascal code optimization." ACM SIGPLAN Notices v14 #8 (August 1979) pp. 201-207.

[Pittman 80] Pittman, T.J., "A Practical Optimizer: Zero-Address to Multi-Address Code." M.S. thesis, University of California, Santa Cruz, June 1980.

[Rau 78] Rau, B. R., "Levels of Representation of Programs and the Architecture of Universal Host Machines." Proceedings of Micro-11, Asilomar, CA, November 1978.

[Richards 75] Richards, M., "The portability of the BCPL compiler." Software, Practice and Experience v1 (1971) pp. 135-146.

[SCG 81] Software Concepts Group, special issue on Smalltalk. BYTE Magazine, volume 6, number 8, August 1981.

[XSIS 83] Masinter, L. M., Ed., "Interlisp Reference Manual." Xerox Special Information Systems, Pasadena, CA, 1983.

[Zellweger 79] Zellweger, P. T., "Machine-Independent Optimization in SOPAIPILLA." The S-1 Project 1979 Annual Report (Chapter 8), Lawrence Livermore Laboratory (1979).

Enhancing Server Availability and Security Through Failure-Oblivious Computing

Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy,
Tudor Leu, and William S. Beebee, Jr.
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

We present a new technique, *failure-oblivious computing*, that enables servers to execute through memory errors without memory corruption. Our safe compiler for C inserts checks that dynamically detect invalid memory accesses. Instead of terminating or throwing an exception, the generated code simply discards invalid writes and manufactures values to return for invalid reads, enabling the server to continue its normal execution path.

We have applied failure-oblivious computing to a set of widely-used servers from the Linux-based open-source computing environment. Our results show that our techniques 1) make these servers invulnerable to known security attacks that exploit memory errors, and 2) enable the servers to continue to operate successfully to service legitimate requests and satisfy the needs of their users even after attacks trigger their memory errors.

We observed several reasons for this successful continued execution. When the memory errors occur in irrelevant computations, failure-oblivious computing enables the server to execute through the memory errors to continue on to execute the relevant computation. Even when the memory errors occur in relevant computations, failure-oblivious computing converts requests that trigger unanticipated and dangerous execution paths into anticipated invalid inputs, which the error-handling logic in the server rejects. Because servers tend to have small error propagation distances (localized errors in the computation for one request tend to have little or no effect on the computations for subsequent requests), redirecting reads that would otherwise cause addressing errors and discarding writes that would otherwise corrupt critical data structures (such as the call stack) localizes the effect of the memory errors, prevents addressing exceptions from terminating the computation, and enables the server to continue on to successfully process subsequent requests. The overall result is a substantial extension of the range of requests that the server can successfully process.

1 Introduction

Memory errors such as out of bounds array accesses and invalid pointer accesses are a common source of program failures. Safe languages such as ML and Java use dynamic checks to eliminate such errors — if, for example, the program attempts to access an out of bounds array element, the implementation intercepts the attempt and throws an exception. The rationale is that an invalid memory access indicates an unanticipated programming error and it is unsafe to continue the execution without first taking some action to recover from the error.

Recently, several research groups have developed compilers that augment programs written in unsafe languages such as C with dynamic checks that intercept out of bounds array accesses and accesses via invalid pointers (we call such a compiler a *safe-C* compiler) [17, 58, 45, 36, 50, 37]. These checks use additional information about the layout of the address space to distinguish illegal accesses from legal accesses. If the program fails a check, it terminates after printing an error message.

1.1 Failure-Oblivious Computing

Note that it is possible for the compiler to automatically transform the program so that, instead of throwing an exception or terminating, it simply ignores any memory errors and continues to execute normally. Specifically, if the program attempts to read an out of bounds array element or use an invalid pointer to read a memory location, the implementation can simply (via any number of mechanisms) manufacture a value to supply to the program as the result of the read, and the program can continue to execute with that value. Similarly, if the program attempts to write a value to an out of bounds array element or use an invalid pointer to write a memory location, the implementation can simply discard the value and continue. We call a computation that uses this strategy a *failure-oblivious* computation, since it is oblivious to its failure to correctly access memory.

It is not immediately clear what will happen when a program uses this strategy to execute through a memory error. When we started this project, our hypothesis was that, for at least some programs, this continued execution would produce acceptable results. To test this hypothesis, we implemented a C compiler that generates failure-oblivious code, obtained some C programs with known memory errors, and observed the execution of failure-oblivious versions of these programs. Here is a summary of our observations:

- **Acceptable Continued Execution:** We targeted memory errors in servers that correspond to security vulnerabilities as documented at vulnerability tracking web sites [13, 12]. For all of our tested servers, failure-oblivious computing 1) eliminates the security vulnerability and 2) enables the server to successfully execute through the error to continue to serve the needs of its users.

- **Acceptable Performance:** Failure-oblivious computing entails the insertion of dynamic bounds checks into the compiled program. Previous experiments with safe-C compilers have indicated that these checks usually cause the program to run less than a factor of two slower than the version without checks, but that in some cases the program may run as much as eight to twelve times slower [58, 50]. Our results are consistent with these previous results. Note that many of our servers implement interactive computations for which the appropriate performance measure is the observed pause times for processing interactive requests. For all of our interactive servers, the application of failure-oblivious computing does not perceptibly increase the pause times.

Our conclusion is that continued execution through memory errors produces completely acceptable results for all of our servers *as long as failure-oblivious computing prevents these errors from corrupting the server's address space or data structures*.

1.2 Reason for Successful Execution

Memory errors can damage a computation in several ways: 1) they can cause the computation to terminate with an addressing exception, 2) they can cause the computation to become stuck in an infinite loop, 3) they can change the flow of control to cause the computation to generate a new and unacceptable interaction sequence (either with the user or with I/O devices), 4) they can corrupt data structures that must be consistent for the remainder of the computation to execute acceptably, or 5) they can cause the computation to produce unacceptable results.

Because failure-oblivious computing intercepts all invalid memory accesses, it eliminates the possibility that the computation may terminate with an addressing exception. It is still possible for the computation to infinite loop, but we have found a sequence of return values for invalid reads that, in practice, appears to eliminate this problem for our server programs. Our servers have simple interaction sequences — read a request, process the request without further interaction, then return the response. As long as the computation that processes the request terminates, control will appropriately flow back to the code that reads the next request and there will be no unacceptable interaction sequences. Discarding invalid writes tends to localize any memory corruption effects. In particular, it prevents an access to one data unit (such as a buffer, array, or allocated memory block) from corrupting another data unit. In practice, this localization protects many critical data structures (such as widely used application data structures or the call stack) that must remain consistent for the program to execute acceptably.

The remaining issue is the potential production of unacceptable results. Manufacturing values for reads clearly has the potential to cause a subcomputation to produce an incorrect or unexpected result. The key question is how (or even if) the incorrect or unexpected result may propagate through the remaining computation to affect the overall results of the program.

All of our initially targeted memory errors eventually boil down to buffer-overrun problems: as it processes a request, the server allocates a fixed-size buffer, then (under certain circumstances) fails to check that the data actually fits into this buffer. An attacker can exploit this error by submitting a request that causes the server to write beyond the bounds of the buffer to overwrite the contents of the stack or heap, typically with injected code that the server then executes. Such attacks are currently the most common source of exploited security vulnerabilities in modern networked computer systems [2]. Estimates place the total cost of such attacks in the billions of dollars annually [3].

Failure-oblivious computing makes a server invulnerable to this kind of attack — the server simply discards the out of bounds writes, preserving the consistency of the call stack and other critical data structures. For two of our servers the memory errors occur in computations and buffers that are irrelevant to the overall results that the server produces for that request. Because failure oblivious computing eliminates any addressing exceptions that would otherwise terminate the computation, the server executes through the irrelevant computation and proceeds on to process the request (and subsequent requests) successfully. For the other servers (in these servers the memory errors occur in relevant computa-

tions and buffers) , failure-oblivious computing converts the attack request (which would otherwise trigger a dangerous, unanticipated execution path) into an anticipated invalid input which the server's standard error-handling logic rejects. The server then proceeds on to read and process subsequent requests acceptably.

One of the reasons that failure-oblivious computing works well for our servers is that they have short error propagation distances — an error in the computation for one request tends to have little or no effect on the computation for subsequent requests. By discarding invalid writes, failure-oblivious computing isolates the effect of any memory errors to data local to the computation for the request that triggered the errors. The result is that the server has short data error propagation distances — the errors do not propagate to data structures required to process subsequent requests. The servers also have short control flow error propagation distances: by preventing addressing exceptions from terminating the computation, failure-oblivious computing enables the server to return to a control flow path that leads it back to read and process the next request. Together, these short data and control flow propagation distances ensure that any effects of the memory error quickly work their way out of the computation, leaving the server ready to successfully process subsequent requests.

1.3 Scope

Our expectation is that failure-oblivious computing will work best with computations, such as servers, that have short error propagation distances. Failure-oblivious computing enables these programs to survive otherwise fatal errors or attacks and to continue on to execute and interact acceptably. Failure-oblivious computing should also be appropriate for multipurpose systems with many components — it can prevent an error in one component from corrupting data in other components and keep the system as a whole operating so that other components can continue to successfully fulfill their purpose in the computation.

Until we develop technology that allows us to track results derived from computations with memory errors, we anticipate that failure-oblivious computing will be less appropriate for programs (such as many numerical computing programs) in which a single error can propagate through to affect much of the computation. We also anticipate that it will be less appropriate for programs in which it is acceptable and convenient to terminate the computation and await external intervention. This situation occurs, for example, during development — the program is typically not producing any useful results and developers with the ability and motivation to find and eliminate any errors are readily available. We therefore see failure-oblivious computing as useful primarily for deployed programs whose users 1) need the results that the program produces and 2) are unable or unwilling to tolerate failures or to find and fix errors in the program.

1.4 Advantages and Drawbacks

The primary characteristic of failure-oblivious computing as compared with previous approaches is continued execution combined with the elimination of data structure corruption caused by memory errors. The potential benefits include:

- **Availability:** The combination of protection against data structure corruption and continued execution in the face of memory errors can significantly increase the availability of the server. This combination enables the server to continue to provide service to legitimate users even in the face of repeated attacks (or, for that matter, other infrequently-triggered fatal memory errors).

- **Security:** Failure-oblivious computing eliminates the possibility that an attacker can exploit memory errors to corrupt the address space of the server. The result is a more secure system that is immune to buffer-overrun attacks.

- **Minimal Adoption Cost:** The net adoption cost to the developer is to recompile the server using a compiler that generates failure-oblivious code. There is no need to change programming languages, write exception handling code, or modify the software in any way. Failure-oblivious computing can therefore be applied immediately to today's software infrastructure.

- **Reduced Administration Overhead:** One of the most challenging system administration tasks is ensuring that servers are kept up to date with a constant stream of (potentially disruptive) patches and upgrades; this stream is driven, in large part, by the need to eliminate memory-error based security vulnerabilities in otherwise perfectly acceptable servers. Because failure-oblivious computing eliminates this class of errors, it may enable system administrators to safely ignore patches whose purpose is to eliminate security vulnerabilities caused by memory errors. Ideally, administrators would become able to patch their systems primarily to obtain new functionality, not because they need to close security vulnerabilities in programs that are otherwise fully serving the needs of their users.

There are also several potential drawbacks:

- **Unanticipated Execution Paths:** Failure-oblivious computing has the potential to take the program down an execution path that was unanticipated by the programmer, with the prospect of this path producing unacceptable results.[1] This possibility can be especially problematic if errors in the unanticipated path have long propagation distances through the relevant data or when control fails to flow back to an appropriate point in the program. This drawback is, in our view, an unavoidable consequence of *any* mechanism that is intended to increase the resilience of programs in the face of errors — errors occur precisely because the program encountered a situation that the programmer either did not anticipate or did not deem worth handling correctly.

- **The Bystander Effect:** A more abstract issue is the potential for failure-oblivious computing to trigger the *bystander effect* in developers. In a variety of settings that range from manufacturing [25] to personal relationships [40, 24], the mere presence of mechanisms that may detect and compensate for errors has the effect of reducing the effectiveness of the participants in the setting and, in the end, the overall quality of the system as a whole. A potential explanation is that the participants start to rely psychologically on the error recovery mechanisms, which reduces their motivation to eliminate errors in their own work. Deploying failure-oblivious computing into a software development setting may therefore reduce the quality of the software that the developers are able to deliver. One obvious way to combat the bystander effect in this setting is to ban the use of failure-oblivious computing during development. Once again, note that the possibility of triggering the bystander effect is not restricted to failure-oblivious computing — *any* error recovery mechanism has the potential to trigger this effect.

1.5 Contributions

This paper makes the following contributions:

- **Failure-Oblivious Computing:** It introduces the concept of failure-oblivious computing, in which the program discards illegal writes, manufactures values for illegal reads, and continues to execute through memory errors without address space or data structure corruption.

- **Experience:** It presents our experience using failure-oblivious computing to enhance the security and availability of a range of widely used open-source servers. Our results show that:

  - **Standard Compilation:** With the standard unsafe C compiler, the servers are vulnerable to memory errors and attacks that exploit these memory errors.

  - **Safe Compilation:** With a C compiler that generates code that exits with an error message when it detects a memory error, the servers exit when presented with an input that triggers a memory error (denying the user access to the services that the server is intended to provide).

  - **Failure-Oblivious Compilation:** With our C compiler that generates failure-oblivious code, all of our servers execute successfully through memory errors and attacks to continue to satisfy the needs of their users. Failure-oblivious computing improves both the availability and the security of the servers in our test suite.

- **Explanation:** By relating the properties of servers to the properties of failure-oblivious computing, we explain why failure-oblivious computing may work well for this general class of programs.

2 Example

We next present a simple example that illustrates how failure-oblivious computing operates. Figure 1 presents a (somewhat simplified) version of a procedure from the Mutt mail client discussed in Section 4.6. This procedure takes as input a string encoded in the UTF-8 format and returns as output the same string encoded in modified UTF-7 format. This conversion may increase the size of the string; the problem is that the procedure fails to allocate sufficient space in the return string for the worst-case size increase. Specifically, the procedure assumes a worst-case increase ratio of 2; the actual worst-case ratio is 7/3. When passed (the very rare) inputs with large increase ratios, the procedure attempts to write beyond the end of its output array.

With standard compilers, these writes succeed, corrupt the address space, and the program terminates with a segmentation violation. With safe-C compilers, Mutt exits with a memory error and does not even start the user interface. With our compiler, which generates failure-oblivious code, the program discards all writes beyond the end of the array and the procedure returns with an incompletely translated (truncated) version of the string. Mutt then uses the return value to tell the mail server

[1] We note in passing that this potential is already present in every program — the mere absence of memory errors provides no guarantee that the program is, in fact, operating acceptably.

```
static char *
utf8_to_utf7 (const char *u8, size_t u8len) {
    char *buf, *p;
    int ch, int n, i, b = 0, k = 0, base64 = 0;

    /* The following line allocates the return
       string.  The allocated string is too small;
       instead of u8len*2+1, a safe length would
       be u8len*4+1.
    */
    p = buf = safe_malloc (u8len * 2 + 1);

    while (u8len) {
        unsigned char c = *u8;
        if (c < 0x80) ch = c, n = 0;
        else if (c < 0xc2) goto bail;
        else if (c < 0xe0) ch = c & 0x1f, n = 1;
        else if (c < 0xf0) ch = c & 0x0f, n = 2;
        else if (c < 0xf8) ch = c & 0x07, n = 3;
        else if (c < 0xfc) ch = c & 0x03, n = 4;
        else if (c < 0xfe) ch = c & 0x01, n = 5;
        else goto bail;

        u8++, u8len--;
        if (n > u8len) goto bail;
        for (i = 0; i < n; i++) {
            if ((u8[i] & 0xc0) != 0x80) goto bail;
            ch = (ch << 6) | (u8[i] & 0x3f);
        }
        if (n>1 && !(ch >> (n*5+1))) goto bail;
        u8 += n, u8len -= n;

        if (ch < 0x20 || ch >= 0x7f) {
            if (!base64) {
                *p++ = '&';
                base64 = 1;
                b = 0;
                k = 10;
            }
            if (ch & ~0xffff) ch = 0xfffe;
            *p++ = B64Chars[b | ch >> k];
            k -= 6;
            for (; k >= 0; k -= 6)
                *p++ = B64Chars[(ch >> k) & 0x3f];
            b = (ch << (-k)) & 0x3f;
            k += 16;
        } else {
            if (base64) {
                if (k > 10) *p++ = B64Chars[b];
                *p++ = '-';
                base64 = 0;
            }
            *p++ = ch;
            if (ch == '&') *p++ = '-';
        }
    }

    if (base64) {
        if (k > 10) *p++ = B64Chars[b];
        *p++ = '-';
    }

    *p++ = '\0';
    safe_realloc ((void **) &buf, p - buf);
    return buf;

bail:
    safe_free ((void **) &buf);
    return 0;
}
```

Figure 1: String Encoding Conversion Procedure

which mail folder it wants to open. The mail server responds with an error code indicating that the folder does not exist. Mutt correctly handles this error and continues to execute, enabling the user to process email from other, legitimate, folders.

This example illustrates two key aspects of applying failure-oblivious computing:

- **Subtle Errors:** Real-world programs can contain subtle memory errors that can be very difficult to detect by either testing or code inspection, and these errors can have significant negative consequences for the program and its users.

- **Mostly Correct Programs:** Testing usually ensures that the program is mostly correct and works well except for exceptional operating conditions or inputs. Failure-oblivious computing can therefore be seen as a way to enable the program to proceed past such exceptional situations to return back within its normal operating envelope. And as this example illustrates, failure-oblivious computing can actually facilitate this return by converting unanticipated memory corruption errors into anticipated error cases that the program handles correctly.

3 Implementation

A failure-oblivious compiler generates two kinds of additional code: checking code and continuation code. The checking code detects memory errors and can be the same as in any memory-safe implementation. The continuation code executes when the checking code detects an attempt to perform an illegal access. This code is relatively simple: it discards erroneous writes and manufactures a sequence of values for erroneous reads.

Our implementation uses a checking scheme originally developed by Jones and Kelly [37] and then significantly enhanced by Ruwase and Lam [50]. This checking scheme maintains a table that maps locations to data units (each struct, array, and variable is a data unit) and uses this table to distinguish in bounds and out of bounds pointers.

Our implementation of the write continuation code simply discards the value. Our implementation of the read continuation code redirects the read to a preallocated buffer of values. In principle, any sequence of manufactured values should work. In practice, these values are sometimes used to determine loop conditions. Midnight Commander (see Section 4.5), for example, contains a loop that, for some inputs, searches past the end of a buffer looking for the "/" character. If the sequence of generated values does not include this character, the loop never terminates and Midnight Commander hangs. We therefore generate a sequence that iterates through

all small integers, increasing the chance that, if the values are used to determine loop conditions, the computation will hit upon a value that will exit the loop (and avoid nontermination). Because zero and one are usually the most commonly loaded values in computer programs [59], the sequence is designed to return these values more frequently than other, less common, values.

One potential concern is that failure-oblivious computing may hide errors that would otherwise be detected and eliminated. To help make the errors more apparent, our compiler can optionally augment the generated code to produce a log containing information about the program's attempts to commit memory errors. This log may help administrators to detect and respond appropriately to the presence such errors. Note, however, that hiding errors is one of the primary goals of this research, and that any technique that makes programs more resilient in the face of errors will reduce the negative impact of the errors and therefore the incentive to find and eliminate them.

4 Experience

We implemented a compiler that generates failure-oblivious code, obtained several widely-used open-source servers with known memory errors, and evaluated the impact of failure-oblivious computing on their behavior. Many of these servers are key components of the Linux-based open-source interactive computing environment.

4.1 Methodology

We evaluate the behavior of three different versions of each server: the *Standard* version compiled with a standard C compiler (this version is vulnerable to any memory errors that the server may contain), the *Bounds Check* version compiled with the CRED safe-C compiler [50] (this version terminates the server with an error message at the first memory error), and the *Failure Oblivious* version compiled with our compiler. We evaluate three aspects of each server's behavior:

- **Security and Resilience:** We chose a workload that contains an input that triggers a known memory error in the server; this input typically exploits a security vulnerability as documented by vulnerability-tracking organizations such as Security Focus [13] and SecuriTeam [12]. We observe the behavior of the different versions on this workload; for the Failure Oblivious version we focus on the acceptability of the continued execution after the error.

- **Performance:** We chose a workload that both the Standard and Failure Oblivious versions can execute successfully. We use this workload to measure the *request processing time*, or the time required for each version to process representative requests. We obtain this time by instrumenting the server to record the time when it starts processing the request and the time when it stops processing the request, then subtracting the start time from the stop time.

- **Stability:** When possible, we deploy the Failure Oblivious version of each server into daily use as part of our normal computational environment. During this deployment we ensure that the workload contains attacks that trigger memory errors in each server. We focus on the long-term acceptability of the continued execution of the Failure Oblivious version of the deployed server.

We note that two of our servers (Pine and Midnight Commander) use out of bounds pointers in pointer inequality comparisons. While this is, strictly speaking, an error, the intention of the programmer is clear. To avoid having these errors cripple the Bounds Check versions of these servers, we (manually) rewrote the code containing the inequality comparisons to eliminate pointer comparisons involving out of bounds pointers.

We ran all the servers on a Dell workstation with two 2.8 GHz Pentium 4 processors, 2 GBytes of RAM, and running Red Hat 8.0 Linux.

4.2 Pine

Pine is a widely used mail user agent (MUA) that is distributed with the Linux operating system [11]. Pine allows users to read mail, fetch mail from an IMAP server, compose and forward mail messages, and perform other email-related tasks. We use Pine 4.44, which is distributed with Red Hat Linux version 8.0. This version of Pine has a memory error associated with a failure to correctly parse certain From fields [10].

4.2.1 The Memory Error

When Pine displays a list of messages, it processes the From field of each message to quote certain characters. This quoting is implemented by transferring the From field into a heap-allocated character buffer for display, inserting a \ character into the buffer before any quoted character. As part of the transfer, the length of the string can increase because of the additional \ characters. The procedure that calculates the maximum possible length of the character buffer fails to correctly account for the potential increase and produces a length that is too short for messages whose From fields contain many quoted characters.

4.2.2 Security and Resilience

The Standard version of Pine writes beyond the end of the buffer, corrupts its heap, and terminates with a segmentation violation. The Bounds Check version detects

the memory error and terminates the computation. With both of these versions, the user is unable to use Pine to read mail because Pine aborts or terminates during initialization as the mail file is loaded and before the user has a chance to interact with the server. The user must manually eliminate the From field from the mail file (using some other mail reader or file editor) before he or she can use Pine to read mail at all.

The Failure Oblivious version discards the out of bounds writes (in effect, truncating the translated From field) and continues to execute through the memory error, enabling the user to process their mail. Because the mail list user interface displays only an initial segment of long From fields, the truncation is not visible to the user. If the user selects the message, a different execution path correctly translates the From field. The displayed message contains the complete From field and the user can read, forward, and otherwise process the message.

4.2.3 Performance

Figure 2 presents the request processing times for the Standard and Failure Oblivious versions of Pine. All times are given in milliseconds. The Read request displays a selected empty message, the Compose request brings up the user interface to compose a message, and the Move request moves an empty message from one folder to another. We performed each request at least twenty times and report the means and standard deviations of the request processing times. All times are given in milliseconds.

Request	Standard	Failure Oblivious	Slowdown
Read	$0.287 \pm 7.1\%$	$1.98 \pm 1.5\%$	6.9
Compose	$0.385 \pm 4.3\%$	$3.11 \pm 2.6\%$	8.1
Move	$1.34 \pm 10.4\%$	$1.80 \pm 11.2\%$	1.34

Figure 2: Request Processing Times for Pine (milliseconds)

As these numbers indicate, the Failure Oblivious version is not substantially slower than the Standard version. Because Pine is an interactive program, its performance is acceptable as long as it feels responsive to its users. Assuming a pause perceptibility threshold of 100 milliseconds for this kind of interactive program [21], it is clear that failure-oblivious computing should not degrade the program's interactive feel. Our subjective experience confirms this expectation: all pause times are imperceptible for all versions.

4.2.4 Stability

During our stability testing period, we used Pine as a default mail reader. Our activities included reading mail, replying to mails, forwarding mails, and managing mail folders. During this time we used Pine to process roughly 25 new mail messages a day (after spam filtering). To test Pine's ability to successfully execute through errors, we periodically sent an email that triggered the memory error discussed above in Section 4.2.1. We also used the failure-oblivious version of Pine to successfully process a large mail folder containing over 100,000 messages. During this usage period, the Failure Oblivious version executed successfully through all errors to perform all requests flawlessly.

4.3 Apache

The Apache HTTP server is the most widely used web server in the world: a recent survey found that 64% of the web sites on the Internet use Apache [9]. Apache version 2.0.47 contains a (under certain circumstances) remotely exploitable memory error [1].

4.3.1 The Memory Error

Apache can be configured to automatically redirect incoming URLs via a set of URL rewrite rules. Each rewrite rule contains a *match pattern* (a regular expression that may match an incoming URL) and a *replacement pattern*. The match pattern may contain parenthesized *captures*, each of which may match a substring from the incoming URL. The replacement pattern may reference these captures. When an incoming URL matches the match pattern, Apache replaces the URL with the replacement pattern after substituting out any referenced captures with the corresponding captured substrings from the incoming URL. As Apache processes the incoming URL, it uses a (stack-allocated) buffer to hold pairs of offsets that identify the captured substrings in the incoming URL. The buffer contains enough room for ten captures. If there are more, Apache writes the corresponding pairs of offsets beyond the end of the buffer.

4.3.2 Security and Resilience

The Standard version performs the out of bounds writes, corrupts its stack, and terminates with a segmentation violation. The Bounds Check version correctly processes legitimate requests without memory errors until it is presented with a URL that triggers the memory error. At this point the child process serving the connection detects the error and terminates. Apache uses a pool of child processes to serve incoming requests. When one of the child processes terminates, the main Apache process creates a new child process to take its place. This mechanism allows both the Standard and Bounds Check versions of Apache to continue to service requests even when repeatedly presented with inputs that cause the child processes to terminate because of memory errors.

The Failure Oblivious version discards the out of bounds writes and continues to execute. It proceeds on to copy the first ten pairs of offsets into another data struc-

ture. Apache uses this data structure to apply the rewrite rule and generate the new URL. Because the rewrite rule uses a single digit to reference each captured substring (these substrings have names $0 through $9), it will never attempt to access any discarded substring offset data. The Failure-Oblivious version of Apache therefore processes each input correctly and continues on to successfully process any subsequent requests. Because the memory errors occur in irrelevant data structures and computations, Failure Oblivious computing eliminates the memory error without affecting the results of the computation at all.

Because Apache isolates request processing inside a pool of regenerating processes, the Bounds Check version successfully processes subsequent requests. The overhead of killing and restarting child processes, however, makes this version vulnerable to an attack that ties up the server by repeatedly presenting it with requests that trigger the error. To investigate this effect, we used several (local) machines to load the server with requests that trigger the error. We then used another client machine to repeatedly fetch the home page of our research project and measured the request throughput at the client. For this workload, the Failure Oblivious version provides a throughput roughly 5.7 times more than the Bounds Check version provides (the insecure Standard version provides a throughput roughly 4.8 times less than the Failure Oblivious version). We attribute the slowdown for the Bounds Check and Standard versions to process management overhead.

4.3.3 Performance

Figure 5 presents the request processing times for the Standard and Failure Oblivious versions of Apache. The Small request serves an 5KByte page (this is the home page for our research project); the large request serves an 830KByte file used only for this experiment. Both requests were local — they came from the same machine on which Apache was running. We performed each request at least twenty times and report the means and standard deviations of the request processing times. All times are given in milliseconds.

Request	Standard	Failure Oblivious	Slowdown
Small	$44.4 \pm 1.3\%$	$47.1 \pm 2.5\%$	1.06
Large	$48.7 \pm 1.8\%$	$50.0 \pm 1.3\%$	1.03

Figure 3: Request Processing Times for Apache (milliseconds)

4.3.4 Stability

For the last nine months we have been using the Failure Oblivious version of Apache to serve our research project's web site at www.flexc.csail.mit.edu; during this time period we measured approximately 400 requests a day from outside our institution. We also generated tens of thousands of requests from another machine, all of which were served correctly. We anticipate that we will continue to use the Failure Oblivious version to serve this web site for the foreseeable future.

During this time period we periodically presented the web server with requests that triggered the vulnerability discussed above. The Failure Oblivious version executed successfully through all of these attacks to continue to successfully service legitimate requests. We observed no anomalous behavior and received no complaints from the users of the web site.

4.4 Sendmail

Sendmail is the standard mail transfer agent for Linux and other Unix systems [15]. It is typically configured to run as a daemon which creates a new process to service each new mail transfer connection. This process executes a simple command language that allows the remote agent to transfer email messages to the Sendmail server, which may deliver the messages to local users or (if necessary) forward some or all of the messages on to other Sendmail servers. Versions of Sendmail earlier than 8.11.7 and 8.12.9 (8.11 and 8.12 are separate development threads) have a memory error vulnerability which is triggered when a remote attacker sends a carefully crafted email message through the Sendmail daemon [14]. We worked with Sendmail version 8.11.6.

4.4.1 The Memory Error

The memory error occurs when Sendmail parses a mail address. A prescan procedure processes the address one character at a time to transfer characters from the address into a fixed-size stack-allocated buffer. This transfer is coded to use a lookahead character and to treat the \ character specially. It is possible for there to be no lookahead character, in which case the integer variable that holds the lookahead character is set to -1. If this variable is set to -1 or contains a \ character that appears in an odd position (first, third, fifth, ...) in a sequence of contiguous \ characters in the address, the prescan skips the block of code that writes the lookahead character into the buffer (also skipping a check to see if the buffer has enough space to hold the lookahead character). It later writes a \ character into the buffer without a check if the lookahead character was \ and not -1. If the execution platform performs sign extension on character to integer assignments, an attack message containing an appropriately placed alternating sequence of -1 and \ characters in the address can therefore cause the prescan to write arbitrarily many \ characters beyond the end of the buffer.

4.4.2 Security and Resilience

The Standard version of Sendmail performs the out of bounds writes and corrupts its call stack. It is apparently possible for an attacker to exploit the memory error to cause the Sendmail server to execute arbitrary injected code [14]. The Bounds Check version exits with a memory error during initialization and fails to operate at all. The Failure Oblivious version is not vulnerable to the attack — when sent the attack message, it discards the out of bounds writes (preserving the integrity of the stack) and returns back out of the prescan to continue to parse the email address. The next step is to check if the input mail address is too long. This check fails, throwing Sendmail into an anticipated error case. The standard error processing logic in Sendmail then rejects the address, enabling Sendmail to continue on to successfully process subsequent commands.

4.4.3 Performance

Figure 4 presents the means and standard deviations of the request processing times for the Standard and Failure Oblivious versions of Sendmail. All times are given in milliseconds. The Receive Small request receives a message whose body is 4 bytes long; the Send Small request sends the same message. The Receive Large request receives a message whose body is 4Kbytes long; the Send Large request sends the same message. We performed each test at least twenty times to obtain the numbers in Figure 4.

Request	Standard	Failure Oblivious	Slowdown
Recv Small	$15.6 \pm 2.9\%$	$60.4 \pm 1.5\%$	3.9
Recv Large	$16.8 \pm 4.3\%$	$65.1 \pm 2.3\%$	3.9
Send Small	$20.3 \pm 3.2\%$	$75.0 \pm 3.4\%$	3.7
Send Large	$21.5 \pm 5.7\%$	$76.9 \pm 3.8\%$	3.6

Figure 4: Request Processing Times for Sendmail (milliseconds)

4.4.4 Stability

We installed the Failure Oblivious version of Sendmail on one of our machines and, over the course of several days, used it to send and receive hundreds of thousands of email messages. During this time we repeatedly sent the attack message through the Sendmail daemon, which continued through the attack to correctly process all subsequent Sendmail commands. All of the messages were correctly delivered with no problems. Our memory error logs indicate that Sendmail generates a steady stream of memory errors during its normal execution. In particular, every time the Sendmail daemon wakes up to check for incoming messages, it generates a memory error. This memory error apparently completely disables the Bounds Check version.

4.5 Midnight Commander

Midnight Commander is an open source file management tool that allows users to browse files and archives, copy files from one folder to another, and delete files [6]. Midnight Commander is vulnerable to a memory-error attack associated with accessing an uninitialized buffer when processing symbolic links in `tgz` archives [5]. We used Midnight Commander version 4.5.55 for our experiments.

4.5.1 The Memory Error

Midnight Commander converts absolute symbolic links in `tgz` files into links relative to the start of the `tgz` file. It uses the `strcat` procedure to build up the name of the relative link in a stack-allocated buffer. Unfortunately, the buffer is never initialized. If there are multiple symbolic links in the directory, the component names from all of the links simply accumulate sequentially in the buffer as Midnight Commander processes the set of links. If the combined length of all of the component names exceeds the length of the buffer, `strcat` writes the component names beyond the end of the buffer.

4.5.2 Security and Resilience

The Standard version performs the writes, corrupts its stack, and terminates with a segmentation violation. The Bounds Check version detects the out of bounds access and terminates. The Failure Oblivious version discards the out of bounds writes, enabling Midnight Commander to continue and attempt to look up the data for the referenced file. This lookup always fails (apparently even for the first symbolic link, when the name in the buffer is correct). This is an anticipated case in the Midnight Commander code, which treats the symbolic link as a dangling link and displays it as such to the user. Midnight Commander then continues on to successfully process any subsequent user commands.

4.5.3 Performance

Figure 5 presents the request processing times for the Standard and Failure Oblivious versions of Midnight Commander. The Copy request copies a 31Mbyte directory structure, the Move request moves a directory of the same size, the MkDir request makes a new directory, and the Delete request deletes a 3.2 Mbyte file. We performed each request at least twenty times and report the means and standard deviations of the request processing times. All times are given in milliseconds.

As these numbers indicate, the Failure Oblivious version is not dramatically slower than the Standard version. Moreover, because Midnight Commander is an interactive program, its performance is acceptable as long as it feels responsive to its users, and these performance results make it clear that the application of failure-

Request	Standard	Failure Oblivious	Slowdown
Copy	$377 \pm 0.7\%$	$535 \pm 2.0\%$	1.4
Move	$0.30 \pm 2.4\%$	$0.406 \pm 1.8\%$	1.4
MkDir	$0.69 \pm 7.0\%$	$1.27 \pm 6.6\%$	1.8
Delete	$2.54 \pm 11.3\%$	$2.72 \pm 11.1\%$	1.1

Figure 5: Request Processing Times for Midnight Commander (milliseconds)

oblivious computing to this program should not degrade its interactive feel. Our subjective experience confirms this expectation: all pause times are imperceptible for both the Standard and Failure Oblivious versions.

4.5.4 Stability

One of the authors uses Midnight Commander on a daily basis as his standard file manipulation tool. During the stability testing period, he used the Failure Oblivious version of Midnight Commander to manage his files. Periodically during the sessions he attempted to open the problematic archive (causing the program to execute through the resulting memory error), then went back to using the Midnight Commander to accomplish his work. Midnight Commander performed without a problem during this time.

The error log shows that Midnight Commander has a memory error that is triggered whenever a blank line occurs in its configuration file. We verified that this error completely disabled the Bounds Check version until we removed the blank lines. The Failure Oblivious version, on the other hand, executed successfully through all memory errors to perform flawlessly for all requests.

4.6 Mutt

Mutt is a customizable, text-based mail user agent that is widely used in the Unix system administration community [8]. It is descended from ELM [4] and supports a variety of features including email threading and correct NFS mail spool locking. We used Mutt version 1.4. As described at [7] and discussed in Section 2, this version is vulnerable to an attack that exploits a memory error in the conversion from UTF-8 to UTF-7 string formats.

4.6.1 The Memory Error

When Mutt opens a mailbox with an IMAP address, it converts the mail folder name from UTF-8 to UTF-7 character encoding. Mutt allocates (in the heap) a temporary character buffer to hold the UTF-7 encoded name. Because UTF-8 to UTF-7 conversion can increase the length of the name, Mutt allocates a buffer twice as long as the UTF-8 name to hold the converted UTF-7 name. However, this buffer is not, in general, long enough — the conversion can increase the length of the UTF-8 name by as much as a factor of 7/3 and not just a factor of 2. When presented with an appropriately constructed UTF-8 folder name, Mutt writes the converted name beyond the end of the UTF-7 buffer.

4.6.2 Security and Resilience

The Standard version performs the writes, corrupts its heap, and terminates with a segmentation violation. The Bounds Check version detects the memory error and terminates before the user interface comes up. The Failure Oblivious version discards the out of bounds writes, effectively truncating the converted name. Note that even though the UTF-7 buffer may contain no null characters, the folder name is effectively null-terminated: reads beyond the end of the buffer will eventually return null. Once Mutt has obtained the converted folder name, the next step is to place a quoted and escaped version of the name into yet another buffer, then pass this name on as part of a command to the IMAP server. The IMAP server returns an error code indicating that the folder does not exist, Mutt's standard error-handling logic handles the returned error code, and Mutt continues on to successfully process any subsequent user commands.

4.6.3 Performance

Figure 6 presents the request processing times for the Standard and Failure Oblivious versions of Mutt. The Read request reads a selected empty message and the Move request moves an empty message from one folder to another. We performed each request at least twenty times and report the means and standard deviations of the request processing times. All times are given in milliseconds.

Request	Standard	Failure Oblivious	Slowdown
Read	$.655 \pm 4.3\%$	$2.31 \pm 4.8\%$	3.6
Move	$6.94 \pm 6.2\%$	$9.78 \pm 6.2\%$	1.4

Figure 6: Request Processing Times for Mutt (milliseconds)

Because Mutt is an interactive program, its performance is acceptable as long as it feels responsive to its users. These performance results make it clear that the application of failure-oblivious computing to this program should not degrade its interactive feel. Our subjective experience confirms this expectation: all pause times are imperceptible for both the Standard and Failure Oblivious versions.

4.6.4 Stability

During the stability testing period we used the Failure Oblivious version of Mutt to process email messages. We configured Mutt to trigger the security vulnerability described above when it loaded. Mutt successfully executed through the resulting memory errors to correctly

execute all of his requests. We were able to read, forward, and compose mail with no problems even after executing through the memory error. We also used Mutt to process (with no problems) a large mail folder containing over 100,000 messages.

4.7 Discussion

Despite the fact that the dynamic bounds checks have, in theory, the potential to substantially degrade the performance, for several of our servers the overhead is relatively small — the execution times of many of the tasks we measured are apparently dominated by activities (such I/O or operating system functionality) outside the program. Because failure-oblivious computing does not affect the efficiency of these activities, the amortized overhead is relatively small. Moreover, several of our servers are interactive, and interactive tasks can tolerate substantial execution time increases as long as the system maintains its interactive feel. Our results show that failure-oblivious computing maintained acceptable interactive response times for all of our interactive tasks, even for tasks with substantial execution time increases.

For servers, a monitor that detects memory errors and reboots the server when it commits such an error might seem to provide an obvious potential alternative to failure-oblivious computing. Apache, for example, implements a regenerating pool of child processes. The net effect is that the Bounds Check version of Apache can terminate child processes at the first memory error without impairing its ability to continue to service new requests. In comparison with the Failure Oblivious version, the only downside is the performance degradation associated with the resulting increase in process management overhead.

The situation is somewhat different for Pine, Mutt, and Midnight Commander. All of these programs initialize with no memory errors on standard workloads. But once the mailbox contains a message that elicits a memory error (Pine), the system is configured to use a mail folder whose name elicits a memory error (Mutt), or the configuration file contains a blank line (Midnight Commander), the Bounds Check versions exit during initialization. In this situation, restarting is of no use because the restarted computations would, once again, simply exit during initialization. Because these errors are triggered only by carefully crafted or unusual inputs, they could easily make it through a fairly rigorous testing process without being detected. These servers illustrate how aggressively terminating computations at the first memory error can leave deployed systems vulnerable to unanticipated inputs that trigger memory errors and persist or recurr in the environment.

Because Sendmail has a memory error whenever it wakes up to check for work, the Bounds Check version is simply unusable with or without restarting. But note that because the memory errors occur on every execution, it should be possible to use the Bounds Check version to find and eliminate them (as well as any other reproducible memory errors that occur during testing). Even with this change, however, terminating and restarting Sendmail might prove to be problematic — the Sendmail monitor would somehow have to avoid repeatedly presenting Sendmail with messages that triggered a memory error. In contrast, the Failure Oblivious version of Sendmail correctly executed through memory errors to correctly process subsequent messages and the Failure Oblivious version of Pine correctly processed mail messages with headers that elicited memory errors.

5 Related Work

We first note that failure-oblivious computing is an instance of acceptability-oriented computing [47]. Acceptability-oriented computing replaces the concept of program correctness with a set of *acceptability properties* that must hold for the execution of the program to remain acceptable. The programmer then builds and deploys *acceptability enforcement mechanisms* whose actions ensure that these acceptability properties do, in fact, hold. In the case of failure-oblivious computing, the acceptability properties are the absence of memory errors and continued execution; the acceptability enforcement mechanism discards invalid writes and returns manufactured values for invalid reads.

Memory errors, failures, and failure recovery have been core concerns in the field of computer systems since its inception. We discuss related work in these areas.

5.1 Variants and Extensions

We have implemented with several variants and extensions of our basic failure-oblivious compiler. These include a compiler that implements *boundless memory blocks* — instead of discarding invalid writes, the generated code stores the values in a hash table indexed under the data unit identifier and offset [48]. Corresponding invalid reads return the appropriate stored values. This variant eliminates size calculation errors — if the program logic is otherwise acceptable, the program will execute acceptably. Another variant redirects out of bounds accesses back into the accessed data unit at an appropriate offset. This strategy may help related sets of out of bounds reads return consistent values from properly initialized data units. Our experience indicates that our set of servers works acceptably with both of these variants.

5.2 Transactional Function Termination

Researchers have also developed a technique to protect servers against buffer-overflow attacks by dynamically detecting buffer overflows, then immediately terminating

the enclosing function and continuing on to execute the code immediately following the corresponding function call [52]. The results indicate that, in many cases, the program can continue on to execute acceptably after the premature function termination. This experience is consistent with our experience that servers can continue to execute successfully through memory errors if they simply discard out of bounds writes and manufacture values for out of bounds reads.

5.3 Safe-C Compilers

Our work builds directly on previous research into memory-safe C implementations [17, 58, 45, 36, 50, 37]. Building on Ruwase and Lam's implementation enabled us to apply failure-oblivious computing directly to legacy programs without modification (some implementations also have this property [58]); some other implementations may require source code changes [22, 38].

It is also feasible to apply failure-oblivious computing to safe languages such as Java or ML by simply replacing the generated code that throws an exception in response to a memory error. As for safe-C implementations, the new code would simply discard illegal writes and return manufactured values for illegal reads.

5.4 Static Analysis

It is also possible to attack the memory error problem directly at its source: a combination of static analysis and program annotations should, in principle, enable programmers to deliver programs that are completely free of memory errors [28, 27, 57, 49]. All of these techniques share the same advantage (a static guarantee that the program will not exhibit a specific kind of memory error) and drawbacks (the need for programmer annotations or the possibility of conservatively rejecting safe programs). Even if the analysis is not able to verify that the entire program is free of memory errors, it may be able to statically recognize some accesses that will never cause a memory error, remove the dynamic checks for those accesses, and thereby reduce any dynamic checking overhead [32, 18, 49].

Researchers have also developed unsound, incomplete analyses that heuristically identify potential errors [54, 19]. The advantage is that such approaches typically require no annotations and scale better to larger programs; the disadvantage is that (because they are unsound) they may miss some genuine memory errors.

5.5 Buffer-Overrun Detection Tools

Researchers have developed techniques that are designed to detect buffer-overrun attacks after they have occurred, then halt the execution of the program before the attack can take effect. StackGuard [23] and StackShield [16] modify the compiler to generate code to detect attacks that overwrite the return address on the stack; StackShield also performs range checks to detect overwritten function pointers. It is also possible to apply buffer-overrun detection directly to binaries. Purify instruments the binary to detect a range of memory errors, including buffer overruns [34]. Program shepherding uses an efficient binary interpreter to prevent an attacker from executing injected code [39]. A key difference is that failure-oblivious computing prevents the attack from performing the writes that corrupt the address space, which enables the program to continue to execute successfully.

5.6 Rebooting

A traditional and widely used error recovery mechanism is to reboot the system, with repair applied during the reboot if necessary to bring the system back up successfully [30]. Mechanisms such as fast reboots [51] and checkpointing [41, 42] can improve the performance of the basic reboot process.

It is also possible to subdivide (potentially recursively) a system into isolated components, then apply a partial reboot strategy at the granularity of the components. By promoting the construction of the operating system as a collection of small components, microkernel architectures [46, 33, 29] support the application of this approach to operating systems. It is also possible to use mechanisms such as software-based fault isolation [55] or fine-grained hardware memory protection [56] to apply this strategy to selected parts of monolithic operating systems such as kernel extensions. The experimental results show that this approach can eliminate the vast majority of system crashes caused by such extensions [53]. Helper agents are often useful to facilitate the clean termination and reintegration of the restarted component back into the running system (this approach generalizes to support arbitrary recovery actions) [53]. It may also be worthwhile to recursively restart larger and larger subsystems until the system successfully recovers [20].

Failure-oblivious computing differs in that it is designed to keep the system operating through errors instead of restarting. The potential advantages include better availability because of the elimination of down time and the elimination of vulnerabilities to persistent errors. Rebooting, on the other hand, may help ensure that the system stays more closely within the anticipated operating envelope.

5.7 Manual Error Detection and Recovery

Motivated in part by the need to avoid rebooting, researchers have developed more fine-grain error recovery mechanisms. The Lucent 5ESS switch and the IBM MVS operating system, for example, both contain software components that detect and attempt to repair inconsistent data structures [35, 44, 31]. Other techniques

include failure recovery blocks and exception handlers, both of which may contain hand-coded recovery algorithms [43].

To apply these techniques, the programmer must anticipate some aspects of the error and, based on this understanding, develop an appropriate recovery strategy. Failure-oblivious computing, on the other hand, can be applied without programmer intervention to any system and may therefore make the system oblivious to even completely unanticipated errors. Of course, this generality cuts both ways — in particular, failure-oblivious computing may produce less appropriate responses to anticipated errors. We therefore view failure-oblivious computing as largely orthogonal to more application-tailored recovery mechanisms (although failure-oblivious computing may eliminate some of the errors that these mechanisms would otherwise have handled).

Data structure repair [26] occupies a middle ground. Like more traditional error detection and recovery techniques, it requires the programmer to provide some application-specific information (in the case of data structure repair, a data structure consistency specification). But because there is no explicit recovery procedure and because the consistency specification is not tied to specific blocks of code, data structure repair may enable systems to more effectively recover from unanticipated data structure corruption errors.

6 Conclusion

The seemingly inherent brittleness, complexity, and vulnerability (to both errors and attacks) of computer programs can make them frustrating or even dangerous to use. While existing memory-safe languages and memory-safe implementations of unsafe languages may eliminate memory-error vulnerabilities, they can also decrease availability by aggressively throwing exceptions or even terminating the program at the first sign of an error.

Our results show that failure-oblivious computation enhances availability, resilience, and security by continuing to execute through memory errors while ensuring that such errors do not corrupt the address space or data structures of the computation. In many cases failure-oblivious computing can automatically convert unanticipated and dangerous inputs or data into anticipated error cases that the program is designed to handle correctly. The result is that the program survives the unanticipated situation, returns back into its normal operating envelope, and continues to satisfy the needs of its users.

One of the major long-term goals of computer science has been understanding how to build more robust, resilient programs that can flexibly and successfully cope with unanticipated situations. Our research suggests that, remarkably, current systems may already have a substantial capacity for exhibiting this kind of desirable behavior if we only provide a way for them to ignore their errors, protect their data structures from damage, and continue to execute.

Acknowledgements

The authors would like to thank our shepherd David Wagner and the anonymous reviewers for their thoughtful and helpful comments. This research was supported in part by the Singapore-MIT Alliance and NSF grants CCR00-86154, CCR00-63513, CCR00-73513, CCR-0209075, CCR-0341620, and CCR-0325283.

References

[1] Apache HTTP Server exploit. www.securityfocus.com/bid/8911/discussion/.

[2] CERT/CC. Advisories 2002. www.cert.org/advisories.

[3] CNN Report on Code Red. www.cnn.com/2001/TECH/internet/08/08/code.red.II/.

[4] ELM. www.instinct.org/elm/.

[5] Midnight Commander exploit. www.securityfocus.com/bid/8658/discussion/.

[6] Midnight Commander website. www.ibiblio.org/mc/.

[7] Mutt exploit. www.securiteam.com/unixfocus/5FP0T0U9FU.html.

[8] Mutt website. www.mutt.org.

[9] Netcraft website. http://news.netcraft.com/archives/web_server_survey.html.

[10] Pine exploit. www.securityfocus.com/bid/6120/discussion.

[11] Pine website. www.washington.edu/pine/.

[12] SecuriTeam website. www.securiteam.com.

[13] Security Focus website. www.securityfocus.com.

[14] Sendmail exploit. www.securityfocus.com/bid/7230/discussion/.

[15] Sendmail website. www.sendmail.org.

[16] Stackshield. www.angelfire.com/sk/stackshield.

[17] T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 2004.

[18] R. Bodik, R. Gupta, and V. Sarkar. Eliminating array bounds checks on demand. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2002.

[19] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic, programming errors. *Software - Practice and Experience*, 2000.

[20] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 110–115, Schloss Elmau, Germany, May 2001.

[21] S. Card, T. Moran, and A. Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983.

[22] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, June 2003.

[23] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Conference*, January 1998.

[24] J. Darley and B. Latane. Bystander intervention in emergencies: Diffusion of responsibility. *Journal of Personality and Social Psychology*, pages 377–383, Aug. 1968.

[25] W. E. Deming. *Out of the Crisis*. MIT Press, 2000.

[26] B. Demsky and M. Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2003.

[27] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, June 2003.

[28] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.

[29] D. Engler, M. F. Kaashoek, and J. James O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles* , Dec. 1995.

[30] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[31] N. Gupta, L. Jagadeesan, E. Koutsofios, and D. Weiss. Auditdraw: Generating audits the FAST way. In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, 1997.

[32] R. Gupta. Optimizing array bounds checks using flow analysis. In *ACM Letters on Programming Languages and Systems, 2(1-4):135-150*, March 1993.

[33] G. Hamilton and P. Kougiouris. The Spring Nucleus: A Microkernel for Objects. In *Proceedings of the 1993 Summer Usenix Conference* , June 1993.

[34] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.

[35] G. Haugk, F. Lax, R. Royer, and J. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Technical Journal*, 64(6 part 2):1385–1416, July-August 1985.

[36] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, June 2002.

[37] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of Third International Workshop On Automatic Debugging*, May 1997.

[38] S. C. Kendall. Bcc: run-time checking for C programs. In *USENIX Summer Conference Proceedings*, 1983.

[39] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of 11th USENIX Security Symposium*, August 2002.

[40] B. Latane and J. Darley. Group inhibition of bystander intervention in emergencies. *Journal of Personality and Social Psychology*, pages 215–221, Oct. 1968.

[41] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, 1988.

[42] M. Litzkow and M. Solomon. The Evolution of Condor Checkpointing.

[43] M. R. Lyu. *Software Fault Tolerance*. John Wiley & Sons, 1995.

[44] S. Mourad and D. Andrews. On the reliability of the IBM MVS/XA operating system. *IEEE Transactions on Software Engineering*, September 1987.

[45] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, 2002.

[46] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: A New Kernel Foundation For UNIX Development. In *Proceedings of the 1986 Summer USENIX Conference* , July 1986.

[47] M. Rinard. Acceptability-oriented computing. In *2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Companion (OOPSLA '03 Companion) Onwards! Session*, Oct. 2003.

[48] M. Rinard, C. Cadar, D. Roy, D. Dumitran, and T. Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *Proceedings of the 20th Annual Computer Security Applications Conference*, Dec. 2004.

[49] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation* , June 2000.

[50] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, February 2004.

[51] M. I. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. In *Proceedings of the Sixth workshop on Hot Topics in Operating Systems*, 1997.

[52] S. Sidiroglou, G. Giovanidis, and A. Keromytis. Using execution transactions to recover from buffer overflow attacks. Technical Report CUCS-031-04, Columbia University Computer Science Department, September 2004.

[53] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the Nineteenth ACM Symposium on Operating System Principles* , Dec. 2003.

[54] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the Year 2000 Network and Distributed System Security Symposium*, 2000.

[55] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles* , Dec. 1994.

[56] E. Witchel, J. Cates, and K. Asanovic. Mondriaan memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[57] H. Xi and F. Pfenning. Eliminating Array Bound Checking Through Dependent Types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.

[58] S. H. Yong and S. Horwitz. Protecting C Programs from Attacks via Invalid Pointer Dereferences. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, 2003.

[59] Y. Zhang, J. Yang, and R. Gupta. Frequent value locality and value-centric data cache design. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr
University of Utah, School of Computing
{jxyang, chenyang, eeide, regehr}@cs.utah.edu

Abstract

Compilers should be correct. To improve the quality of C compilers, we created Csmith, a randomized test-case generation tool, and spent three years using it to find compiler bugs. During this period we reported more than 325 previously unknown bugs to compiler developers. Every compiler we tested was found to crash and also to silently generate wrong code when presented with valid input. In this paper we present our compiler-testing tool and the results of our bug-hunting study. Our first contribution is to advance the state of the art in compiler testing. Unlike previous tools, Csmith generates programs that cover a large subset of C while avoiding the undefined and unspecified behaviors that would destroy its ability to automatically find wrong-code bugs. Our second contribution is a collection of qualitative and quantitative results about the bugs we have found in open-source C compilers.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—testing tools; D.3.2 [*Programming Languages*]: Language Classifications—C; D.3.4 [*Programming Languages*]: Processors—compilers

General Terms Languages, Reliability

Keywords compiler testing, compiler defect, automated testing, random testing, random program generation

1. Introduction

The theory of compilation is well developed, and there are compiler frameworks in which many optimizations have been proved correct. Nevertheless, the practical art of compiler construction involves a morass of trade-offs between compilation speed, code quality, code debuggability, compiler modularity, compiler retargetability, and other goals. It should be no surprise that optimizing compilers—like all complex software systems—contain bugs.

Miscompilations often happen because optimization safety checks are inadequate, static analyses are unsound, or transformations are flawed. These bugs are out of reach for current and future automated program-verification tools because the specifications that need to be checked were never written down in a precise way, if they were written down at all. Where verification is impractical, however, other methods for improving compiler quality can succeed. This paper reports our experience in using testing to make C compilers better.

The definitive version was published in *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Jose, CA, Jun. 2011, http://doi.acm.org/10.1145/NNNNNNN.NNNNNNN

```
1  int foo (void) {
2    signed char x = 1;
3    unsigned char y = 255;
4    return x > y;
5  }
```

Figure 1. We found a bug in the version of GCC that shipped with Ubuntu Linux 8.04.1 for x86. At all optimization levels it compiles this function to return 1; the correct result is 0. The Ubuntu compiler was heavily patched; the base version of GCC did not have this bug.

We created Csmith, a randomized test-case generator that supports compiler bug-hunting using differential testing. Csmith generates a C program; a test harness then compiles the program using several compilers, runs the executables, and compares the outputs. Although this compiler-testing approach has been used before [6, 16, 23], Csmith's test-generation techniques substantially advance the state of the art by generating random programs that are expressive—containing complex code using many C language features—while also ensuring that every generated program has a single interpretation. To have a unique interpretation, a program must not execute any of the 191 kinds of undefined behavior, nor depend on any of the 52 kinds of unspecified behavior, that are described in the C99 standard.

For the past three years, we have used Csmith to discover bugs in C compilers. Our results are perhaps surprising in their extent: to date, we have found and reported more than 325 bugs in mainstream C compilers including GCC, LLVM, and commercial tools. Figure 1 shows a representative example. Every compiler that we have tested, including several that are routinely used to compile safety-critical embedded systems, has been crashed and also shown to silently miscompile valid inputs. As measured by the responses to our bug reports, the defects discovered by Csmith are important. Most of the bugs we have reported against GCC and LLVM have been fixed. Twenty-five of our reported GCC bugs have been classified as P1, the maximum, release-blocking priority for GCC defects. Our results suggest that fixed test suites—the main way that compilers are tested—are an inadequate mechanism for quality control.

We claim that Csmith is an effective bug-finding tool in part because it generates tests that explore atypical combinations of C language features. Atypical code is *not* unimportant code, however; it is simply underrepresented in fixed compiler test suites. Developers who stray outside the well-tested paths that represent a compiler's "comfort zone"—for example by writing kernel code or embedded systems code, using esoteric compiler options, or automatically generating code—can encounter bugs quite frequently. This is a significant problem for complex systems. Wolfe [30], talking about independent software vendors (ISVs) says: "An ISV with a complex code can work around correctness, turn off the optimizer in one or two files, *and usually they have to do that for any of the compilers they use*" (emphasis ours). As another example, the front

page of the Web site for GMP, the GNU Multiple Precision Arithmetic Library, states, "Most problems with compiling GMP these days are due to problems not in GMP, but with the compiler."

Improving the correctness of C compilers is a worthy goal: C code is part of the trusted computing base for almost every modern computer system including mission-critical financial servers and life-critical pacemaker firmware. Large-scale source-code verification efforts such as the seL4 OS kernel [12] and Airbus's verification of fly-by-wire software [24] can be undermined by an incorrect C compiler. The need for correct compilers is amplified because operating systems are almost always written in C and because C is used as a portable assembly language. It is targeted by code generators from a wide variety of high-level languages including Matlab/Simulink, which is used to generate code for industrial control systems.

Despite recent advances in compiler verification, testing is still needed. First, a verified compiler is only as good as its specification of the source and target language semantics, and these specifications are themselves complex and error-prone. Second, formal verification seldom provides end-to-end guarantees: "details" such as parsers, libraries, and file I/O usually remain in the trusted computing base. This second point is illustrated by our experience in testing CompCert [14], a verified C compiler. Using Csmith, we found previously unknown bugs in unproved parts of CompCert—bugs that cause this compiler to silently produce incorrect code.

Our goal was to discover serious, previously unknown bugs:

- in mainstream C compilers like GCC and LLVM;
- that manifest when compiling core language constructs such as arithmetic, arrays, loops, and function calls;
- targeting ubiquitous architectures such as x86 and x86-64; and
- using mundane optimization flags such as –O and –O2.

This paper reports our experience in achieving this goal. Our first contribution is to advance the state of the art in compiler test-case generation, finding—as far as we know—many more previously unknown compiler bugs than any similar effort has found. Our second contribution is to qualitatively and quantitatively characterize the bugs found by Csmith: What do they look like? In what parts of the compilers are they primarily found? How are they distributed across a range of compiler versions?

2. Csmith

Csmith began as a fork of Randprog [27], an existing random C program generator about 1,600 lines long. In earlier work, we extended and adapted Randprog to find bugs in C compilers' translation of accesses to volatile-qualified objects [6], resulting in a 7,000-line program. Our previous paper showed that in many cases, these bugs could be worked around by turning volatile-object accesses into calls to helper functions. The key observation was this: while the rules regarding the addition, elimination, and reordering of accesses to volatile objects are not at all like the rules governing ordinary variable accesses in C, they are almost identical to the rules governing function calls.

For some test programs generated by Randprog, our rewriting procedure was insufficient to correct a defect that we had found in the C compiler. Our hypothesis was that this was always due to "regular" compiler bugs not related to the volatile qualifier. To investigate these compiler defects, we shifted our research emphasis toward looking for generic wrong-code bugs. We turned Randprog into Csmith, a 40,000-line C++ program for randomly generating C programs. Compared to Randprog, Csmith can generate C programs that utilize a much wider range of C features including complex control flow and data structures such as pointers, arrays, and structs. Most of Csmith's complexity arises from the requirement that it interleave static analysis with code generation in order to produce meaningful test cases, as described below.

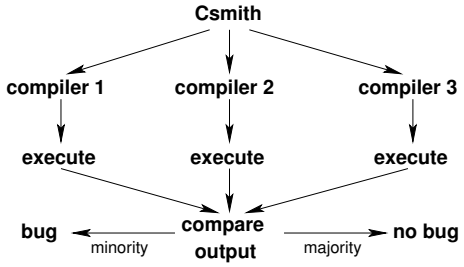


Figure 2. Finding bugs in three compilers using randomized differential testing

2.1 Randomized Differential Testing using Csmith

Random testing [9], also called fuzzing [17], is a black-box testing method in which test inputs are generated randomly. Randomized differential testing [16] has the advantage that no oracle for test results is needed. It exploits the idea that if one has multiple, deterministic implementations of the same specification, all implementations must produce the same result from the same valid input. When two implementations produce different outputs, one of them must be faulty. Given three or more implementations, a tester can use voting to heuristically determine which implementations are wrong. Figure 2 shows how we use these ideas to find compiler bugs.

2.2 Design Goals

Csmith has two main design goals. First and most important, every generated program must be well formed and have a single meaning according to the C standard. The meaning of a C program is the sequence of side effects it performs. The principal side effect of a Csmith-generated program is to print a value summarizing the computation performed by the program.[1] This value is a checksum of the program's non-pointer global variables at the end of the program's execution. Thus, if changing the compiler or compiler options causes the checksum emitted by a Csmith-generated program to change, a compiler bug has been found.

The C99 language [11] has 191 *undefined behaviors*—e.g., dereferencing a null pointer or overflowing a signed integer—that destroy the meaning of a program. It also has 52 *unspecified behaviors*—e.g., the order of evaluation of arguments to a function—where a compiler may choose from a set of options with no requirement that the choice be made consistently. Programs emitted by Csmith must avoid all of these behaviors or, in certain cases such as argument-evaluation order, be independent of the choices that will be made by the compiler. Many undefined and unspecified behaviors can be avoided structurally by generating programs in such a way that problems never arise. However, a number of important undefined and unspecified behaviors are not easy to avoid in a structural fashion. In these cases, Csmith solves the problem using static analysis and by adding run-time checks to the generated code. Section 2.4 describes the hazards that Csmith must avoid and its strategies for avoiding them.

Csmith's second design goal is to maximize expressiveness subject to constraints imposed by the first goal. An "expressive" generator supports many language features and combinations of features. Our hypothesis is that expressiveness is correlated with bug-finding power.

[1] Accesses to volatile objects are also side effects as described in the C standard. We do not discuss these "secondary" side effects of Csmith-generated programs further in this paper.

Csmith creates programs with the following features:

- function definitions, and global and local variable definitions
- most kinds of C expressions and statements
- control flow: `if/else`, function calls, `for` loops, `return`, `break`, `continue`, `goto`
- signed and unsigned integers of all standard widths
- arithmetic, logical, and bitwise operations on integers
- structs: nested, and with bit-fields
- arrays of and pointers to all supported types, including pointers and arrays
- the `const` and `volatile` type qualifiers, including at different levels of indirection for pointer-typed variables

The most important language features not currently supported by Csmith are strings, dynamic memory allocation, floating-point types, unions, recursion, and function pointers. We plan to add some of these features to future versions of our tool.

2.3 Randomly Generating Programs

The shape of a program generated by Csmith is governed by a grammar for a subset of C. A program is a collection of type, variable, and function definitions; a function body is a block; a block contains a list of declarations and a list of statements; and a statement is an expression, control-flow construct (e.g., `if`, `return`, `goto`, or `for`), assignment, or block. Assignments are modeled as statements—not expressions—which reflects the most common idiom for assignments in C code. We leverage our grammar to produce other idiomatic code as well: in particular, we include a statement kind that represents a loop iterating over an array. The grammar is implemented by a collection of hand-coded C++ classes.

Csmith maintains a *global environment* that holds top-level definitions: i.e., types, global variables, and functions. The global environment is extended as new entities are defined during program generation. To hold information relevant to the current program-generation point, Csmith also maintains a *local environment* with three primary kinds of information. First, the local environment describes the current call chain, supporting context-sensitive pointer analysis. Second, it contains effect information describing objects that may have been read or written since (1) the start of the current function, (2) the start of the current statement, and (3) the previous sequence point.[2] Third, the local environment carries points-to facts about all in-scope pointers. These elements and their roles in program generation are further described in Section 2.4.

Csmith begins by randomly creating a collection of struct type declarations. For each, it randomly decides on a number of members and the type of each member. The type of a member may be a (possibly qualified) integral type, a bit-field, or a previously generated struct type.

After the preliminary step of producing type definitions, Csmith begins to generate C program code. Csmith generates a program top-down, starting from a single function called by `main`. Each step of the program generator involves the following sub-steps:

1. Csmith randomly selects an allowable production from its grammar for the current program point. To make the choice, it consults a probability table and a filter function specific to the current point: there is a table/filter pair for statements, another for expressions, and so on. The table assigns a probability to each of the alternatives, where the sum of the probabilities is one. After choosing a production from the table, Csmith executes the filter, which decides if the choice is acceptable in the current context. Filters enforce basic semantic restrictions (e.g., `continue` can only appear within a loop), user-controllable limits (e.g., maximum statement depth and number of functions), and other user-controllable options. If the filter rejects the selected production, Csmith simply loops back, making selections from the table until the filter succeeds.
2. If the selected production requires a target—e.g., a variable or function—then the generator randomly selects an appropriate target or defines a new one. In essence, Csmith dynamically constructs a probability table for the potential targets and includes an option to create a new target. Function and variable definitions are thus created "on demand" at the time that Csmith decides to refer to them.
3. If the selected production allows the generator to select a type, Csmith randomly chooses one. Depending on the current context, the choice may be restricted (e.g., while generating the operands of an integral-typed expression) or unrestricted (e.g., while generating the types of parameters to a new function). Random choices are guided by the grammar, probability tables, and filters as already described.
4. If the selected production is nonterminal, the generator recurses. It calls a function to generate the program fragment that corresponds to the nonterminal production. More generally, Csmith recurses for each nonterminal element of the current production: e.g., for each subcomponent of a compound statement, or for each parameter in a function call.
5. Csmith executes a collection of dataflow transfer functions. It passes the points-to facts from the local environment to the transfer functions, which produce a new set of points-to facts. Csmith updates the local environment with these facts.
6. Csmith executes a collection of safety checks. If the checks succeed, the new code fragment is committed to the generated program. Otherwise, the fragment is dropped and any changes to the local environment are rolled back.

When Csmith creates a call to a new function—one whose body does not yet exist—generation of the current function is suspended until the new function is finished. Thus, when the top-level function has been completely generated, Csmith is finished. At that point it pretty-prints all of the randomly generated definitions in an appropriate order: types, globals, prototypes, and functions. Finally, Csmith outputs a `main` function. The `main` function calls the top-level randomly generated function, computes a checksum of the non-pointer global variables, prints the checksum, and exits.

2.4 Safety Mechanisms

Table 1 lists the mechanisms that Csmith uses to avoid generating C programs that execute undefined behaviors or depend on unspecified behaviors. This section provides additional detail about the hazards that Csmith must avoid and its strategies for avoiding them.

Integer safety More and more, compilers are aggressively exploiting the undefined nature of integer behaviors such as signed overflow and shift-past-bitwidth. For example, recent versions of Intel CC, GCC, and LLVM evaluate `(x+1)>x` to 1 while also evaluating `(INT_MAX+1)` to `INT_MIN`. In another example, discovered by the authors of Google's Native Client software [3], routine refactoring of C code caused the expression `1<<32` to be evaluated on a

[2] As explained in Section 3.8 of the C FAQ [25], "A sequence point is a point in time at which the dust has settled and all side effects which have been seen so far are guaranteed to be complete. The sequence points listed in the C standard are at the end of the evaluation of a full expression (a full expression is an expression statement, or any other expression which is not a subexpression within any larger expression); at the `||`, `&&`, `?:`, and comma operators; and at a function call (after the evaluation of all the arguments, and just before the actual call)."

Problem	Code-Generation-Time Solution	Code-Execution-Time Solution
use without initialization	explicit initializers, avoid jumping over initializers	—
qualifier mismatch	static analysis	—
infinite recursion	disallow recursion	—
signed integer overflow	bounded loop vars	safe math wrappers
OOB array access	bounded loop vars	force index in bounds
unspecified eval. order of function arguments	effect analysis	—
R/W and W/W conflicts betw. sequence points	effect analysis	—
access to out-of-scope stack variable	pointer analysis	—
null pointer dereference	pointer analysis	null pointer checks

Table 1. Summary of Csmith's strategies for avoiding undefined and unspecified behaviors. When both a code-generation-time and code-execution-time solution are listed, Csmith uses both.

platform with 32-bit integers. The compiler exploited this undefined behavior to turn a sandboxing safety check into a nop.

To keep Csmith-generated programs from executing integer undefined behaviors, we implemented a family of wrapper functions for arithmetic operators whose (promoted) operands might overflow. This was not difficult, but had a few tricky aspects. For example, the C99 standard does not explicitly identify the evaluation of `INT_MIN%-1` as being an undefined behavior, but most compilers treat it as such. The C99 standard also has very restrictive semantics for signed left-shift: it is illegal (for implementations using 2's complement integers) to shift a 1-bit into or past the sign bit. Thus, evaluating `1<<31` destroys the meaning of a C99 program on a platform with 32-bit ints.

Several safe math libraries for C that we examined themselves execute operations with undefined behavior while performing checks. Apparently, avoiding such behavior is indeed a tricky business.

Type safety The aspect of C's type system that required the most care was *qualifier safety*: ensuring that `const` and `volatile` qualifiers attached to pointers at various levels of indirection are not removed by implicit casts. Accessing a const- or volatile-qualified object through a non-qualified pointer results in undefined behavior.

Pointer safety Null-pointer dereferences are easy to avoid using dynamic checks. There is, on the other hand, no portable run-time method for detecting references to a function-scoped variable whose lifetime has ended. (Hacks involving the stack pointer are not robust under inlining.) Although there are obvious ways to structurally avoid this problem, such as using a type system to ensure that a pointer to a function-scoped variable never outlives the function, we judged this kind of strategy to be too restrictive. Instead, Csmith freely permits pointers to local variables to escape (e.g., into global variables) but uses a whole-program pointer analysis to ensure that such pointers are not dereferenced or used in comparisons once they become invalid.

Csmith's pointer analysis is flow sensitive, field sensitive, context sensitive, path insensitive, and array-element insensitive. A points-to fact is an explicit set of locations that may be referenced, and may include two special elements: the null pointer and the invalid (out-of-scope) pointer. Points-to sets containing a single element serve as must-alias facts unless the pointed-to object is an array element. Because Csmith does not generate programs that use the heap, assigning names to storage locations is trivial.

Effect safety The C99 standard states that "[t]he order of evaluation of the function designator, the actual arguments, and subexpressions within the actual arguments is unspecified." Also, undefined behavior occurs if "[b]etween two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored."

To avoid these problems, Csmith uses its pointer analysis to perform a conservative interprocedural analysis and determine the *effect* of every expression, statement, and function that it generates. An effect consists of two sets: locations that may be read and locations that may be written. Csmith ensures that no location is both read and written, or written more than once, between any pair of sequence points. As a special case, in an assignment, a location can be read on the RHS and also written on the LHS.

Effects are computed, and effect safety guaranteed, incrementally. At each sequence point, Csmith resets the current effect (i.e., may-read and may-write sets). As fragments of code are generated, Csmith tests if the new code has a read/write or write/write conflict with the current effect. If a conflict is detected, the new code is thrown away and the process restarts. For example, if Csmith is generating an expression `p + func()` and it happens that `func` may modify `p`, the call to `func` is discarded and a new subexpression is generated. If there is no conflict, the read and write sets are updated and the process continues. Probabilistic progress is guaranteed: by design, Csmith always has a non-zero chance of generating code that introduces no new conflicts, such as a constant expression.

Array safety Csmith uses several methods to ensure that array indices are in bounds. First, it generates index variables that are modified only in the "increment" parts of `for` loops and whose values never exceed the bounds of the arrays being indexed. Second, variables with arbitrary value are forced to be in bounds using the modulo operator. Finally, as needed, Csmith emits explicit checks against array lengths.

Initializer safety A C program must not use an uninitialized function-scoped variable. For the most part, initializer safety is easy to ensure structurally by initializing variables close to where they are declared. Gotos introduce the possibility that initializers may be jumped over; Csmith solves this by forbidding gotos from spanning initialization code.

2.5 Efficient Global Safety

Csmith never commits to a code fragment unless it has been shown to be safe. However, loops and function calls threaten to invalidate previously validated code. For example, consider the following code, in which Csmith has just added the loop back-edge at line 7.

```
1   int i;
2   int *p = &i;
3   while (...) {
4     *p = 3;
5     ...
6     p = 0;
7   }
```

The assignment through `p` at line 4 was safe when it was generated. However, the newly added line 7 makes line 4 unsafe, due to the back-edge carrying a null-valued `p`.

One solution to this problem is to be conservative: run the whole-program dataflow analysis before committing any new statement to the program. This is not efficient. We therefore restrict the analysis to local scope except when function calls and loops are involved. For a function call, the callee is re-analyzed at each call site immediately.

Csmith uses a different strategy for loops. This is because so many statements are inside loops, and the extra calls to the dataflow analysis add substantial overhead to the code generator. Csmith's strategy is to optimistically generate code that is *locally safe*. Local safety includes running a single step of the dataflow engine (which reaches a sound result when generating code not inside any loop).

The global fixpoint analysis is run when a loop is closed by adding its back-edge. If Csmith finds that the program contains unsafe statements, it deletes code starting from the tail of the loop until the program becomes globally safe. This strategy is about three times faster than pessimistically running the global dataflow analysis before adding every piece of code.

2.6 Design Trade-offs

Allow implementation-defined behavior An ideally portable test program would be "strictly conforming" to the C language standard. This means that the program's output would be independent of all unspecified and unspecified behaviors and, in addition, be independent of any *implementation-defined behavior*. C99 has 114 kinds of implementation-defined behavior, and they have pervasive impact on the behavior of real C programs. For example, the result of performing a bitwise operation on a signed integer is implementation-defined, and operands to arithmetic operations are implicitly cast to `int` (which has implementation-defined width) before performing the operation. We believe it is impossible to generate realistically expressive C code that retains a single interpretation across all possible choices of implementation-defined behaviors.

Programs generated by Csmith do not generate the same output across compilers that differ in (1) the width and representation of integers, (2) behavior when casting to a signed integer type when the value cannot be represented in an object of the target type, and (3) the results of bitwise operations on signed integers. In practice there is not much diversity in how C implementations define these behaviors. For mainstream desktop and embedded targets, there are roughly three equivalence classes of compiler targets: those where `int` is 32 bits and `long` is 64 bits (e.g., x86-64), those where `int` and `long` are 32 bits (e.g., x86, ARM, and PowerPC), and those where `int` is 16 bits and `long` is 32 bits (e.g., MSP430 and AVR). Using Csmith, we can perform differential testing within an equivalence class but not across classes.

No ground truth Csmith's programs are not self-checking: we are unable to predict their outputs without running them. This is not a problem when we use Csmith for randomized differential testing.

We have never seen an "interesting" split vote where randomized differential testing of a collection of C compilers fails to produce a clear consensus answer, nor have we seen any cases in which a majority of tested compilers produces the same incorrect result. (We would catch the problem by hand as part of verifying the failure-inducing program.) In fact, we have not seen even two unrelated compilers produce the same incorrect output for a Csmith-generated test case. It therefore seems unlikely that all compilers under test would produce the same incorrect output for a test case. Of course, if that did happen we would not detect that problem; this is an inherent limitation of differential testing without an oracle. In summary, despite the fact that Knight and Leveson [13] found a substantial number of correlated errors in an experiment on N-version programming, Csmith has yielded no evidence of correlated failures among unrelated C compilers. Our hypothesis is that the observed lack of correlation stems from the fact that most compiler bugs are in passes that operate on an intermediate representation and there is substantial diversity among IRs.

No guarantee of termination It is not difficult to generate random programs that always terminate. However, we judged that this would limit Csmith's expressiveness too much: for example, it would force loops to be highly structured. Additionally, always-terminating tests cannot find compiler bugs that wrongfully terminate a non-terminating program. (We have found bugs of this kind.) About 10% of the programs generated by Csmith are (apparently) non-terminating. In practice, during testing, they are easy to deal with using timeouts.

Target middle-end bugs Commercial test suites for C compilers [1, 19, 20] are primarily aimed at checking standards conformance. Csmith, on the other hand, is mainly intended to find bugs in the parts of a compiler that perform transformations on an intermediate representation—the so-called "middle end" of a compiler. As a result, we have found large numbers of middle-end bugs missed by existing testing techniques (Section 3.6). At the same time, Csmith is rather poor at finding gaps in standards conformance. For example, it makes no attempt to test a compiler's handling of trigraphs, long identifier names, or variadic functions.

Targeting the middle end has several aspects. First, all generated programs pass the lexer, parser, and typechecker. Second, we performed substantial manual tuning of the 80 probabilities that govern Csmith's random choices. Our goal was to make the generated programs "look right"—to contain a balanced mix of arithmetic and bitwise operations, of references to scalars and aggregates, of loops and straight-line code, of single-level and multi-level indirections, and so on. Third, Csmith specifically generates idiomatic code (e.g., loops that access all elements of an array) to stress-test parts of the compiler we believe to be error-prone. Fourth, we designed Csmith with an eye toward generating programs that exercise the constructs of a compiler's intermediate representation, and we decided to avoid generating source-level diversity that is unlikely to improve the "coverage" of a compiler's intermediate representations. For example, since additional levels of parentheses around expressions are stripped away early in the compilation process, we do not generate them, nor do we generate all of C's syntactic loop forms since they are typically all lowered to the same IR constructs. Finally, Csmith was designed to be fast enough that it can generate programs that are a few tens of thousands of lines long in a few seconds. Large programs are preferred because (empirically—see Section 3.3) they find more bugs. In summary, many aspects of Csmith's design and implementation were informed by our understanding of how modern compilers work and how they break.

3. Results

We conducted five experiments using Csmith, our random program generator. This section summarizes our findings.

Our first experiment was uncontrolled and unstructured: over a three-year period, we opportunistically found and reported bugs in a variety of C compilers. We found bugs in all the compilers we tested—hundreds of defects, many classified as high-priority bugs. (§3.1)

In the second experiment, we compiled and ran one million random programs using several years' worth of versions of GCC and LLVM, to understand how their robustness is evolving over time. As measured by our tests over the programs that Csmith produces, the quality of both compilers is generally improving. (§3.2)

Third, we evaluated Csmith's bug-finding power as a function of the size of the generated C programs. The largest number of bugs is found at a surprisingly large program size: about 81 KB. (§3.3)

Fourth, we compared Csmith's bug-finding power to that of four previous random C program generators. Over a week, Csmith was able to find significantly more distinct compiler crash errors than previous program generators could. (§3.4)

Finally, we investigated the effect of testing random programs on branch, function, and line coverage of the GCC and LLVM source code. We found that these metrics did not significantly improve when we added randomly generated programs to the compilers' existing test suites. Nevertheless, as shown by our other results, Csmith-generated programs allowed us to discover bugs that are missed by the compilers' standard test suites. (§3.5)

We conclude the presentation of results by analyzing some of the bugs we found in GCC and LLVM. (§3.6, §3.7)

	GCC	LLVM
Crash	2	10
Wrong code	2	9
Total	4	19

Table 2. Crash and wrong-code bugs found by Csmith that manifest when compiler optimizations are disabled (i.e., when the –O0 command-line option is used)

3.1 Opportunistic Bug Finding

We reported bugs to 11 different C compiler development teams. Five of these compilers (GCC, LLVM, CIL, TCC, and Open64) were open source and five were commercial products. The eleventh, CompCert, is publicly available but not open source.

What kinds of bugs are there? It is useful to distinguish between errors whose symptoms manifest at compile time and those that only manifest when the compiler's output is executed. Compile-time bugs that we see include assertion violations or other internal compiler errors; involuntary compiler termination due to memory-safety problems; and cases in which the compiler exhausts the RAM or CPU time allocated to it. We say that a compile-time *crash error* has occurred whenever the compiler process exits with a status other than zero or fails to produce executable output. Errors that manifest at run time include the computation of a wrong result; a crash or other abnormal termination of the generated code; termination of a program that should have executed forever; and non-termination of a program that should have terminated. We refer to these run-time problems as *wrong-code errors*. A *silent wrong-code error* is one that occurs in a program that was produced without any sort of warning from the compiler; i.e., the compiler silently miscompiled the test program.

Experience with commercial compilers There exist many more commercial C compilers than we could easily test. The ones we chose to study are fairly popular and were produced by what we believe are some of the strongest C compiler development teams. Csmith found wrong-code errors and crash errors in each of these tools within a few hours of testing.

Because we are not paying customers, and because our findings represent potential bad publicity, we did not receive a warm response from any commercial compiler vendor. Thus, for the most part, we simply tested these compilers until we found a few crash errors and a few wrong-code errors, reported them, and moved on.

Experience with open-source compilers For several reasons, the bulk of our testing effort went towards GCC and LLVM. First and most important, compiler testing is inherently interactive: we require feedback from the development team in the form of bug fixes. Bugs that occur with high probability can mask tricky, one-in-a-million bugs; thus, testing proceeds most smoothly when we can help developers rapidly destroy the easy bugs. Both the GCC and LLVM teams were responsive to our bug reports. The LLVM team in particular fixed bugs quickly, often within a few hours and usually within a week. The second reason we prefer dealing with open-source compilers is that their development process is transparent: we can watch the mailing lists, participate in discussions, and see fixes as they are committed. Third, we want to help harden the open-source development tools that we and many others use daily.

So far we have reported 79 GCC bugs and 202 LLVM bugs—the latter figure represents about 2% of all LLVM bug reports. Most of our reported bugs have been fixed, and twenty-five of the GCC bugs were marked by developers as P1: the maximum, release-blocking priority for a bug. To date, we have reported 325 in total across all tested compilers (GCC, LLVM, and others).

An error that occurs at the lowest level of optimization is pernicious because it defeats the conventional wisdom that compiler bugs can be avoided by turning off the optimizer. Table 2 counts these kinds of bugs, causing both crash and wrong-code errors, that we found using Csmith.

Testing CompCert CompCert [14] is a verified, optimizing compiler for a large subset of C; it targets PowerPC, ARM, and x86. We put significant effort into testing this compiler.

The first silent wrong-code error that we found in CompCert was due to a miscompilation of this function:

```
1  int bar (unsigned x) {
2    return -1 <= (1 && x);
3  }
```

CompCert 1.6 for PowerPC generates code returning 0, but the proper result is 1 because the comparison is signed. This bug and five others like it were in CompCert's unverified front-end code. Partly in response to these bug reports, the main CompCert developer expanded the verified portion of CompCert to include C's integer promotions and other tricky implicit casts.

The second CompCert problem we found was illustrated by two bugs that resulted in generation of code like this:

```
stwu r1, -44432(r1)
```

Here, a large PowerPC stack frame is being allocated. The problem is that the 16-bit displacement field is overflowed. CompCert's PPC semantics failed to specify a constraint on the width of this immediate value, on the assumption that the assembler would catch out-of-range values. In fact, this is what happened. We also found a handful of crash errors in CompCert.

The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.

3.2 Quantitative Comparison of GCC and LLVM Versions

Figure 3 shows the results of an experiment in which we compiled and ran 1,000,000 randomly generated programs using LLVM 1.9–2.8, GCC 3.[0–4].0, and GCC 4.[0–5].0. Every program was compiled at –O0, –O1, –O2, –Os, and –O3. A test case was considered valid if every compiler terminated (successfully or otherwise) within five minutes and if every compiled random program terminated (correctly or otherwise) within five seconds. All compilers targeted x86. Running these tests took about 1.5 weeks on 20 machines in the Utah Emulab testbed [28]. Each machine had one quad-core Intel Xeon E5530 processor running at 2.4 GHz.

Compile-time failures The top row of graphs in Figure 3 shows the observed rate of crash errors. (Note that the y-axes of these graphs are logarithmic.) These graphs also indicate the number of crash bugs that were fixed in response to our bug reports. Both compilers became at least three orders of magnitude less "crashy" over the range of versions covered in this experiment. The GCC results appear to tell a nice story: the 3.x release series increases in quality, the 4.0.0 release regresses because it represents a major change to GCC's internals, and then quality again starts to improve.

The middle row of graphs in Figure 3 shows the number of *distinct assertion failures* in LLVM and the number of *distinct internal compiler errors* in GCC induced by our tests. These are the numbers of code locations in LLVM and GCC at which an internal

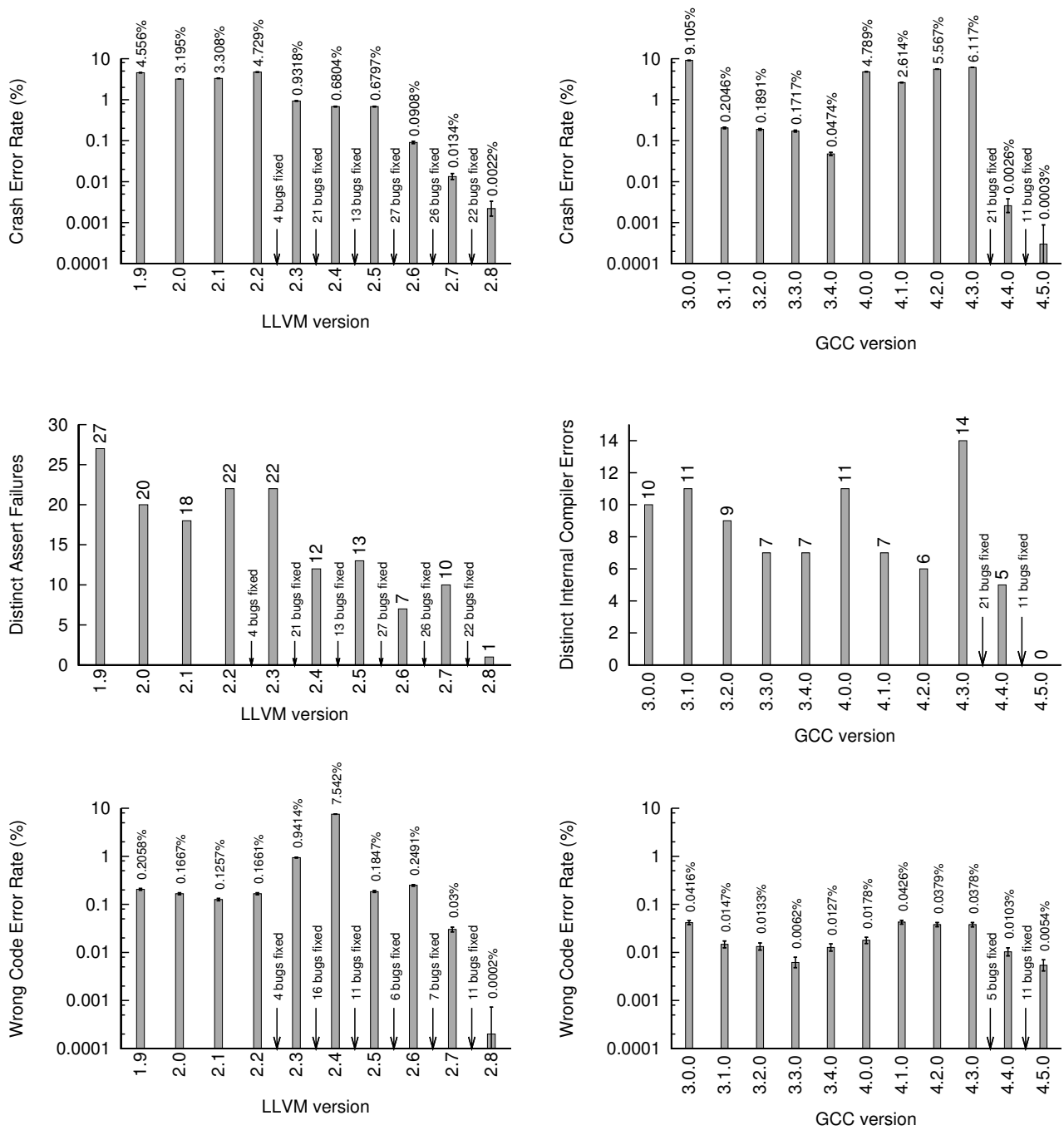


Figure 3. Distinct crash errors found, and rates of crash and wrong-code errors, from recent LLVM and GCC versions

consistency check failed. These graphs conservatively estimate the number of distinct failures in these compilers, since we encountered many segmentation faults caused by use of free memory, null-pointer dereferences, and similar problems. We did not include these faults in our graphed results due to the difficulty of mapping crashes back to distinct causes.

It is not clear which of these two metrics of crashiness is preferable. The rate of crashes is easy to game: we can make it arbitrarily high by biasing Csmith to generate code triggering known bugs, and compiler writers can reduce it to zero by eliminating error messages and always returning a "success" status code to the operating system. The number of distinct crashes, on the other hand, suffers from the drawback that it depends on the quantity and style of assertions in the compiler under test. Although GCC has more total assertions than LLVM, LLVM has a higher density: about one assertion per 100 lines of code, compared to one in 250 for GCC.

Run-time failures The bottom pair of graphs in Figure 3 shows the rate of wrong-code errors in our experiment. Unfortunately, we

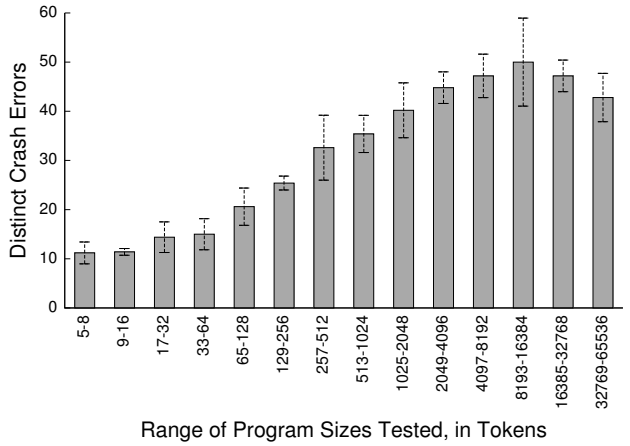


Figure 4. Number of distinct crash errors found in 24 hours of testing with Csmith-generated programs in a given size range

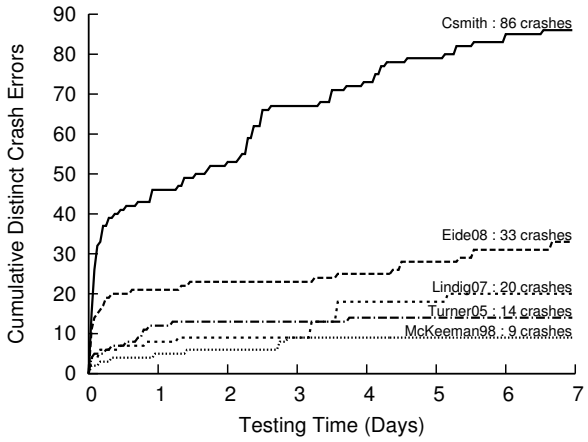


Figure 5. Comparison of the ability of five random program generators to find distinct crash errors

can only report the rate of errors, and not the number of bugs causing them, because we do not know how to automatically map failing tests back to the bugs that cause them. These graphs also indicate the number of wrong-code bugs that were fixed in response to our bug reports.

3.3 Bug-Finding Performance as a Function of Test-Case Size

There are many ways in which a random test-case generator might be "tuned" for particular goals, e.g., to focus on certain kinds of compiler defects. We performed an experiment to answer this question: given the goal of finding many defects quickly, should one configure Csmith to generate small programs or large ones? Other factors being equal, small test cases are preferable because they are closer to being reportable to compiler developers.

Using the same compilers and optimization options that we used for the experiments in Section 3.2, we ran our testing process multiple times. For each run we selected a size range for test inputs, configured Csmith to generate programs in that range,[3] executed the test process for 24 hours, and counted the distinct crash errors found. We repeated this for various ranges of test-input sizes.

Figure 4 shows that the rate of crash-error detection varies significantly as a function of the sizes of the test programs produced by Csmith. The greatest number of distinct crash errors is found by programs containing 8 K–16 K tokens: these programs averaged 81 KB before preprocessing. The confidence intervals are at 95% and were computed based on five repetitions.

We hypothesize that larger test cases expose more compiler errors for two reasons. First, throughput is increased because compiler start-up costs are better amortized. Second, the combinatorial explosion of feature interactions within a single large test case works in Csmith's favor. The decrease in bug-finding power at the largest sizes appears to come from algorithms—in Csmith and in the compilers—that have superlinear running time.

3.4 Bug-Finding Performance Compared to Other Tools

To evaluate Csmith's ability to find bugs, we compared it to four other random program generators: the two versions of Randprog described in Section 2 and two others described in Section 5. We ran each generator in its default configuration on one of five identical and otherwise-idle machines, using one CPU on each host. Each generator repeatedly produced programs that we compiled and tested using the same compilers and optimization options that were used for the experiments in Section 3.2. Figure 5 plots the cumulative number of distinct crash errors found by these program generators during the one-week test. Csmith significantly outperforms the other tools.

[3] Although we can tune Csmith to prefer generating larger or smaller output, it lacks the ability to construct a test case of a specific size on demand. We ran this experiment by precomputing seeds to Csmith's random-number generator that cause it to generate programs of the sizes we desired.

		Line Coverage	Function Coverage	Branch Coverage
GCC	make check-c	75.13%	82.23%	46.26%
	make check-c & random	75.58%	82.41%	47.11%
	% change	+0.45%	+0.13%	+0.85%
	absolute change	+1,482	+33	+4,471
Clang	make test	74.54%	72.90%	59.22%
	make test & random	74.69%	72.95%	59.48%
	% change	+0.15%	+0.05%	+0.26%
	absolute change	+655	+74	+926

Table 3. Augmenting the GCC and LLVM test suites with 10,000 randomly generated programs did not improve code coverage much

3.5 Code Coverage

Because we find many bugs, we hypothesized that randomly generated programs exercise large parts of the compilers that were not covered by existing test suites. To test this, we enabled code-coverage monitoring in GCC and LLVM. We then used each compiler to build its own test suite, and also to build its test suite plus 10,000 Csmith-generated programs. Table 3 shows that the incremental coverage due to Csmith is so small as to be a negative result. Our best guess is that these metrics are too shallow to capture Csmith's effects, and that we would generate useful additional coverage in terms of deeper metrics such as path or value coverage.

3.6 Where Are the Bugs?

Table 4 characterizes the GCC and LLVM bugs we found by compiler part. Tables 5 and 6 show the ten buggiest files in LLVM and GCC as measured by our experiment in Section 3.1. Most of the bugs we found in GCC were in the middle end: the machine-independent optimizers. LLVM is a younger compiler and our testing shook out some front-end and back-end bugs that would probably not be present in a more mature software base.

	GCC	LLVM
Front end	0	10
Middle end	49	75
Back end	17	74
Unclassified	13	43
Total	79	202

Table 4. Distribution of bugs across compiler stages. A bug is unclassified either because it has not yet been fixed or the developer who fixed the bug did not indicate what files were changed.

C File Name	Purpose	Wrong-Code Bugs	Crash Bugs
fold-const	constant folding	3	6
combine	instruction combining	1	5
tree-ssa-pre	partial redundancy elim.	0	4
tree-vrp	variable range propagation	0	4
tree-ssa-dce	dead code elimination	0	3
tree-ssa-reassoc	arithmetic expr. reassociation	0	2
reload1	register reloading	1	1
tree-ssa-loop-niter	loop iteration counting	1	1
dse	dead store elimination	2	0
tree-scalar-evolution	scalar evolution	2	0
Other (15 files)	n/a	19	24
Total (25 files)	n/a	29	50

Table 5. Top ten buggy files in GCC

C++ File Name	Purpose	Wrong-Code Bugs	Crash Bugs
Instruction-Combining	mid-level instruction combining	9	6
SimpleRegister-Coalescing	register coalescing	1	10
DAGCombiner	instruction combining	5	3
LoopUnswitch	loop unswitching	1	4
LICM	loop invariant code motion	0	5
LoopStrength-Reduce	loop strength reduction	1	3
FastISel	fast instruction selection	1	3
llvm-convert	GCC-LLVM IR conversion	0	4
ExprConstant	constant folding	2	2
JumpThreading	jump threading	0	4
Other (72 files)	n/a	46	92
Total (82 files)	n/a	66	136

Table 6. Top ten buggy files in LLVM

3.7 Examples of Wrong-Code Bugs

This section characterizes a few of the bugs that were revealed by miscompilation of programs generated by Csmith. These bugs fit into a simple model in which optimizations are structured like this:

```
analysis
if (safety check) {
  transformation
}
```

An optimization can fail to be semantics-preserving if the analysis is wrong, if the safety check is insufficiently conservative, or if the transformation is incorrect. The most common root cause for bugs that we found was an incorrect safety check.

GCC Bug #1: wrong safety check[4] If `x` is variable and `c1` and `c2` are constants, the expression `(x/c1)!=c2` can be profitably rewritten as `(x-(c1*c2))>(c1-1)`, using unsigned arithmetic to avoid problems with negative values. Prior to performing the transformation, expressions such as `c1*c2` and `(c1*c2)+(c1-1)` are checked for overflow. If overflow occurs, further simplifications can be made; for example, `(x/1000000000)!=10` always evaluates to 0 when `x` is a 32-bit integer. GCC falsely detected overflow for some choices of constants. In the failure-inducing test case that we discovered, `(x/-1)!=1` was folded to 0. This expression should evaluate to 1 for many values of `x`, such as 0.

GCC Bug #2: wrong transformation[5] In C, if an argument of type `unsigned char` is passed to a function with a parameter of type `int`, the values seen inside the function should be in the range 0..255. We found a case in which a version of GCC inlined this kind of function call and then sign-extended the argument rather than zero-extending it, causing the function to see negative values of the parameter when the function was called with arguments in the range 128..255.

GCC Bug #3: wrong analysis[6] We found a bug that caused GCC to miscompile this code:

```
1 static int g[1];
2 static int *p = &g[0];
3 static int *q = &g[0];
4
5 int foo (void) {
6   g[0] = 1;
7   *p = 0;
8   *p = *q;
9   return g[0];
10 }
```

The generated code returned 1 instead of 0. The problem occurred when the compiler failed to recognize that `p` and `q` are aliases; this happened because `q` was mistakenly identified as a read-only memory location, which is defined not to alias a mutable location. The wrong not-alias fact caused the store in line 7 to be marked as dead so that a subsequent dead-store elimination pass removed it.

GCC Bug #4: wrong analysis[7] A version of GCC miscompiled this function:

```
1 int x = 4;
2 int y;
3
4 void foo (void) {
5   for (y = 1; y < 8; y += 7) {
6     int *p = &y;
7     *p = x;
8   }
9 }
```

When `foo` returns, `y` should be 11. A loop-optimization pass determined that a temporary variable representing `*p` was invariant with value `x+7` and hoisted it in front of the loop, while retaining a dataflow fact indicating that `x+7 == y+7`, a relationship that no longer held after code motion. This incorrect fact lead GCC to generate code leaving 8 in `y`, instead of 11.

[4] `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=42721`

[5] `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=43438`

[6] `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=42952`

[7] `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=43360`

LLVM Bug #1: wrong safety check[8] `(x==c1)||(x<c2)` can be simplified to `x<c2` when `c1` and `c2` are constants and `c1<c2`. An LLVM version incorrectly transformed `(x==0)||(x<-3)` to `x<-3`. LLVM did a comparison between 0 and -3 in the safety check for this optimization, but performed an unsigned comparison rather than a signed one, leading it to incorrectly determine that the transformation was safe.

LLVM Bug #2: wrong safety check[9] `(x|c1)==c2` evaluates to 0 if `c1` and `c2` are constants and `(c1&~c2)!=0`. In other words, if any bit that is set in `c1` is unset in `c2`, the original expression cannot be true. A version of LLVM contained a logic error in the safety check for this optimization, wrongly replacing this kind of expression with 0 even when `c1` was not a constant.

LLVM Bug #3: wrong safety check[10] "Narrowing" is a strength-reduction optimization that can be applied to loads when only part of an object is needed, or to stores where only part of an object is modified. For example, at the level of the abstract machine this code loads and stores an `unsigned int`:

```
1 unsigned y;
2
3 void bar (void) {
4   y |= 255;
5 }
```

Optimizing compilers for x86 may translate `bar` into the following code, which loads nothing and stores a single byte:

```
bar:
  movb $-1, y
  ret
```

We found a case in which LLVM attempted to perform an analogous narrowing operation, but a logic error caused the safety check to succeed even when a different store modified the object prior to the store that was the target of the narrowing transformation.

LLVM Bug #4: wrong analysis[11] This code should print "5":

```
1 void foo (void) {
2   int x;
3   for (x = 0; x < 5; x++) {
4       if (x) continue;
5       if (x) break;
6   }
7   printf("%d", x);
8 }
```

LLVM's scalar evolution analysis computes properties of loop induction variables, including the maximum number of iterations. Line 5 of the program above caused this analysis to mistakenly conclude that `x` was 1 after the loop executed.

4. Discussion

Are we finding bugs that matter? One might suspect that random testing finds bugs that do not matter in practice. Undoubtedly this happens sometimes, but in a number of instances we have direct confirmation that Csmith is finding bugs that matter, because bugs that we have found and reported have been independently rediscovered and re-reported by application developers. By a very conservative estimate—counting only the times that a compiler developer explicitly labeled a wrong-code bug report as a duplicate of one of ours—this has happened eight times: four times for GCC and four for LLVM. We also have indirect confirmation that our bugs matter. The developers of open-source compilers fixed almost all of the bugs that we reported, and the GCC development team marked 25 of our bugs as P1: the maximum, release-blocking priority.

Creating reportable bugs Reporting compiler crash bugs is easy, but reporting wrong-code bugs is harder. Compiler developers will (rightfully) ignore a wrong-code bug report that is based on a large random program. Rather, a bug report must be accompanied by compelling evidence that a bug exists; in most cases the best evidence is a small test case that is obviously miscompiled. Delta debugging [31] automates test-case reduction, but all existing variants that are intended for reducing C programs—such as hierarchical delta debugging [18] and Wilkerson's implementation [29]—introduce undefined behavior. The resulting programs are small but useless. To avoid undefined behavior during reduction, we rely on compiler warnings, dynamic checkers, and manual test-case reduction. There is substantial room for improvement.

The relationship between testing and verification As our CompCert results make plain, verification does not obviate testing, but rather complements it. Testing can provide end-to-end evidence that numerous paths through a system work properly. Verification, on the other hand, typically focuses on a narrow slice of a stack of tools, and the parts outside the slice remain in the trusted computing base. There does not yet appear to be a nuanced understanding of the kinds of testing, and the amount of testing effort, that are rendered unnecessary by artifacts like CompCert [14] and seL4 [12].

Toward realistic, correct compilers Compilers must support rapid development to cope with new optimizations, new source languages, and new target architectures. Generated code often needs to be resource-efficient to support application developers' goals. Finally, compilers should generate correct code. Meeting even two of these goals is challenging, and it is not clear how to meet all three in a single tool. There seem to be four paths forward.

Compiler verification. Although it is difficult to imagine a verified compiler for C++0x, due to the immense complexity of the draft standard, CompCert is an existence proof that a verified, optimizing C compiler is within reach. However, the burden of verification is significant. CompCert still lacks a number of useful C features and few mainstream compiler developers have the formal verification skills that are needed to add new language features and optimization passes. On the other hand, projects such as XCERT [26] may dramatically lower the bar for working on verified compilation.

Compiler simplicity. For non-bottleneck applications, compiler optimization adds little end-user value. It would seem possible to take a simple compiler such as TCC [2], which does not optimize across statement boundaries, and validate it through code inspections, heavy use, and other techniques. At present, however, TCC is much buggier than more heavily-used compilers such as GCC and LLVM.

Compiler testing. We hypothesize that it is possible to gain high confidence in a complex compiler like GCC by choosing a fixed configuration, disabling optimization passes whose effects are significantly non-local, and performing "just enough testing." A test plan would be sufficient if all code paths through the compiler that are used to compile an application of interest had been tested. Clearly, a sophisticated way to abstract over paths is needed.

Equivalence checking. If equivalence checkers for machine code [7] could scale to large programs, verified compilers would be largely unnecessary because one compiler's output could be proved equivalent to another's. Although these tools are not likely to scale up to multi-megabyte applications anytime soon, it should

[8] `http://llvm.org/bugs/show_bug.cgi?id=2844`

[9] `http://llvm.org/bugs/show_bug.cgi?id=7750`

[10] `http://llvm.org/bugs/show_bug.cgi?id=7833`

[11] `http://llvm.org/bugs/show_bug.cgi?id=7845`

be possible to automatically partition applications into smaller parts so that equivalence checking can be done piecewise.

Future work Augmenting Csmith with white-box testing techniques, where the structure of the tested system is taken into account in a first-class way, would be productive. This will be difficult for several reasons. First, we anticipate substantial challenges in integrating the necessary constraint-solving machinery with Csmith's existing logic for generating valid C programs. It is possible that we will need to start over, next time engineering a version of Csmith in which all constraints are explicit and declarative, rather than being buried in a small mountain of C++. Second, the inverse problems that must be solved to generate an input become prohibitively difficult when inputs pass through a parser, particularly if the parser contains hash tables. Godefroid et al. [8] showed a way to solve this problem by integrating a constraint solver with a grammar for the language being generated. However, due to its non-local pointer and effect analyses, the validity decision problem for programs in the subset of C that Csmith generates is far harder than the question of whether a program can be generated by the JavaScript grammar used by Godefroid et al.

5. Related Work

Compilers have been tested using randomized methods for nearly 50 years. Boujarwah and Saleh [4] gave a good survey in 1997. In 1962, Sauder [22] tested the correctness of COBOL compilers by placing random variables in programs' data sections. In 1970, Hanford [10] used a PL/1 grammar to drive the generation of random programs. The grammar was extensible and was augmented by "syntax generators" that could be used, for example, to ensure that variables were declared before being used. In 1972, Purdom [21] used a syntax-directed method to generate test sentences for a parser. He gave an efficient algorithm for generating short sentences from a context-free grammar such that each production of the grammar was used at least once, and he tested LR(1) parsers using this technique.

Burgess and Saidi [5] designed an automatic generator of test cases for FORTRAN compilers. The tests were designed to be self-checking and to contain features that optimizing compilers were known to exploit. In order to predict test cases' results, the code generator restricted assignment statements to be executed only once during the execution of the sub-program or main program. These tests found four bugs in two FORTRAN 77 compilers.

In 1998, McKeeman [16] coined the term "differential testing." His work resulted in DDT, a family of program generators that conform to the C standard at various levels, from level 1 (random characters) to level 7 (generated code is "model conforming," incorporating some high-level structure). DDT is more expressive than Csmith (DDT is capable of generating all legal C programs) and it was used to find numerous bugs in C compilers. To our knowledge, McKeeman's paper contains the first acknowledgment that it is important to avoid undefined behavior in generated C programs used for compiler testing. However, DDT avoided only a small subset of all undefined behaviors, and only then during test-case reduction, not during normal testing. Thus, it is not a suitable basis for automatic bug-finding.

Lindig [15] used randomly generated C programs to find several compiler bugs related to calling conventions. His tool, called Quest, was specially targeted: rather than generating code with control flow and arithmetic, Quest generates code that creates complex data structures, loads them with constant values, and passes them to a function where assertions check the received values. Because its tests are self-checking, Quest is not based on differential testing. Self-checking tests are convenient, but the drawback is that Quest is far less expressive than Csmith. Lindig used Quest to test GCC, LCC, ICC, and a few other compilers and found 13 bugs.

Sheridan [23] also used a random generator to find bugs in C compilers. A script rotated through a list of constants of the principal arithmetic types, producing a source file that applied various operators to pairs of constants. This tool found two bugs in GCC, one bug in SUSE Linux's version of GCC, and five bugs in CodeSourcery's version of GCC for ARM. Sheridan's tool produces self-checking tests. However, it is less expressive than Csmith and it fails to avoid undefined behavior such as signed overflow.

Zhao et al. [32] created an automated program generator for testing an embedded C++ compiler. Their tool allows a general test requirement, such as which optimization to test, to be specified in a script. The generator constructs a program template based on the test requirement and uses it to drive further code generation. Zhao et al. used GCC as the reference to check the compiler under test. They reported greatly improved statement coverage in the tested modules and found several new compiler bugs.

6. Conclusion

Using randomized differential testing, we found and reported hundreds of previously unknown bugs in widely used C compilers, both commercial and open source. Many of the bugs we found cause a compiler to emit incorrect code without any warning. Most of our reported defects have been fixed, meaning that compiler implementers found them important enough to track down, and 25 of the bugs we reported against GCC were classified as release-blocking. All of this evidence suggests that there is substantial room for improvement in the state of the art for compiler quality assurance.

To create a random program generator with high bug-finding power, the key problem we solved was the expressive generation of C programs that are free of undefined behavior and independent of unspecified behavior. Csmith, our program generator, uses both static analysis and dynamic checks to avoid these hazards.

The return on investment from random testing is good. Our rough estimate—including faculty, staff, and student salaries, machines purchased, and university overhead—is that each of the more than 325 bugs we reported cost less than $1,000 to find. The incremental cost of a new bug that we find today is much lower.

Software Csmith is open source and available for download at `http://embed.cs.utah.edu/csmith/`.

Acknowledgments

The authors would like to thank Bruce Childers, David Coppit, Chucky Ellison, Robby Findler, David Gay, Casey Klein, Gerwin Klein, Chris Lattner, Sorin Lerner, Xavier Leroy, Bill McKeeman, Diego Novillo, Alastair Reid, Julian Seward, Zach Tatlock, our shepherd Atanas Rountev, and the anonymous reviewers for their invaluable feedback on drafts of this paper. We also thank Hans Boehm, Xavier Leroy, Michael Norrish, Bryan Turner, and the GCC and LLVM development teams for their technical assistance in various aspects of our work.

This research was primarily supported by an award from DARPA's Computer Science Study Group.

References

[1] ACE Associated Computer Experts. SuperTest C/C++ compiler test and validation suite. `http://www.ace.nl/compiler/supertest.html`.

[2] F. Bellard. TCC: Tiny C compiler, ver. 0.9.25, May 2009. `http://bellard.org/tcc/`.

[3] C. L. Biffle. Undefined behavior in Google NaCl, Jan. 2010. `http://code.google.com/p/nativeclient/issues/detail?id=245`.

[4] A. S. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information and Software Technology*, 39(9):617–625, 1997.

[5] C. J. Burgess and M. Saidi. The automatic generation of test cases for optimizing Fortran compilers. *Information and Software Technology*, 38(2):111–119, 1996.

[6] E. Eide and J. Regehr. Volatiles are miscompiled, and what to do about it. In *Proc. EMSOFT*, pages 255–264, Oct. 2008.

[7] X. Feng and A. J. Hu. Cutpoints for formal equivalence verification of embedded software. In *Proc. EMSOFT*, pages 307–316, Sept. 2005.

[8] P. Godefroid, A. Kieżun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proc. PLDI*, pages 206–215, June 2008.

[9] R. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley, second edition, 2001.

[10] K. V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, Dec. 1970.

[11] International Organization for Standardization. *ISO/IEC 9899:TC2: Programming Languages—C*, May 2005. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`.

[12] G. Klein et al. seL4: Formal verification of an OS kernel. In *Proc. SOSP*, pages 207–220, Oct. 2009.

[13] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Software Eng.*, 12(1):96–109, Jan. 1986.

[14] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[15] C. Lindig. Random testing of C calling conventions. In *Proc. AADEBUG*, pages 3–12, Sept. 2005.

[16] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, Dec. 1998.

[17] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[18] G. Misherghi and Z. Su. HDD: Hierarchical delta debugging. In *Proc. ICSE*, pages 142–151, May 2006.

[19] Perennial, Inc. ACVS ANSI/ISO/FIPS-160 C validation suite, ver. 4.5, Jan. 1998. `http://www.peren.com/pages/acvs_set.htm`.

[20] Plum Hall, Inc. The Plum Hall validation suite for C. `http://www.plumhall.com/stec.html`.

[21] P. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[22] R. L. Sauder. A general test data generator for COBOL. In *AFIPS Joint Computer Conferences*, pages 317–323, May 1962.

[23] F. Sheridan. Practical testing of a C99 compiler using output comparison. *Software—Practice and Experience*, 37(14):1475–1488, Nov. 2007.

[24] J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal verification of avionics software products. In *Proc. FM*, pages 532–546, Nov. 2009.

[25] S. Summit. comp.lang.c frequently asked questions. `http://c-faq.com/`.

[26] Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. In *Proc. PLDI*, pages 111–121, June 2010.

[27] B. Turner. Random Program Generator, Jan. 2007. `http://sites.google.com/site/brturn2/randomcprogramgenerator`.

[28] B. White et al. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, pages 255–270, Dec. 2002.

[29] D. S. Wilkerson. Delta ver. 2006.08.03, Aug. 2006. `http://delta.tigris.org/`.

[30] M. Wolfe. How compilers and tools differ for embedded systems. In *Proc. CASES*, Sept. 2005. Keynote address. `http://www.pgroup.com/lit/articles/pgi_article_cases.pdf`.

[31] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, Feb. 2002.

[32] C. Zhao et al. Automated test program generation for an industrial optimizing compiler. In *Proc. ICSE Workshop on Automation of Software Test*, pages 36–43, May 2009.

Embedded in Academia

John Regehr, Professor of Computer Science, University of Utah, USA

A Guide to Undefined Behavior in C and C++, Part 1

Also see Part 2 and Part 3.

Programming languages typically make a distinction between normal program actions and erroneous actions. For Turing-complete languages we cannot reliably decide offline whether a program has the potential to execute an error; we have to just run it and see.

In a *safe* programming language, errors are trapped as they happen. Java, for example, is largely safe via its exception system. In an *unsafe* programming language, errors are not trapped. Rather, after executing an erroneous operation the program keeps going, but in a silently faulty way that may have observable consequences later on. Luca Cardelli's article on type systems has a nice clear introduction to these issues. C and C++ are unsafe in a strong sense: executing an erroneous operation causes the entire program to be meaningless, as opposed to just the erroneous operation having an unpredictable result. In these languages erroneous operations are said to have *undefined behavior*.

The C FAQ defines "undefined behavior" like this:

> *Anything at all can happen; the Standard imposes no requirements. The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended.*

This is a good summary. Pretty much every C and C++ programmer understands that accessing a null pointer and dividing by zero are erroneous actions. On the

other hand, the full implications of undefined behavior and its interactions with aggressive compilers are not well-appreciated. This post explores these topics.

A Model for Undefined Behavior

For now, we can ignore the existence of compilers. There is only the "C implementation" which — if the implementation conforms to the C standard — acts the same as the "C abstract machine" when executing a conforming program. The C abstract machine is a simple interpreter for C that is described in the C standard. We can use it to determine the meaning of any C program.

The execution of a program consists of simple steps such as adding two numbers or jumping to a label. If every step in the execution of a program has defined behavior, then the entire execution is well-defined. Note that even well-defined executions may not have a unique result due to unspecified and implementation-defined behavior; we'll ignore both of these here.

If any step in a program's execution has undefined behavior, then the entire execution is without meaning. This is important: it's not that evaluating (1<<32) has an unpredictable result, but rather that the entire execution of a program that evaluates this expression is meaningless. Also, it's not that the execution is meaningful up to the point where undefined behavior happens: the bad effects can actually precede the undefined operation.

As a quick example let's take this program:

```
#include <limits.h>
#include <stdio.h>

int main (void)
{
  printf ("%d\n", (INT_MAX+1) < 0);
  return 0;
}
```

The program is asking the C implementation to answer a simple question: if we add one to the largest representable integer, is the result negative? This is perfectly legal behavior for a C implementation:

```
$ cc test.c -o test
$ ./test
1
```

So is this:

```
$ cc test.c -o test
$ ./test
0
```

And this:

```
$ cc test.c -o test
$ ./test
42
```

And this:

```
$ cc test.c -o test
$ ./test
Formatting root partition, chomp chomp
```

One might say: Some of these compilers are behaving improperly because the C standard says a relational operator must return 0 or 1. But since the program has no meaning at all, the implementation can do whatever it likes. Undefined behavior trumps all other behaviors of the C abstract machine.

Will a real compiler emit code to chomp your disk? Of course not, but keep in mind that practically speaking, undefined behavior often does lead to Bad Things because many security vulnerabilities start out as memory or integer operations that have undefined behavior. For example, accessing an out of bounds array element is a key part of the canonical stack smashing attack. In summary: the compiler does not need to emit code to format your disk. Rather, following the OOB array access your computer will begin executing exploit code, and that code is what will format your disk.

No Traveling

It is very common for people to say — or at least think — something like this:

> *The x86 ADD instruction is used to implement C's signed add operation, and it has two's complement behavior when the result overflows. I'm developing for an x86 platform, so I should be able to expect two's complement semantics when 32-bit signed integers overflow.*

THIS IS WRONG. You are saying something like this:

> *Somebody once told me that in basketball you can't hold the ball and run. I got a basketball and tried it and it worked just fine. He obviously didn't understand basketball.*

(This explanation is due to Roger Miller via Steve Summit.)

Of course it is physically possible to pick up a basketball and run with it. It is also possible you will get away with it during a game. However, it is against the rules; good players won't do it and bad players won't get away with it for long. Evaluating (INT_MAX+1) in C or C++ is exactly the same: it may work sometimes, but don't expect to keep getting away with it. The situation is actually a bit subtle so let's look in more detail.

First, are there C implementations that guarantee two's complement behavior when a signed integer overflows? Of course there are. Many compilers will have

this behavior when optimizations are turned off, for example, and GCC has an option (-fwrapv) for enforcing this behavior at all optimization levels. Other compilers will have this behavior at all optimization levels by default.

There are also, it should go without saying, compilers that do not have two's complement behavior for signed overflows. Moreover, there are compilers (like GCC) where integer overflow behaved a certain way for many years and then at some point the optimizer got just a little bit smarter and integer overflows suddenly and silently stopped working as expected. This is perfectly OK as far as the standard goes. While it may be unfriendly to developers, it would be considered a win by the compiler team because it will increase benchmark scores.

In summary: There's nothing inherently bad about running with a ball in your hands and also there's nothing inherently bad about shifting a 32-bit number by 33 bit positions. But one is against the rules of basketball and the other is against the rules of C and C++. In both cases, the people designing the game have created arbitrary rules and we either have to play by them or else find a game we like better.

Why Is Undefined Behavior Good?

The good thing — the only good thing! — about undefined behavior in C/C++ is that it simplifies the compiler's job, making it possible to generate very efficient code in certain situations. Usually these situations involve tight loops. For example, high-performance array code doesn't need to perform bounds checks, avoiding the need for tricky optimization passes to hoist these checks outside of loops. Similarly, when compiling a loop that increments a signed integer, the C compiler does not need to worry about the case where the variable overflows and becomes negative: this facilitates several loop optimizations. I've heard that certain tight loops speed up by 30%-50% when the compiler is permitted to take advantage of the undefined nature of signed overflow. Similarly, there have been C compilers that optionally give undefined semantics to unsigned overflow to speed up other loops.

Why Is Undefined Behavior Bad?

When programmers cannot be trusted to reliably avoid undefined behavior, we end up with programs that silently misbehave. This has turned out to be a really bad problem for codes like web servers and web browsers that deal with hostile data because these programs end up being compromised and running code that arrived over the wire. In many cases, we don't actually need the performance gained by exploitation of undefined behavior, but due to legacy code and legacy toolchains, we're stuck with the nasty consequences.

A less serious problem, more of an annoyance, is where behavior is undefined in cases where all it does is make the compiler writer's job a bit easier, and no performance is gained. For example a C implementation has undefined behavior when:

> *An unmatched ‘ or ” character is encountered on a logical source line during tokenization.*

With all due respect to the C standard committee, this is just lazy. Would it really impose an undue burden on C implementors to require that they emit a compile-time error message when quote marks are unmatched? Even a 30 year-old (at the time C99 was standardized) systems programming language can do better than this. One suspects that the C standard body simply got used to throwing behaviors into the “undefined” bucket and got a little carried away. Actually, since the C99 standard lists 191 different kinds of undefined behavior, it's fair to say they got a lot carried away.

Understanding the Compiler's View of Undefined Behavior

The key insight behind designing a programming language with undefined behavior is that the compiler is only obligated to consider cases where the behavior is defined. We'll now explore the implications of this.

If we imagine a C program being executed by the C abstract machine, undefined behavior is very easy to understand: each operation performed by the program is either defined or undefined, and usually it's pretty clear which is which. Undefined behavior becomes difficult to deal with when we start being concerned with all possible executions of a program. Application developers, who need code to be correct in every situation, care about this, and so do compiler developers, who need to emit machine code that is correct over all possible executions.

Talking about all possible executions of a program is a little tricky, so let's make a few simplifying assumptions. First, we'll discuss a single C/C++ function instead of an entire program. Second, we'll assume that the function terminates for every input. Third, we'll assume the function's execution is deterministic; for example, it's not cooperating with other threads via shared memory. Finally, we'll pretend that we have infinite computing resources, making it possible to exhaustively test the function. Exhaustive testing means that all possible inputs are tried, whether they come from arguments, global variables, file I/O, or whatever.

The exhaustive testing algorithm is simple:

1. Compute next input, terminating if we've tried them all
2. Using this input, run the function in the C abstract machine, keeping track of whether any operation with undefined behavior was executed
3. Go to step 1

Enumerating all inputs is not too difficult. Starting with the smallest input (measured in bits) that the function accepts, try all possible bit patterns of that size. Then move to the next size. This process may or may not terminate but it doesn't matter since we have infinite computing resources.

For programs that contain unspecified and implementation-defined behaviors, each input may result in several or many possible executions. This doesn't fundamentally complicate the situation.

OK, what has our thought experiment accomplished? We now know, for our function, which of these categories it falls into:

- Type 1: Behavior is defined for all inputs
- Type 2: Behavior is defined for some inputs and undefined for others
- Type 3: Behavior is undefined for all inputs

Type 1 Functions

These have no restrictions on their inputs: they behave well for all possible inputs (of course, "behaving well" may include returning an error code). Generally, API-level functions and functions that deal with unsanitized data should be Type 1. For example, here's a utility function for performing integer division without executing undefined behaviors:

```
int32_t safe_div_int32_t (int32_t a, int32_t b) {
  if ((b == 0) || ((a == INT32_MIN) && (b == -1))) {
    report_integer_math_error();
    return 0;
  } else {
    return a / b;
  }
}
```

Since Type 1 functions never execute operations with undefined behavior, the compiler is obligated to generate code that does something sensible regardless of the function's inputs. We don't need to consider these functions any further.

Type 3 Functions

These functions admit no well-defined executions. They are, strictly speaking, completely meaningless: the compiler is not even obligated to generate even a return instruction. Do Type 3 functions really exist? Yes, and in fact they are common. For example, a function that — regardless of input — uses a variable without initializing it is easy to unintentionally write. Compilers are getting smarter and smarter about recognizing and exploiting this kind of code. Here's a great example from the Google Native Client project:

When returning from trusted to untrusted code, we must sanitize the return address before taking it. This ensures that untrusted code cannot use the syscall interface to vector execution to an arbitrary address. This role is entrusted to the function NaClSandboxAddr, in sel_ldr.h. Unfortunately, since r572, this function has been a no-op on x86.

-- What happened?

During a routine refactoring, code that once read

aligned_tramp_ret = tramp_ret & ~(nap->align_boundary - 1);

was changed to read

return addr & ~(uintptr_t)((1 << nap->align_boundary) - 1);

Besides the variable renames (which were intentional and correct), a shift was introduced, treating nap->align_boundary as the log2 of bundle size.

We didn't notice this because NaCl on x86 uses a 32-byte bundle size. On x86 with gcc, (1 << 32) == 1. (I believe the standard leaves this behavior undefined, but I'm rusty.) Thus, the entire sandboxing sequence became a no-op.

This change had four listed reviewers and was explicitly LGTM'd by two. Nobody appears to have noticed the change.

-- Impact

There is a potential for untrusted code on 32-bit x86 to unalign its instruction stream by constructing a return address and making a syscall. This could subvert the validator. A similar vulnerability may affect x86- 64.

> *ARM is not affected for historical reasons: the ARM implementation masks the untrusted return address using a different method.*

What happened? A simple refactoring put the function containing this code into Type 3. The person who sent this message believes that x86-gcc evaluates (1<<32) to 1, but there's no reason to expect this behavior to be reliable (in fact it is not on a few versions of x86-gcc that I tried). This construct is definitely undefined and of course the compiler can do done anything it wants. As is typical for a C compiler, it chose to simply not emit the instructions corresponding to the undefined operation. (A C compiler's #1 goal is to emit efficient code.) Once the Google programmers gave the compiler the license to kill, it went ahead and killed. One might ask: Wouldn't it be great if the compiler provided a warning or something when it detected a Type 3 function? Sure! But that is not the compiler's priority.

The Native Client example is a good one because it illustrates how competent programmers can be suckered in by an optimizing compiler's underhanded way of exploiting undefined behavior. A compiler that is very smart at recognizing and silently destroying Type 3 functions becomes effectively evil, from the developer's point of view.

Type 2 Functions

These have behavior that is defined for some inputs and undefined for others. This is the most interesting case for our purposes. Signed integer divide makes a good example:

```
int32_t unsafe_div_int32_t (int32_t a, int32_t b) {
  return a / b;
}
```

This function has a precondition; it should only be called with arguments that satisfy this predicate:

```
(b != 0) && (!((a == INT32_MIN) && (b == -1)))
```

Of course it's no coincidence that this predicate looks a lot like the test in the Type 1 version of this function. If you, the caller, violate this precondition, your program's meaning will be destroyed. Is it OK to write functions like this, that have non-trivial preconditions? In general, for internal utility functions this is perfectly OK as long as the precondition is clearly documented.

Now let's look at the compiler's job when translating this function into object code. The compiler performs a case analysis:

- Case 1: `(b != 0) && (!((a == INT32_MIN) && (b == -1)))`
  Behavior of / operator is defined → Compiler is obligated to emit code computing a / b
- Case 2: `(b == 0) || ((a == INT32_MIN) && (b == -1))`
  Behavior of / operator is undefined → Compiler has no particular obligations

Now the compiler writers ask themselves the question: What is the most efficient implementation of these two cases? Since Case 2 incurs no obligations, the simplest thing is to simply not consider it. The compiler can emit code only for Case 1.

A Java compiler, in contrast, has obligations in Case 2 and must deal with it (though in this particular case, it is likely that there won't be runtime overhead since processors can usually provide trapping behavior for integer divide by zero).

Let's look at another Type 2 function:

```
int stupid (int a) {
  return (a+1) > a;
}
```

The precondition for avoiding undefined behavior is:

```
(a != INT_MAX)
```

Here the case analysis done by an optimizing C or C++ compiler is:

- Case 1: `a != INT_MAX`
  Behavior of + is defined → Computer is obligated to return 1
- Case 2: `a == INT_MAX`
  Behavior of + is undefined → Compiler has no particular obligations

Again, Case 2 is degenerate and disappears from the compiler's reasoning. Case 1 is all that matters. Thus, a good x86-64 compiler will emit:

```
stupid:
  movl $1, %eax
  ret
```

If we use the -fwrapv flag to tell GCC that integer overflow has two's complement behavior, we get a different case analysis:

- Case 1: `a != INT_MAX`
  Behavior is defined → Computer is obligated to return 1
- Case 2: `a == INT_MAX`
  Behavior is defined → Compiler is obligated to return 0

Here the cases cannot be collapsed and the compiler is obligated to actually perform the addition and check its result:

```
stupid:
  leal 1(%rdi), %eax
  cmpl %edi, %eax
  setg %al
  movzbl %al, %eax
  ret
```

Similarly, an ahead-of-time Java compiler also has to perform the addition because Java mandates two's complement behavior when a signed integer overflows (I'm using GCJ for x86-64):

```
_ZN13HelloWorldApp6stupidEJbii:
  leal 1(%rsi), %eax
  cmpl %eax, %esi
  setl %al
  ret
```

This case-collapsing view of undefined behavior provides a powerful way to explain how compilers really work. Remember, their main goal is to give you fast code that obeys the letter of the law, so they will attempt to forget about undefined behavior as fast as possible, without telling you that this happened.

A Fun Case Analysis

About a year ago, the Linux kernel started using a special GCC flag to tell the compiler to avoid optimizing away useless null-pointer checks. The code that caused developers to add this flag looks like this (I've cleaned up the example just a bit):

```
static void __devexit agnx_pci_remove (struct pci_dev *pdev)
{
  struct ieee80211_hw *dev = pci_get_drvdata(pdev);
  struct agnx_priv *priv = dev->priv;

  if (!dev) return;
```

```
  ... do stuff using dev ...
}
```

The idiom here is to get a pointer to a device struct, test it for null, and then use it. But there's a problem! In this function, the pointer is dereferenced before the null check. This leads an optimizing compiler (for example, gcc at -O2 or higher) to perform the following case analysis:

- Case 1: `dev == NULL`
  "dev->priv" has undefined behavior → Compiler has no particular obligations
- Case 2: `dev != NULL`
  Null pointer check won't fail → Null pointer check is dead code and may be deleted

As we can now easily see, neither case necessitates a null pointer check. The check is removed, potentially creating an exploitable security vulnerability.

Of course the problem is the use-before-check of pci_get_drvdata()'s return value, and this has to be fixed by moving the use after the check. But until all such code can be inspected (manually or by a tool), it was deemed safer to just tell the compiler to be a bit conservative. The loss of efficiency due to a predictable branch like this is totally negligible. Similar code has been found in other parts of the kernel.

Living with Undefined Behavior

In the long run, unsafe programming languages will not be used by mainstream developers, but rather reserved for situations where high performance and a low resource footprint are critical. In the meantime, dealing with undefined behavior is not totally straightforward and a patchwork approach seems to be best:

- Enable and heed compiler warnings, preferably using multiple compilers
- Use static analyzers (like Clang's, Coverity, etc.) to get even more warnings
- Use compiler-supported dynamic checks; for example, gcc's -ftrapv flag generates code to trap signed integer overflows
- Use tools like Valgrind to get additional dynamic checks

- When functions are “type 2” as categorized above, document their preconditions and postconditions
- Use assertions to verify that functions’ preconditions are postconditions actually hold
- Particularly in C++, use high-quality data structure libraries

Basically: be very careful, use good tools, and hope for the best.

regehr / July 9, 2010 / Computer Science, Software Correctness

---

29 thoughts on “A Guide to Undefined Behavior in C and C++, Part 1”

Michael Norrish

July 9, 2010 at 6:13 am

Nice write-up!

Eric LaForest

July 9, 2010 at 6:56 am

Thought-provoking post, thank you.

But I’m puzzled: what type of optimization can infer the indeterminacy of stupid() or the null-pointer check? I’ve not heard of it.

Wouldn’t it be better for the compiler to decide what to do based on what knowledge it has at compile-time?

For example: stupid(foo) would compile the full code, while stupid(5) would compile to a literal, as per constant propagation and expression simplification.

Similarly, in the case of the NULL-check, why bother at all? The value of dev is not known at compile time, and the first dereference would trigger a segfault in the case of a NULL anyway.

---

regehr

July 9, 2010 at 10:13 am

Hi Eric- I may need to update this post to be a bit more clear about these things!

I don' t think these optimizations have any specific names, nor do I think they're written up in the textbooks. But basically they all fall under the umbrella of "standard dataflow optimizations." In other words, the compiler learns some facts and propagates them around in order to make changes elsewhere. The only difference is in the details of what facts are learned.

Just to be clear, these optimizations are absolutely based on knowledge the compiler has at compile time. Everything I described in this post is just a regular old compile-time optimization.

stupid(foo) — where foo is a free variable — compiles to "return 1" using the case analysis.

Re. the null check example, remember this is in the Linux kernel. In the best possible case, accessing a null pointer crashes the machine. In the worst case there is no crash: exploit code is waiting to take over the machine when you access the null pointer. This is precisely the case that the kernel programmers are worried about. This is not theoretical: if your Linux kernel accesses a null pointer, I can probably own your machine.

Matthias Felleisen

July 9, 2010 at 10:44 am

Thank you.

What's really sad is that some so-called high-level languages like Scheme intentionally include undefined behavior, too. This may make Scheme look like a real systems language after all.

I have been preaching the 'unsafe and undefined gospel' for a long time in PL courses. Indeed, I have been preaching it for such a long time that former students still have T shirts with my quote "Scheme is just as bad as FORTRAN" for the same issue. Indeed, some HotSpot compiler authors may still own this T shirt.

Sadly, I never got support from 'real' compiler colleagues at Rice nor from the systems people. Real man just cope. Shut up and work.

So thanks for speaking up as a "systems" person.

— Matthias

---

regehr

July 9, 2010 at 11:15 am

Hi Matthias- Thanks for the note!

I feel like certain languages were designed by and for compiler people. Optimizations good, everything else: irrelevant. Hopefully these languages will lose (or be revised) to cope with the modern situation where machine resources are relatively cheap and program errors are relatively costly. Of course multicores will probably set us back a couple decades in terms of program correctness…

I was very surprised to learn from Matthew about Scheme's undefined behaviors.

Matthias Felleisen

July 9, 2010 at 12:23 pm

You write “Of course multicores will probably set us back a couple decades in terms of program correctness…” How sad and how true!

In Scheme’s case, it is fortunately acceptable for a compiler to implement a safe and determinate language, which is what Matthew’s PLT Scheme did and what Racket does now. Sadly, compiler writers love indeterminate specs and they love writing language specifications even more. Perhaps the latter is the real reason that we will not get away from such bad languages for ever.

Adam Morrison

July 10, 2010 at 12:20 pm

I’m not sure about the Google Native client example. Looks to me like the compiler emitted x86 code that, when passed 32 as input, would calculate (1<<32) == 1.

It just so happened that this function was always called with the "bundle_size" being 32 and as a result it became a no-op semantically. It didn't mask out the low bits of the address like it was supposed to.

Ben L. Titzer

July 12, 2010 at 5:43 pm

Though I completely agree with the idea of making programs’ semantics as well defined as possible (independent of implementation and target machine of course), there are always cracks to slip through. E.g. most semantics assume that the target machine has enough resources to actually run the program. Of course then they define what should happen if the machine does not– OutOfMemoryError, StackOverFlowError, and the like. But what about the case when the machine *almost* has enough resources. Then the impact of optimizations can be felt. E.g. how much of the heap is occupied by class metadata? Even with the same heap size, the same program might run to completion (or continue working) or throw OutOfMemory on different VMs. Similarly with stack size–if the compiler or VM performs tail recursion

elimination, the program may work fine, but fail immediately without (note that no compliant JVM does).

Although one can define semantics to be nondeterministic, it is unusual to go to all the trouble of formally defining them in order to leave some part nondeterministic. We try to eradicate it but nondeterminism keeps creeping in, like these choice examples in Java:

* result of java.lang.System.identityHashCode()
* order of finalizers being run, and on which thread
* policy for clearing of SoftReferences

Ben L. Titzer

July 12, 2010 at 5:46 pm

Just to be clear though: nondeterministic behavior is far preferable to undefined behavior 🙂

---

regehr

July 12, 2010 at 7:30 pm

Hey Ben- Let's be clear that there are two separate questions here. First, whether or not an error occurs. Second, what should the language do when an error occurs.

People developing security-, safety-, and mission-critical software care a lot about the first one and it's a really hard problem, optimizations matter, etc.

This post was only about the second one: does the program halt with a nice error or does it keep going in some screwed-up state. Nailing this down seems like the first order of business, then later on we can worry about making offline guarantees about error-freedom and all that.

Eugene Toder

July 30, 2010 at 4:32 pm

GCC actually tends to evaluate (1 <= 32 to 0. This is even more optimal than evaluating it to x, as there's no need to evaluate x, literal 0 can trigger further optimizations and literal 0 is very cheap on most platforms. In this case most likely GCC was not able to find undefined behaviour at compile-time and generated x86 shl instruction. 32-bit shl on x86 only looks at the lowest 5 bits of it's RHS, thus (1 << 32) == (1 << 0) == 1. However non-intuitive this is, this is a result of CPU optimization, not compiler optimization.

Nadav

August 17, 2010 at 11:30 pm

This is a good article!

I wanted to point out that in Verilog and VHDL (hardware description languages) you have syntax that is undefined. It is a part of the standard of the language but it is unsynthesizable to hardware circuits.

Neil Harding

August 19, 2010 at 10:18 am

The reason that arithmetic operations are undefined is due to not requiring a particular implementation, so if you used a processor with BCD (binary coded decimal) values for integer operations, or 1's complement format then INT_MAX + 1 would return different values than a 2's complement format architecture.

ASSERT can be used to check for preconditions, but since you are running in debug mode when this is enabled, some optimizations are not enabled and so the preconditions may hold true in debug mode, but not in release mode.

I actually prefer coding in 68000 (6502 & Z80 required too much work for the simple operations) to programming in C, or Java. I found I could do optimal code, and use the condition code flags to perform multiple checks at one time.

So a = a + 1, would set Z flag if a was = -1, V flag (overflow) if a = MAX_INT, and N flag (if result < 0, which include MAX_INT). I've done some x86 assembly but since it is such a horrible mess, then C/C++ is preferable.

Peter da Silva

August 19, 2010 at 4:23 pm

The reason for things like "An unmatched ‘ or ” character is encountered on a logical source line during tokenization" being undefined is not to make the compiler's job easier, it's to make the standards body's job possible. Many of these kinds of undefined behavior are cases where:

* Important compilers did it differently.
* Important code depended on what their particular compiler did.

I am 99.44% positive that there are a number of cases where it would make a lot of sense to define certain behavior as an error, but if you did that you'd have to rewrite parts of the Linux kernel or the NT kernel because GCC and Microsoft C did things different ways... and since you're never going to compile the Linux kernel with anything but GCC (try it some time) or the NT kernel with Microsoft C, neither side has a good reason to back down.

---

regehr

August 19, 2010 at 9:59 pm

Neil, how many non-2's complement targets are out there? I know this rationale was used historically but it hardly seems relevant to C99 and C++0x.

I agree with you that the lack of access to overflow flags in C and C++ is really annoying. It makes certain kinds of code hard or impossible to express efficiently.

---

regehr

August 19, 2010 at 10:02 pm

Peter of course you're right, thanks for pointing this out. The standards body certainly had an unenviable task.

Kumari Swarnim

August 20, 2010 at 4:50 am

It is nice.

Steve

August 20, 2010 at 8:53 am

You go too far when you say that anyone relying on twos-complement overflow behaviour will one day be proven wrong. Of course pedantically it's true, but in reality, people know broadly what kind of platform they're developing for even if not precisely which platform. Twos-complement is pretty much universal now, and C/C++ compiler developers would be insane to not ensure it just worked as expected irrespective of the letter of the standards. Anyone developing for an obscure sign-magnitude platform or whatever will know about it.

---

regehr ▲

August 20, 2010 at 11:18 am

Hi Steve- I must have failed to explain things clearly. This is the situation: today's C and C++ compilers do not have 2's complement semantics for signed overflow. Did you read the example in the post where real C compilers evaluate (x+1)>x as "true"? Do I need to point out that this result is not correct under a 2's complement interpretation of integers?

Steve

August 20, 2010 at 3:20 pm

Hi – no you didn't explain badly, but my point still stands. When dealing with the boundary between what is formally-undefined-but-expected and what is just plain undefined, there's always going to be a degree of subjectivity, but here's the thing – it is (for example) impractical if not impossible to implement a big-integer library without making (reasonable but standards-undefined) assumptions about integer representation and overflow behaviour. Big integer libraries exist. They won't run on every platform everywhere, but they still manage a fair bit of portability.

If you write "(x+1)>x", the real question is "why are you doing this?". Of course the optimisation makes sense, just as the 2s complement assumption would make sense. But equally, this is an artificial example. For real world code, you can rely on the fact that compiler writers actually want scripting languages and other big-integer clients to carry on working too. I believe GCC even *uses* GNUs big integer library to do it's compile-time calculations these days.

I repeat – pedantically, yes, you are going to encounter problems and odd corner cases – but stick within the kinds of coding patterns that are widely used and your code will work irrespective of "undefined behaviour".

The purpose of a real-world compiler is to compile real-world code and, while optimisation can sometimes get overzealous, the integer overflow issue isn't as bad as you make out.

OTOH – pointer alias analysis (or rather the failure to detect an alias due to pointer casts and arithmetic) is a real expletive-causing issue. I've had that with GCC recently, and I couldn't really figure out a resolution other than (1) have tons of template bloat to avoid having a type-unsafe implementation, or (2) dial down the optimisations. Yet no-one can seriously claim that there's no history of pointer casts and arithmetic in C and C++.

But IMO this ones a reason to complain to the compiler writers – as I said, the purpose of a real-world compiler is to compile real-world code. It's virtually impossible to write a real-world app that doesn't invoke some kind of undefined behaviour in C or C++, so compiler writers have more responsibilities than just complying with the standards. The end users, after all, are you and me – not just the standards people.

That said, so far I haven't looked that hard for a resolution, and that's the real issue here with the alias analysis. C and C++ are languages for people who are willing to patch up the problems from time to time (or stick to known versions of known compilers), and I just haven't checked how to fix this one low-priority library yet.

---

regehr

August 20, 2010 at 6:42 pm

Hi Steve- There is much merit to what you say. However, you are wrong about one fundamental point: when dealing with a programming language the real question is not "why are you doing this." I actually wrote a post about this exact topic a while ago:

http://blog.regehr.org/archives/47

I'll tell you what, let's run an experiment. If you've looked at part 2 of this series of posts, you'll see that my group has a tool for detecting integer undefined behaviors. I'll run this on GMP, which I suspect is the most popular bignum package (if you have other ideas, let me know).

My expectation is that every single signed overflow in GMP will be considered a bug by the GMP developers and will be fixed after I report it. Your position, if I guess correctly, is that they are happy to leave these in there because the compiler somehow understands what the developer is trying to do, and respects 2's complement behavior when it really matters. Does that sound right?

Steve

August 21, 2010 at 1:59 pm

I got your e-mails, and to be honest, I'm surprised this is going on so long. After all, we agree that C and C++ leave a lot undefined, and that means that those languages aren't as safe as e.g. Ada.

As for what happens on GMP, my guess is that they'll look at the specific cases and see what they can do. Especially as, based on you finding a grand total of nine issues, it looks like I was wrong in saying that doing big integers is impractical without relying on integer representation and overflow behaviour.

Since it's impossible to ever say "I'm wrong" without a "but" 😉

Am I correct in saying that until C99 and stdint.h, there was no way in standard C to specify that you want a 32bit (or any other specific size) integer? GMP was first released in 1991, certainly. Evolving towards greater portability is hardly a surprise, but it *is* evolution, with problems fixed as and when they're found.

Those undefined behaviour "bugs" are there now and have probably been there a while. I assume GMP have strong unit tests, so if a real problem arose, it would have been noticed.

So one relevant question is – is it actually productive to be spending time hunting down and fixing undefined behaviour bugs that don't cause anyone a problem, when they could be investing that time in something else?

As you said, one thing the developer has to do is to hope. I'd say it's more a matter of expecting to do maintenance as platforms and compilers evolve. And even if you code in Ada, you'd still need to do maintenance from time to time – e.g. you might find around now that you need 64 bit integers, where not only didn't you anticipate it years ago, but your compiler wouldn't allow it anyway.

Is the Ada way better that the C/C++ way? I think so, and apparently so do you. But reality is that very few people use Ada, and while C and C++ are less than ideal, they're only occasionally fatal. And lets face it – GNAT even has those mandated overflow checks disabled by default for performance reasons, so using Ada is no guarantee in itself. And I guess if your unit tests are strong enough, it doesn't matter – don't laugh, some people can manage to write unit tests, honest.

Actually, since you mentioned LLVM and Clang in an e-mail – bugpoint is something I must look at some time.

Moving on – if the compiler doesn't care about intent and only cares about the letter of the standard, how do you balance that against the fact that for half of its history C didn't have a standard – the ratio being somewhat worse for C++.

Obviously it's not the job of the compiler to guess, but there's no fundamental difference between the guys who write standards and the guys who write compilers – they're all people and all (hopefully) experienced developers. Someone somewhere figures out what is needed, documents it and develops it, not necessarily in that order. If they develop something that can't cope with real world code, thousands of other developers will shout foul. In the aftermath, either the offending compilers or real-world programming practice will adapt. In a world where perfection is rare, this mostly sort-of works.

For example, can you imagine what would happen if the Python devs suddenly decided (with no standard to say they can't) to change the semantics of the division operator? Errrm. Oh. Errr – actually, forget that bit 😉

Thinking about it, my whole argument requires people to shout foul from time to time, which you were doing. So maybe again I'm on the wrong side of things.

---

regehr

August 21, 2010 at 2:21 pm

Hi Steve- Yeah, we mostly agree. I think the point of disagreement is whether people should fix undefined behaviors if they're not currently causing problems. Of course this is an individual choice made by each developer. My position on the matter, for any software I cared about, would be to fix these issues once I knew about them — it just saves time later. It's sort of like fixing compiling warnings that aren't pointing out major problems — often you just do it to get the tool to shut up, so that next time the tool says something you'll notice.

Steve

August 21, 2010 at 7:16 pm

On the warnings thing, I see the point.

BTW – I just realised a minor misunderstanding, which kind of explains why you said about compilers guessing intent. When I said "If you write "(x+1)>x", the real question is "why are you doing this?".", my intent wasn't to suggest the compiler should work out your intent, but to point out that this isn't sane real-world code. Depending on your choice of common interpretation this either evaluates to (true) or to (x != INT_MAX) – and one of those is what you'd expect to see in real code.

OTOH the issue can happen indirectly, and implicit inlines can lead to the optimisation happening where you wouldn't expect it, which is potentially a problem.

---

regehr

August 22, 2010 at 8:56 pm

Hi Steve- – Yes this is exactly right: machine generated code, macros, inlining, etc. can cause bizarre source code that a human would never write. Also, most good compilers these days will perform inlining across compilation units, so it's not really easy to predict what the code that the compiler finally sees will look like.

Steve

August 25, 2010 at 3:36 pm

Cleared up my full misundertanding here...

http://stackoverflow.com/questions/3569424/how-to-do-a-double-chunk-add-with-no-undefined-behaviour

Basically, big integers without undefined behaviour have been perfectly possible for some time – the languages are (slightly, but in a very significant way) better defined than I realised, and have been for a little over ten years.

Pingback: Undefined behavior in C and C++ « IPhVu::iLearn

Pingback: Undefined Behavior of C and C++ | Itsy Bitsy

Pingback: outerplanet

Comments are closed.

Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior

Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama
MIT CSAIL

Abstract

This paper studies an emerging class of software bugs called *optimization-unstable code*: code that is unexpectedly discarded by compiler optimizations due to undefined behavior in the program. Unstable code is present in many systems, including the Linux kernel and the Postgres database. The consequences of unstable code range from incorrect functionality to missing security checks.

To reason about unstable code, this paper proposes a novel model, which views unstable code in terms of optimizations that leverage undefined behavior. Using this model, we introduce a new static checker called STACK that precisely identifies unstable code. Applying STACK to widely used systems has uncovered 160 new bugs that have been confirmed and fixed by developers.

1 Introduction

The specifications of many programming languages designate certain code fragments as having *undefined behavior* [15: §2.3], giving compilers the freedom to generate instructions that behave in arbitrary ways in those cases. For example, in C the "use of a nonportable or erroneous program construct or of erroneous data" leads to undefined behavior [24: §3.4.3].

One way in which compilers exploit undefined behavior is to optimize a program under the assumption that the program never invokes undefined behavior. A consequence of such optimizations is especially surprising to many programmers: code which works with optimizations turned off (e.g., `-O0`) breaks with a higher optimization level (e.g., `-O2`), because the compiler considers part of the code dead and discards it. We call such code *optimization-unstable code*, or just unstable code for short. If the discarded unstable code happens to be used for security checks, the optimized system will become vulnerable to attacks.

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522728

```
char *buf = ...;
char *buf_end = ...;
unsigned int len = ...;
if (buf + len >= buf_end)
    return;   /* len too large */
if (buf + len < buf)
    return;   /* overflow, buf+len wrapped around */
/* write to buf[0..len-1] */
```

Figure 1: A pointer overflow check found in several code bases. The code becomes vulnerable as gcc optimizes away the second `if` statement [13].

This paper presents the first systematic approach for reasoning about and detecting unstable code. We implement this approach in a static checker called STACK, and use it to show that unstable code is present in a wide range of systems software, including the Linux kernel and the Postgres database. We estimate that unstable code exists in 40% of the 8,575 Debian Wheezy packages that contain C/C++ code. We also show that compilers are increasingly taking advantage of undefined behavior for optimizations, leading to more vulnerabilities related to unstable code.

To understand unstable code, consider the pointer overflow check `buf + len < buf` shown in Figure 1, where `buf` is a pointer and `len` is a positive integer. The programmer's intention is to catch the case when `len` is so large that `buf + len` wraps around and bypasses the first check in Figure 1. We have found similar checks in a number of systems, including the Chromium browser [7], the Linux kernel [49], and the Python interpreter [37].

While this check appears to work with a flat address space, it fails on a segmented architecture [23: §6.3.2.3]. Therefore, the C standard states that an overflowed pointer is undefined [24: §6.5.6/p8], which allows gcc to simply assume that no pointer overflow ever occurs on *any* architecture. Under this assumption, `buf + len` must be larger than `buf`, and thus the "overflow" check always evaluates to *false*. Consequently, gcc removes the check, paving the way for an attack to the system [13].

In addition to introducing new vulnerabilities, unstable code can amplify existing weakness in the system. Figure 2 shows a mild defect in the Linux kernel, where the programmer incorrectly placed the dereference `tun->sk`

```
struct tun_struct *tun = ...;
struct sock *sk = tun->sk;
if (!tun)
    return POLLERR;
/* write to address based on tun */
```

Figure 2: A null pointer dereference vulnerability (CVE-2009-1897) in the Linux kernel, where the dereference of pointer `tun` is before the null pointer check. The code becomes exploitable as gcc optimizes away the null pointer check [10].

before the null pointer check `!tun`. Normally, the kernel forbids access to page zero; a null `tun` pointing to page zero causes a kernel oops at `tun->sk` and terminates the current process. Even if page zero is made accessible (e.g., via `mmap` or some other exploits [25, 45]), the check `!tun` would catch a null `tun` and prevent any further exploits. In either case, an adversary should *not* be able to go beyond the null pointer check.

Unfortunately, unstable code can turn this simple bug into an exploitable vulnerability. For example, when gcc first sees the dereference `tun->sk`, it concludes that the pointer `tun` must be non-null, because the C standard states that dereferencing a null pointer is undefined [24: §6.5.3]. Since `tun` is non-null, gcc further determines that the null pointer check is unnecessary and eliminates the check, making a privilege escalation exploit possible that would not otherwise be [10].

Poor understanding of unstable code is a major obstacle to reasoning about system behavior. For programmers, compilers that discard unstable code are often "baffling" and "make no sense" [46], merely gcc's "creative reinterpretation of basic C semantics" [27]. On the other hand, compiler writers argue that the C standard allows such optimizations, which many compilers exploit (see §2.3); it is the "broken code" [17] that programmers should fix.

Who is right in this debate? From the compiler's point of view, the programmers made a mistake in their code. For example, Figure 2 clearly contains a bug, and even Figure 1 is arguably incorrect given a strict interpretation of the C standard. However, these bugs are quite subtle, and understanding them requires detailed knowledge of the language specification. Thus, it is not surprising that such bugs continue to proliferate.

From the programmer's point of view, the compilers are being too aggressive with their optimizations. However, optimizations are important for achieving good performance; many optimizations fundamentally rely on the precise semantics of the C language, such as eliminating needless null pointer checks or optimizing integer loop variables [20, 29]. Thus, it is difficult for compiler writers to distinguish legal yet complex optimizations from an optimization that goes too far and violates the programmer's intent [29: §3].

This paper helps resolve this debate by introducing a model for identifying unstable code that allows a compiler to generate precise warnings when it removes code based on undefined behavior. The model specifies precise conditions under which a code fragment can induce undefined behavior. Using these conditions we can identify fragments that can be eliminated under the assumption that undefined behavior is never triggered; specifically, any fragment that is reachable only by inputs that trigger undefined behavior is unstable code. We make this model more precise in §3.

The Stack checker implements this model to identify unstable code. For the example in Figure 2, it emits a warning that the null pointer check `!tun` is unstable due to the earlier dereference `tun->sk`. Stack first computes the undefined behavior conditions for a wide range of constructs, including pointer and integer arithmetic, memory access, and library function calls. It then uses a constraint solver [3] to determine whether the code can be simplified away given the undefined behavior conditions, such as whether the code is reachable only when the undefined behavior conditions are *true*. We hope that Stack will help programmers find unstable code in their applications, and that our model will help compilers make better decisions about what optimizations might be unsafe and when an optimizer should produce a warning.

We implemented the Stack checker using the LLVM compiler framework [30] and the Boolector solver [3]. Applying it to a wide range of systems uncovered 160 new bugs, which were confirmed and fixed by the developers. We also received positive feedback from outside users who, with the help of Stack, fixed additional bugs in both open-source and commercial code bases. Our experience shows that unstable code is a widespread threat in systems, that an adversary can exploit vulnerabilities caused by unstable code with major compilers, and that Stack is useful for identifying unstable code.

The main contributions of this paper are:

- a new model for understanding unstable code,
- a static checker for identifying unstable code, and
- a detailed case study of unstable code in real systems.

Another conclusion one can draw from this paper is that language designers should be careful with defining language construct as undefined behavior. Almost every language allows a developer to write programs that have undefined meaning according to the language specification. Our experience with C/C++ indicates that being liberal with what is undefined can lead to subtle bugs.

The rest of the paper is organized as follows. §2 provides background information. §3 presents our model of unstable code. §4 outlines the design of Stack. §5 summarizes its implementation. §6 reports our experience of applying Stack to identify unstable code and evaluates Stack's techniques. §7 covers related work. §8 concludes.

	Construct	Sufficient condition	Undefined behavior
Language	$p + x$	$p_\infty + x_\infty \notin [0, 2^n - 1]$	pointer overflow
	$*p$	$p = \texttt{NULL}$	null pointer dereference
	$x \text{ op}_s y$	$x_\infty \text{ op}_s y_\infty \notin [-2^{n-1}, 2^{n-1} - 1]$	signed integer overflow
	x / y, x % y	$y = 0$	division by zero
	$x << y$, $x >> y$	$y < 0 \vee y \geq n$	oversized shift
	$a[x]$	$x < 0 \vee x \geq \texttt{ARRAY\_SIZE}(a)$	buffer overflow
Library	$\texttt{abs}(x)$	$x = -2^{n-1}$	absolute value overflow
	$\texttt{memcpy}(\texttt{dst}, \texttt{src}, \texttt{len})$	$\lvert\texttt{dst} - \texttt{src}\rvert < \texttt{len}$	overlapping memory copy
	use q after $\texttt{free}(p)$	$\text{alias}(p, q)$	use after free
	use q after $p' := \texttt{realloc}(p, ...)$	$\text{alias}(p, q) \wedge p' \neq \texttt{NULL}$	use after realloc

Figure 3: A list of sufficient (though not necessary) conditions for undefined behavior in certain C constructs [24: §J.2]. Here p, p', q are n-bit pointers; x, y are n-bit integers; a is an array, the capacity of which is denoted as $\texttt{ARRAY\_SIZE}(a)$; op_s refers to binary operators +, -, *, /, % over signed integers; x_∞ means to consider x as infinitely ranged; NULL is the null pointer; $\text{alias}(p, q)$ predicates whether p and q point to the same object.

2 Background

This section provides some background on undefined behavior and how it can lead to unstable code. It builds on earlier surveys [26, 41, 49] and blog posts [29, 39, 40] that describe unstable code examples, and extends them by investigating the evolution of optimizations in compilers.

2.1 Undefined behavior

Figure 3 shows a list of constructs and their undefined behavior conditions, as specified in the C standard [24: §J.2]. One category of undefined behavior is simply programming errors, such as null pointer dereference, buffer overflow, and use after free. The other category is non-portable constructs, the hardware implementations of which often have subtle differences.

For instance, when signed integer overflow or division by zero occurs, a division instruction traps on x86 [22: §3.2], while it silently produces an undefined result on PowerPC [21: §3.3.8]. Another example is shift instructions: left-shifting a 32-bit one by 32 bits produces 0 on ARM and PowerPC, but 1 on x86; however, left-shifting a 32-bit one by 64 bits produces 0 on ARM, but 1 on x86 and PowerPC. Wang et al.'s survey [49] provides more details of such architectural differences.

To build a portable system, the language standard could impose uniform behavior over erroneous or non-portable constructs, as many higher-level languages do. In doing so, the compiler would have to synthesize extra instructions. For example, to enforce well-defined error handling (e.g., run-time exception) on buffer overflow, the compiler would need to insert extra bounds checks for memory access operations. Similarly, to enforce a consistent shift behavior on x86, for every $x << y$ the compiler would need to insert a check against y (unless it is able to prove that y is not oversized), as follows:

$$\text{if } (y < 0 \vee y \geq n) \text{ then } 0 \text{ else } x << y.$$

The C-family languages employ a different approach. Aiming for system programming, their specifications choose to trust programmers [23: §0] and assume that their code will never invoke undefined behavior. This assumption gives more freedom to the compiler than simply saying that the result of a particular operation is architecture-dependent. While it allows the compiler to generate efficient code without extra checks, the assumption also opens the door to unstable code.

2.2 Examples of unstable code

The top row of Figure 4 shows six representative examples of unstable code in the form of sanity checks. All of these checks may evaluate to *false* and become dead code under optimizations, even though none appear to directly invoke undefined behavior. We will use them to test existing compilers in §2.3.

The check $p + 100 < p$ resembles Figure 1, which is dead assuming no pointer overflow.

The null pointer check $!p$ with an earlier dereference is from Figure 2, which is dead assuming no null pointer dereference.

The check $x + 100 < x$ with a signed integer x becomes dead assuming no signed integer overflow. It once led to a harsh debate between some C programmers and gcc developers [17].

Another check $x^+ + 100 < 0$ tests whether optimizations perform more elaborate reasoning assuming no signed integer overflow; x^+ is known to be positive.

The shift check $!(1 << x)$ was intended to catch a large x, from a patch to the ext4 file system [31]. It becomes dead assuming no oversized shift amount.

The check $\texttt{abs}(x) < 0$ was used in the PHP interpreter to catch the most negative value (i.e., -2^{n-1}). It becomes dead when optimizations understand this library function and assume no absolute value overflow [18].

	`if` $(p + 100 < p)$	$*p$; `if` $(!p)$	`if` $(x + 100 < x)$	`if` $(x^+ + 100 < 0)$	`if` $(!(1 << x))$	`if` (`abs`$(x) < 0$)
gcc-2.95.3	–	–	`O1`	–	–	–
gcc-3.4.6	–	`O2`	`O1`	–	–	–
gcc-4.2.1	`O0`	–	`O2`	–	–	`O2`
gcc-4.8.1	`O2`	`O2`	`O2`	`O2`	–	`O2`
clang-1.0	`O1`	–	–	–	–	–
clang-3.3	`O1`	–	`O1`	–	`O1`	–
aCC-6.25	–	–	–	–	–	`O3`
armcc-5.02	–	–	`O2`	–	–	–
icc-14.0.0	–	`O2`	`O1`	`O2`	–	–
msvc-11.0	–	`O1`	–	–	–	–
open64-4.5.2	`O1`	–	`O2`	–	–	`O2`
pathcc-1.0.0	`O1`	–	`O2`	–	–	`O2`
suncc-5.12	–	`O3`	–	–	–	–
ti-7.4.2	`O0`	–	`O0`	`O2`	–	–
windriver-5.9.2	–	–	`O0`	–	–	–
xlc-12.1	`O3`	–	–	–	–	–

Figure 4: Optimizations of unstable code in popular compilers: gcc, clang, aCC, armcc, icc, msvc, open64, pathcc, suncc, TI's TMS320C6000, Wind River's Diab compiler, and IBM's XL C compiler. In the examples, p is a pointer, x is a signed integer, and x^+ is a positive signed integer. In each cell, "`O`n" means that the specific version of the compiler optimizes the check into *false* and discards it at optimization level n, while "–" means that the compiler does not discard the check at any level.

2.3 An evolution of optimizations

We chose 12 well-known C/C++ compilers to see what they do with the unstable code examples: two open-source compilers (gcc and clang) and ten recent commercial compilers (HP's aCC, ARM's armcc, Intel's icc, Microsoft's msvc, AMD's open64, PathScale's pathcc, Oracle's suncc, TI's TMS320C6000, Wind River's Diab compiler, and IBM's XL C compiler). For every unstable code example, we test whether a compiler optimizes the check into *false*, and if so, find the lowest optimization level `-O`n at which it happens. The result is shown in Figure 4.

We further use gcc and clang to study the evolution of optimizations, as the history is easily accessible. For gcc, we chose the following representative versions that span more than a decade:

- gcc 2.95.3, the last 2.x, released in 2001;
- gcc 3.4.6, the last 3.x, released in 2006;
- gcc 4.2.1, the last GPLv2 version, released in 2007 and still widely used in BSD systems;
- gcc 4.8.1, the latest version, released in 2013.

For comparison, we chose two versions of clang, 1.0 released in 2009, and the latest 3.3 released in 2013.

We make the following observations of existing compilers from Figure 4. First, discarding unstable code is common among compilers, not just in recent gcc versions as some programmers have claimed [27]. Even gcc 2.95.3 eliminates $x + 100 < x$. Some compilers discard unstable code that gcc does not (e.g., clang on $1 << x$).

Second, from different versions of gcc and clang, we see more unstable code discarded as the compilers evolve to adopt new optimizations. For example, gcc 4.x is more aggressive in discarding unstable code compared to gcc 2.x, as it uses a new value range analysis [36].

Third, discarding unstable code occurs with standard optimization options, mostly at `-O2`, the default optimization level for release build (e.g., `autoconf` [32: §5.10.3]); some compilers even discard unstable code at the lowest level of optimization `-O0`. Hence, lowering the optimization level as Postgres did [28] is an unreliable way of working around unstable code.

Fourth, optimizations exploit undefined behavior not only from the core language features, but also from library functions (e.g., `abs` [18] and `realloc` [40]) as the compilers evolve to understand them.

As compilers improve their optimizations, for example, by implementing new algorithms (e.g., gcc 4.x's value range analysis) or by exploiting undefined behavior from more constructs (e.g., library functions), we anticipate an increase in bugs due to unstable code.

3 Model for unstable code

Discarding unstable code, as the compilers surveyed in §2 do, is legal as per the language standard, and thus is *not* a compiler bug [39: §3]. But, it is baffling to programmers. Our goal is to identify such unstable code fragments and generate warnings for them. As we will see in §6.2, these warnings often identify code that programmers want to fix, instead of having the compiler remove it silently. This goal requires a precise model for understanding unstable code so as to generate warnings only for code that is unstable, and not for code that is trivially dead and can be safely removed. This section introduces a model for thinking about unstable code and a framework with two algorithms for identifying it.

3.1 Unstable code

To formalize a programmer's misunderstanding of the C specification that leads to unstable code, let C* denote a C dialect that assigns well-defined semantics to code fragments that have undefined behavior in C. For example, C* is defined for a flat address space, a null pointer that maps to address zero, and wrap-around semantics for pointer and integer arithmetic [38]. A code fragment e is a statement or expression at a particular source location in program $\mathcal{P}$. If the compiler can transform the fragment e in a way that would change $\mathcal{P}$'s behavior under C* but not under C, then e is unstable code.

Let $\mathcal{P}[e/e']$ be a program formed by replacing e with some fragment e' at the same source location. When is it legal for a compiler to transform $\mathcal{P}$ into $\mathcal{P}[e/e']$, denoted as $\mathcal{P} \rightsquigarrow \mathcal{P}[e/e']$? In a language specification without undefined behavior, the answer is straightforward: it is legal if for every input, both $\mathcal{P}$ and $\mathcal{P}[e/e']$ produce the same result. In a language specification *with* undefined behavior, the answer is more complicated; namely, it is legal if for every input, one of the following is true:

- both $\mathcal{P}$ and $\mathcal{P}[e/e']$ produce the same results without invoking undefined behavior, or
- $\mathcal{P}$ invokes undefined behavior, in which case it does not matter what $\mathcal{P}[e/e']$ does.

Using this notation, we define unstable code below.

Definition 1 (Unstable code)**.** A code fragment e in program $\mathcal{P}$ is unstable *w.r.t.* language specifications C and C* iff there exists a fragment e' such that $\mathcal{P} \rightsquigarrow \mathcal{P}[e/e']$ is legal under C but *not* under C*.

For example, for the sanity checks listed in Figure 4, a C compiler is entitled to replace them with *false*, as this is legal according to the C specification, whereas a hypothetical C* compiler cannot do the same. Therefore, these checks are unstable code.

3.2 Approach for identifying unstable code

The above definition captures what unstable code is, but does not provide a way of finding unstable code, because it is difficult to reason about how an entire program will behave. As a proxy for a change in program behavior, Stack looks for code that can be transformed by some optimizer O under C but not under C*. In particular, Stack does this using a two-phase scheme:

1. run O without taking advantage of undefined behavior, which resembles optimizations under C*; and
2. run O again, this time taking advantage of undefined behavior, which resembles (more aggressive) optimizations under C.

If O optimizes extra code in the second phase, we assume the reason O did not do so in the first phase is because it would have changed the program's semantics under C*, and so Stack considers that code to be unstable.

Stack's optimizer-based approach to finding unstable code will miss unstable code that a specific optimizer cannot eliminate in the second phase, even if there exists some optimizer that could. This approach will also generate false reports if the optimizer is not aggressive enough in eliminating code in the first phase. Thus, one challenge in Stack's design is coming up with an optimizer that is sufficiently aggressive to minimize these problems.

In order for this approach to work, Stack requires an optimizer that can selectively take advantage of undefined behavior. To build such optimizers, we formalize what it means to "take advantage of undefined behavior" in §3.2.1, by introducing the *well-defined program assumption*, which captures C's assumption that programmers never write programs that invoke undefined behavior. Given an optimizer that can take explicit assumptions as input, Stack can turn on (or off) optimizations based on undefined behavior by supplying (or not) the well-defined program assumption to the optimizer. We build two aggressive optimizers that follow this approach: one that eliminates unreachable code (§3.2.2) and one that simplifies unnecessary computation (§3.2.3).

3.2.1 Well-defined program assumption

We formalize what it means to take advantage of undefined behavior in an optimizer as follows. Consider a program with input $\mathbf{x}$. Given a code fragment e, let $R_e(\mathbf{x})$ denote its *reachability condition*, which is *true* iff e will execute under input $\mathbf{x}$; and let $U_e(\mathbf{x})$ denote its *undefined behavior condition*, or UB condition for short, which indicates whether e exhibits undefined behavior on input $\mathbf{x}$, assuming C semantics (see Figure 3).

Both $R_e(\mathbf{x})$ and $U_e(\mathbf{x})$ are boolean expressions. For example, given a pointer dereference $*\mathtt{p}$ in expression e, one UB condition $U_e(\mathbf{x})$ is $p = \mathtt{NULL}$ (i.e., causing a null pointer dereference).

Intuitively, in a well-defined program to dereference pointer p, p must be non-null. In other words, the negation of its UB condition, $p \neq \mathtt{NULL}$, must hold whenever the expression executes. We generalize this below.

Definition 2 (Well-defined program assumption)**.** A code fragment e is well-defined on an input $\mathbf{x}$ iff executing e never triggers undefined behavior at e:

$$R_e(\mathbf{x}) \rightarrow \neg U_e(\mathbf{x}). \tag{1}$$

Furthermore, a program is well-defined on an input iff every fragment of the program is well-defined on that input, denoted as Δ:

$$\Delta(\mathbf{x}) = \bigwedge_{e \in \mathcal{P}} R_e(\mathbf{x}) \rightarrow \neg U_e(\mathbf{x}). \tag{2}$$

```
1: procedure ELIMINATE(P)
2:     for all e ∈ P do
3:         if R_e(x) is UNSAT then
4:             REMOVE(e)                          ▷ trivially unreachable
5:         else
6:             if R_e(x) ∧ Δ(x) is UNSAT then
7:                 REPORT(e)
8:                 REMOVE(e)                      ▷ unstable code eliminated
```

Figure 5: The elimination algorithm. It reports unstable code that becomes unreachable with the well-defined program assumption.

3.2.2 Eliminating unreachable code

The first algorithm identifies unstable statements that can be eliminated (i.e., $\mathcal{P} \rightsquigarrow \mathcal{P}[e/\varnothing]$ where e is a statement). For example, if reaching a statement requires triggering undefined behavior, then that statement must be unreachable. We formalize this below.

Theorem 1 (Elimination)**.** In a well-defined program $\mathcal{P}$, an optimizer can eliminate code fragment e, if there is no input $\mathbf{x}$ that both reaches e and satisfies the well-defined program assumption $\Delta(\mathbf{x})$:

$$\nexists \mathbf{x}\colon R_e(\mathbf{x}) \wedge \Delta(\mathbf{x}). \tag{3}$$

The boolean expression $R_e(\mathbf{x}) \wedge \Delta(\mathbf{x})$ is referred as the *elimination query*.

Proof. Assuming $\Delta(\mathbf{x})$ is *true*, if the elimination query $R_e(\mathbf{x}) \wedge \Delta(\mathbf{x})$ always evaluates to *false*, then $R_e(\mathbf{x})$ must be *false*, meaning that e must be unreachable. One can then safely eliminate e. □

Consider Figure 2 as an example. There is one input `tun` in this program. To pass the earlier `if` check, the reachability condition of the `return` statement is `!tun`. There is one UB condition `tun = NULL`, from the pointer dereference `tun->sk`, the reachability condition of which is *true*. As a result, the elimination query $R_e(\mathbf{x}) \wedge \Delta(\mathbf{x})$ for the `return` statement is:

$$\texttt{!tun} \wedge (\mathit{true} \rightarrow \neg(\texttt{tun} = \texttt{NULL})).$$

Clearly, there is no `tun` that satisfies this query. Therefore, one can eliminate the `return` statement.

With the above definition it is easy to construct an algorithm to identify unstable due to code elimination (see Figure 5). The algorithm first removes unreachable fragments without the well-defined program assumption, and then warns against fragments that become unreachable with this assumption. The latter are unstable code.

3.2.3 Simplifying unnecessary computation

The second algorithm identifies unstable expressions that can be optimized into a simpler form (i.e., $\mathcal{P} \rightsquigarrow \mathcal{P}[e/e']$ where e and e' are expressions). For example, if evaluating a boolean expression to *true* requires triggering undefined behavior, then that expression must evaluate to *false*. We formalize this below.

```
1: procedure SIMPLIFY(P, oracle)
2:     for all e ∈ P do
3:         for all e' ∈ PROPOSE(oracle, e) do
4:             if e(x) ≠ e'(x) ∧ R_e(x) is UNSAT then
5:                 REPLACE(e, e')
6:                 break                          ▷ trivially simplified
7:             if e(x) ≠ e'(x) ∧ R_e(x) ∧ Δ(x) is UNSAT then
8:                 REPORT(e)
9:                 REPLACE(e, e')
10:                break                          ▷ unstable code simplified
```

Figure 6: The simplification algorithm. It asks an oracle to propose a set of possible e', and reports if any of them is equivalent to e with the well-defined program assumption.

Theorem 2 (Simplification)**.** In a well-defined program $\mathcal{P}$, an optimizer can simplify expression e with another e', if there is no input $\mathbf{x}$ that evaluates $e(\mathbf{x})$ and $e'(\mathbf{x})$ to different values, while both reaching e and satisfying the well-defined program assumption $\Delta(\mathbf{x})$:

$$\exists e' \nexists \mathbf{x}\colon e(\mathbf{x}) \neq e'(\mathbf{x}) \wedge R_e(\mathbf{x}) \wedge \Delta(\mathbf{x}). \tag{4}$$

The boolean expression $e(\mathbf{x}) \neq e'(\mathbf{x}) \wedge R_e(\mathbf{x}) \wedge \Delta(\mathbf{x})$ is referred as the *simplification query*.

Proof. Assuming $\Delta(\mathbf{x})$ is *true*, if the simplification query $e(\mathbf{x}) \neq e'(\mathbf{x}) \wedge R_e(\mathbf{x}) \wedge \Delta(\mathbf{x})$ always evaluates to *false*, then either $e(\mathbf{x}) = e'(\mathbf{x})$, meaning that they evaluate to the same value; or $R_e(\mathbf{x})$ is *false*, meaning that e is unreachable. In either case, one can safely replace e with e'. □

Simplification relies on an oracle to propose e' for a given expression e. Note that there is no restriction on the proposed expression e'. In practice, it should be simpler than the original e since compilers tend to simplify code. STACK currently implements two oracles:

- Boolean oracle: propose *true* and *false* in turn for a boolean expression, enumerating possible values.
- Algebra oracle: propose to eliminate common terms on both sides of a comparison if one side is a subexpression of the other. It is useful for simplifying non-constant expressions, such as proposing $y < 0$ for $x + y < x$, by eliminating x from both sides.

As an example, consider simplifying $p + 100 < p$ using the boolean oracle, where p is a pointer. For simplicity assume its reachability condition is *true*. From Figure 3, the UB condition of $p + 100$ is $p_\infty + 100_\infty \notin [0, 2^n - 1]$. The boolean oracle first proposes *true*. The corresponding simplification query is:

$$\begin{aligned}&(p + 100 < p) \neq \mathit{true}\\ \wedge\ &\mathit{true} \wedge (\mathit{true} \rightarrow \neg(p_\infty + 100_\infty \notin [0, 2^n - 1])).\end{aligned}$$

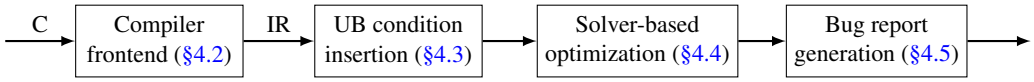


Figure 7: STACK's workflow. It invokes clang to convert a C/C++ program into LLVM IR, and then detects unstable code based on the IR.

Clearly, this is satisfiable. The boolean oracle then proposes *false*. This time the simplification query is:

$$(p + 100 < p) \neq \mathit{false} \\ \wedge \mathit{true} \wedge (\mathit{true} \rightarrow \neg(p_\infty + 100_\infty \notin [0, 2^n - 1])) .$$

Since there is no pointer p that satisfies this query, one can fold $p + 100 < p$ into *false*. §6.2.2 will show more examples of identifying unstable code using simplification.

With the above definition it is straightforward to construct an algorithm to identify unstable code due to simplification (see Figure 6). The algorithm consults an oracle for every possible simpler form e' for expression e. Similarly to elimination, it warns if it finds e' that is equivalent to e only with the well-defined program assumption.

3.3 Discussion

The model focuses on discarding unstable code by exploring two basic optimizations, elimination because of unreachability and simplification because of unnecessary computation. It is possible to exploit the well-defined program assumption in other forms. For example, instead of discarding code, some optimizations reorder instructions and produce unwanted code due to memory aliasing [47] or data races [2], which STACK does not model.

STACK implements two oracles, boolean and algebra, for proposing new expressions for simplification. One can extend it by introducing new oracles.

4 Design

This section describes the design of the STACK checker that detects unstable code by mimicking an aggressive compiler. A challenge in designing STACK is to make it scale to large programs. To address this challenge, STACK uses variants of the algorithms presented in §3 that work on individual functions. A further challenge is to avoid reporting false warnings for unstable code that is generated by the compiler itself, such as macros and inlined functions.

4.1 Overview

STACK works in four stages, as illustrated in Figure 7. In the first stage, a user prepends a script `stack-build` to the actual building command, such as:

```
% stack-build make
```

The script `stack-build` intercepts invocations to gcc and invokes clang instead to compile source code into the LLVM intermediate representation (IR). The remaining three stages work on the IR.

In the second stage, STACK inserts UB conditions listed in Figure 3 into the IR. In the third stage, it performs a solver-based optimization using a variant of the algorithms described in §3.2. In the fourth stage, STACK generates a bug report of unstable code discarded by the solver-based optimization, with the corresponding set of UB conditions. For example, for Figure 2 STACK links the null pointer check `!tun` to the earlier pointer dereference `tun->sk`.

4.2 Compiler frontend

STACK invokes clang to compile C-family source code to the LLVM IR for the rest of the stages. Furthermore, to detect unstable code across functions, it invokes LLVM to inline functions, and works on individual functions afterwards for better scalability.

A challenge is that STACK should focus on unstable code written by programmers, and ignore code generated by the compiler (e.g., from macros and inline functions). Consider the code snippet below:

```
#define IS_A(p) (p != NULL && p->tag == TAG_A)
p->tag == ...;
if (IS_A(p)) ...;
```

Assume `p` is a pointer passed from the caller. Ideally, STACK could inspect the callers and check whether `p` can be null. However, STACK cannot do this because it works on individual functions. STACK would consider the null pointer check `p != NULL` unstable due to the earlier dereference `p->tag`. In our experience, this causes a large number of false warnings, because programmers do not directly write the null pointer check but simply reuse the macro `IS_A`. To reduce false warnings, STACK ignores such compiler-generated code by tracking code origins, at the cost of missing possible bugs (see §4.6).

To do so, STACK implements a clang plugin to record the original macro for macro-expanded code in the IR during preprocessing and compilation. Similarly, it records the original function for inlined code in the IR during inlining. The final stage uses the recorded origin information to avoid generating bug reports for compiler-generated unstable code (see §4.5).

4.3 UB condition insertion

STACK implements the UB conditions listed in Figure 3. For each UB condition, STACK inserts a special function call into the IR at the corresponding instruction:

```
void bug_on(bool expr);
```

This function takes one boolean argument as the UB condition of the instruction.

It is straightforward to represent UB conditions as a boolean argument in the IR. For example, for a division x/y, Stack inserts bug_on($y = 0$) for division by zero. The next stage uses these bug_on calls to compute the well-defined program assumption.

4.4 Solver-based optimization

To detect unstable code, Stack runs the algorithms described in §3.2 in the following order:

- elimination,
- simplification with the boolean oracle, and
- simplification with the algebra oracle.

To implement these algorithms, Stack consults the Boolector solver [3] to decide satisfiability for elimination and simplification queries, as shown in (3) and (4). Both queries need to compute the terms $R_e(\mathbf{x}) \wedge \Delta(\mathbf{x})$. However, it is practically infeasible to precisely compute them for large programs. By definition, computing the reachability condition $R_e(\mathbf{x})$ requires inspecting all paths from the start of the program, and computing the well-defined program assumption $\Delta(\mathbf{x})$ requires inspecting the entire program for UB conditions. Neither scales to a large program.

To address this challenge, Stack computes approximate queries by limiting the computation to a single function. To describe the impact of this change, we use the following two terms. First, let $R'_e(\mathbf{x})$ denote fragment e's reachability condition from the start of current function; Stack replaces $R_e(\mathbf{x})$ with R'_e. Second, let $\mathrm{dom}(e)$ denote e's dominators [35: §7.3], the set of fragments that every execution path reaching e must have reached; Stack replaces the well-defined program assumption $\Delta(\mathbf{x})$ over the entire program with that over $\mathrm{dom}(e)$.

With these terms we describe the variant of the algorithms for identifying unstable code by computing approximate queries. Stack eliminates fragment e if the following query is unsatisfiable:

$$R'_e(\mathbf{x}) \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x}). \tag{5}$$

Similarly, Stack simplifies e into e' if the following query is unsatisfiable:

$$e(\mathbf{x}) \neq e'(\mathbf{x}) \wedge R'_e(\mathbf{x}) \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x}). \tag{6}$$

Appendix A provides a proof that using both approximate queries still correctly identifies unstable code.

Stack computes the approximate queries as follows. To compute the reachability condition $R'_e(\mathbf{x})$ within current function, Stack uses Tu and Padua's algorithm [48]. To compute the UB condition $\bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x})$, Stack collects them from the bug_on calls within e's dominators.

```
1: procedure MinUBCond(Q_e [= H ∧ ⋀_{d∈dom(e)} ¬U_d(x)])
2:     ubset ← ∅
3:     for all d ∈ dom(e) do
4:         Q'_e ← H ∧ ⋀_{d'∈dom(e)\{d}} ¬U_{d'}(x)
5:         if Q'_e is SAT then
6:             ubset ← ubset ∪ {U_d}
7:     return ubset
```

Figure 8: Algorithm for computing the minimal set of UB conditions that lead to unstable code given query Q_e for fragment e.

4.5 Bug report generation

Stack generates a bug report for unstable code based on the solver-based optimization. First, it inspects the recorded origin of each unstable code case in the IR, and ignores code that is generated by the compiler, rather than written by the programmer.

To help users understand the bug report, Stack reports the minimal set of UB conditions that make each report's code unstable [8], using the following greedy algorithm.

Let Q_e be the query with which Stack decided that fragment e is unstable. The query Q_e then must be unsatisfiable. From (5) and (6), we know that the query must be in the following form:

$$Q_e = H \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x}). \tag{7}$$

H denotes the term(s) excluding $\bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x})$ in Q_e. The goal is to find the minimal set of UB conditions that help make Q_e unsatisfiable.

To do so, Stack masks out each UB condition in e's dominators from Q_e individually to form a new query Q'_e; if the new query Q'_e becomes satisfiable, then the UB condition masked out is crucial for making fragment e unstable. The complete algorithm is listed in Figure 8.

4.6 Limitations

The list of undefined behavior Stack implements (see Figure 3) is incomplete. For example, it misses violations of strict aliasing [24: §6.5] and uses of uninitialized variables [24: §6.3.2.1]. We decided not to implement them because gcc already issues decent warnings for both cases. It would be easy to extend Stack to do so as well.

Moreover, since our focus is to find subtle code changes due to optimizations, we choose not to implement undefined behavior that occurs in the frontend. One example is evaluating `(x = 1) + (x = 2)`; this fragment has undefined behavior due to "unsequenced side effects" [24: §6.5/p2]. We believe that the frontend rather than the optimizer should be able to warn against such cases.

As discussed in §4.4, Stack implements approximation algorithms for better scalability, using approximate reachability and UB conditions. Stack may miss unstable code due to these approximations. As Stack consults a constraint solver with elimination and simplification queries,

	# bugs	pointer	null	integer	div	shift	buffer	abs	memcpy	free	realloc
Binutils	8	6	1			1					
e2fsprogs	3		1			1					1
FFmpeg+Libav	21	9	6	1	1	3	1				
FreeType	3	3									
GRUB	2		2								
HiStar [52]	3	1	2								
Kerberos	11	1	9	1							
libX11	2										2
libarchive	2			2							
libgcrypt	2				2						
Linux kernel	32	1	6	1	2	10	5		5	2	
Mozilla	3		2			1					
OpenAFS	11		6				4	1			
plan9port	3	1	1	1							
Postgres	9		1	7			1				
Python	5	5									
QEMU	4					3			1		
Ruby+Rubinius	2		1		1						
Sane	8				1					7	
uClibc	2			2							
VLC	2						2				
Xen	3	1	1			1					
Xpdf	9			8		1					
others (⋆)	10	1	5			2	1		1		
all	160	29	44	23	7	23	14	1	7	9	3

(⋆) Bionic, Dune [1], file, GMP, Mosh [51], MySQL, OpenSSH, OpenSSL, PHP, Wireshark.

Figure 9: New bugs identified by STACK. We also break down the number of bugs by undefined behavior from Figure 3: "pointer" (pointer overflow), "null" (null pointer dereference), "integer" (signed integer overflow), "div" (division by zero), "shift" (oversized shift), "buffer" (buffer overflow), "abs" (absolute value overflow), "memcpy" (overlapped memory copy), "free" (use after free), and "realloc" (use after realloc).

STACK will also miss unstable code if the solver times out. See §6.6 for a completeness evaluation.

STACK reports false warnings when it flags redundant code as unstable, as programmers sometimes simply write useless checks that have no effects (see §6.2.4). Note that even though such redundant code fragments are false warnings, discarding them is allowed by the specification.

5 Implementation

We implemented STACK using the LLVM compiler framework [30] and the Boolector solver [3]. STACK consists of approximately 4,000 lines of C++ code.

6 Evaluation

This section answers the following questions:

- Is STACK useful for finding new bugs? (§6.1)
- What kinds of unstable code does STACK find? (§6.2)
- How precise are STACK's bug reports? (§6.3)
- How long does STACK take to analyze a large system? (§6.4)
- How prevalent is unstable code in real systems, and what undefined behavior causes it? (§6.5)
- What unstable code does STACK miss? (§6.6)

6.1 New bugs

From July 2012 to March 2013, we periodically applied STACK to systems software written in C/C++ to identify unstable code. The systems STACK analyzed are listed in Figure 9, and include OS kernels, virtual machines, databases, multimedia encoders/decoders, language runtimes, and security libraries. Based on STACK's bug reports, we submitted patches to the corresponding developers. The developers confirmed and fixed 160 new bugs. The results show that unstable code is widespread, and that STACK is useful for identifying unstable code.

We also break down the bugs by type of undefined behavior. The results show that several kinds of undefined behavior contribute to the unstable code bugs.

6.2 Analysis of bug reports

This subsection reports our experience of finding and fixing unstable code with the aid of STACK. We manu-

```
int64_t arg1 = ...;
int64_t arg2 = ...;
if (arg2 == 0)
    ereport(ERROR, ...);
int64_t result = arg1 / arg2;
if (arg2 == -1 && arg1 < 0 && result <= 0)
    ereport(ERROR, ...);
```

Figure 10: An invalid signed division overflow check in Postgres, where the division precedes the check. A malicious SQL query will crash it on x86-64 by exploiting signed division overflow.

```
char buf[15]; /* filled with data from user space */
unsigned long node;
char *nodep = strchr(buf, '.') + 1;
if (!nodep)
    return -EIO;
node = simple_strtoul(nodep, NULL, 10);
```

Figure 11: An incorrect null pointer check in the Linux sysctl implementation for `/proc/sys/net/decnet/node_address`. A correct null check should test the result of `strchr`, rather than that plus one, which is always non-null.

ally classify Stack's bug reports into the following four categories based on the impact:

- non-optimization bugs, causing problems regardless of optimizations;
- urgent optimization bugs, where existing compilers are known to cause problems with optimizations turned on, but not with optimizations turned off;
- time bombs, where no known compilers listed in §2.3 cause problems with optimizations, though Stack does and future compilers may do so as well; and
- redundant code: false warnings, such as useless checks that compilers can safely discard.

The rest of this subsection illustrates each category using examples from Stack's bug reports. All the bugs described next were previously unknown but now have been confirmed and fixed by the corresponding developers.

6.2.1 Non-optimization bugs

Non-optimization bugs are unstable code that causes problems even without optimizations, such as the null pointer dereference bug shown in Figure 2, which directly invokes undefined behavior.

To illustrate the subtle consequences of invoking undefined behavior, consider the implementation of the 64-bit signed division operator for SQL in the Postgres database, as shown in Figure 10. The code first rejects the case where the divisor is zero. Since 64-bit integers range from -2^{63} to $2^{63}-1$, the only overflow case is $-2^{63}/-1$, where the expected quotient 2^{63} exceeds the range and triggers undefined behavior. The Postgres developers incorrectly assumed that the quotient must wrap around to -2^{63} in this case, as in some higher-level languages (e.g., Java), and tried to catch it by examining the overflowed quotient *after* the division, using the following check:

```
arg2 == -1 && arg1 < 0 && arg1 / arg2 <= 0.
```

Stack identifies this check as unstable code: the division implies that the overflow must *not* occur to avoid undefined behavior, and thus the overflow check after the division must be *false*.

While signed division overflow is undefined behavior in C, the corresponding x86-64 instruction `IDIV` traps on overflow. One can exploit this to crash the database server on x86-64 by submitting a SQL query that invokes $-2^{63}/-1$, such as:

```
SELECT ((-9223372036854775808)::int8) / (-1);
```

Interestingly, we notice that the Postgres developers tested the $-2^{63}/-1$ crash in 2006, but incorrectly concluded that this "seemed OK" [34]. We believe the reason is that they tested Postgres on x86-32, where there was no 64-bit `IDIV` instruction. In that case, the compiler would generate a call to a library function `lldiv` for 64-bit signed division, which returns -2^{63} for $-2^{63}/-1$ rather than a hardware trap. The developers hence overlooked the crash issue.

To fix this bug, we submitted a straightforward patch that checks whether `arg1` is -2^{63} and `arg2` is -1 before `arg1/arg2`. However, the Postgres developers insisted on their own fix. Particularly, instead of directly comparing `arg1` with -2^{63}, they chose the following check:

```
arg1 != 0 && (-arg1 < 0) == (arg1 < 0).
```

Stack identifies this check as unstable code for similar reasons: the negation `-arg1` implies that `arg1` cannot be -2^{63} to avoid undefined behavior, and thus the check must be *false*. We will further analyze this check in §6.2.3.

By identifying unstable code, Stack is also useful for uncovering programming errors that do not directly invoke undefined behavior. Figure 11 shows an incorrect null pointer check from the Linux kernel. The intention of this check was to reject a network address without any dots. Since `strchr(buf, '.')` returns null if it cannot find any dots in `buf`, a correct check should check whether its result is null, rather than that plus one. One can bypass the check `!nodep` with a malformed network address from user space and trigger an invalid read at page zero. Stack identifies the check `!nodep` as unstable code, because under the no-pointer-overflow assumption `nodep` (a pointer plus one) must be non-null.

6.2.2 Urgent optimization bugs

Urgent optimization bugs are unstable code that existing compilers already optimize to cause problems. §2.2 described a set of examples where compilers either discard the unstable code or rewrite it into some vulnerable form.

To illustrate the consequences, consider the code snippet from FFmpeg/Libav for parsing Adobe's Action Message Format, shown in Figure 12. The parsing code starts

```
const uint8_t *data     = /* buffer head */;
const uint8_t *data_end = /* buffer tail */;
int size = bytestream_get_be16(&data);
if (data + size >= data_end || data + size < data)
    return -1;
data += size;
...
int len = ff_amf_tag_size(data, data_end);
if (len < 0 || data + len >= data_end
            || data + len < data)
    return -1;
data += len;
/* continue to read data */
```

Figure 12: Unstable bounds checks in the form `data + x < data` from FFmpeg/Libav, which gcc optimizes into $x < 0$.

```
void pdec(io *f, int k) {
    if (k < 0) {          /* print negative k */
        if (-k >= 0) {    /* not INT_MIN?     */
            pchr(f, '-'); /* print minus      */
            pdec(f, -k);  /* print -k         */
            return;
        }
        ...               /* print INT_MIN    */
        return;
    }
    ...                   /* print positive k */
}
```

Figure 13: An unstable integer check in plan9port. The function `pdec` prints a signed integer `k`; gcc optimizes the check `-k >= 0` into *true* when it learns that `k` is negative, leading to an infinite loop if the input `k` is `INT_MIN`.

with two pointers, `data` pointing to the head of the input buffer, and `data_end` pointing to one past the end. It first reads in an integer `size` from the input buffer, and fails if the pointer `data + size` falls out of the bounds of the input buffer (i.e., between `data` and `data_end`). The intent of the check `data + size < data` is to reject a large `size` that causes `data + size` to wrap around to a smaller pointer and bypass the earlier check `data + size >= data_end`. The parsing code later reads in another integer `len` and performs similar checks.

Stack identifies the two pointer overflow checks in the form `data + x < data` as unstable code, where x is a signed integer (e.g., `size` and `len`). Specifically, with the algebra oracle Stack simplifies the check `data + x < data` into $x < 0$, and warns against this change. Note that this is slightly different from Figure 1: x is a signed integer, rather than unsigned, so the check is not always *false* under the well-defined program assumption.

Both gcc and clang perform similar optimizations, by rewriting `data + x < data` into $x < 0$. As a result, a large `size` or `len` from malicious input is able to bypass the checks, leading to an out-of-bounds read. A correct fix is to replace `data + x >= data_end || data + x < data` with `x >= data_end - data`, which is simpler and also avoids invoking undefined behavior; one should also add the check $x < 0$ if x can be negative.

```
int64_t arg1 = ...;
if (arg1 != 0 && ((-arg1 < 0) == (arg1 < 0)))
    ereport(ERROR, ...);
```

Figure 14: A time bomb in Postgres. The intention is to check whether `arg1` is the most negative value -2^{n-1}, similar to Figure 13.

```
struct p9_client *c = ...;
struct p9_trans_rdma *rdma = c->trans;
...
if (c)
    c->status = Disconnected;
```

Figure 15: Redundant code from the Linux kernel, where the caller of this code snippet ensures that `c` must be non-null and the null pointer check against `c` is always *true*.

Figure 13 shows an urgent optimization bug that leads to an infinite loop from plan9port. The function `pdec` is used to print a signed integer k; if k is negative, the code prints the minus symbol and then invokes `pdec` again with the negation $-k$. Assuming k is an n-bit integer, one special case is k being -2^{n-1} (i.e., `INT_MIN`), the negation of which is undefined. The programmers incorrectly assumed that `-INT_MIN` would wrap around to `INT_MIN` and remain negative, so they used the check $-k >= 0$ to filter out `INT_MIN` when k is known to be negative.

Stack identifies the check $-k >= 0$ as unstable code; gcc also optimizes the check into *true* as it learns that k is negative from the earlier $k < 0$. Consequently, invoking `pdec` with `INT_MIN` will lead an infinite loop, printing the minus symbol repeatedly. A simple fix is to replace $-k >= 0$ with a safe form k `!= INT_MIN`.

6.2.3 Time bombs

A time bomb is unstable code that is harmless at present, since no compiler listed in §2.3 can currently optimize it. But this situation may change over time. §2.3 already showed how past compiler changes trigger time bombs to become urgent optimization bugs. §6.2.1 illustrated how a time bomb in Postgres emerged as the x86 processor evolved: the behavior of 64-bit signed division on overflow changed from silent wraparound to trap, allowing one to crash the database server with malicious SQL queries.

Figure 14 shows a time bomb example from Postgres. As mentioned in §6.2.1, the Postgres developers chose this approach to check whether `arg1` is -2^{63} without using the constant value of -2^{63}; their assumption was that the negation of a non-zero integer would have a different sign unless it is -2^{63}.

The code currently works; the time bomb does not go off, and does not cause any problems, unlike its "equivalent" form in Figure 13. This luck relies on the fact that no production compilers discard it. Nonetheless, Stack identifies the check as unstable code, and we believe that some research compilers such as Bitwise [43] already discard the check. Relying on compilers to not optimize time

	build time	analysis time	# files	# queries	# query timeouts
Kerberos	1 min	2 min	705	79,547	2 (0.003%)
Postgres	1 min	11 min	770	229,624	1,131 (0.493%)
Linux kernel	33 min	62 min	14,136	3,094,340	1,212 (0.039%)

Figure 16: STACK's performance numbers when running it against Kerberos, Postgres, and the Linux kernel, including the build time, the analysis time, the number of files, the number of total queries STACK made, and the number of queries that timed out.

bombs for system security is risky, and we recommend fixing problems flagged by STACK to avoid this risk.

6.2.4 Redundant code

Figure 15 shows an example of redundant code from the Linux kernel. STACK identifies the null pointer check against the pointer `c` in the `if` condition as unstable code, due to the earlier dereference `c->trans`. The caller of the code snippet ensures that the pointer `c` must be non-null, so the check is always *true*. Our experience shows that redundant code comprises only a small portion of unstable code that STACK reports (see §6.3).

Depending on their coding conventions, it is up to programmers to decide whether to keep redundant code. Based on the feedback from STACK's users, we have learned that programmers often prefer to remove such redundant checks or convert them to assertions for better code quality, even if they are not real bugs.

6.3 Precision

To understand the precision of STACK's results, we further analyzed every bug report STACK produced for Kerberos and Postgres. The results below show that STACK has a low rate of false warnings (i.e., redundant code).

Kerberos. STACK reported 11 bugs in total, all of which were confirmed and fixed by the developers. In addition, the developers determined that one of them was remotely exploitable and requested a CVE identifier (CVE-2013-1415) for this bug. After the developers fixed these bugs, STACK produced zero reports.

Postgres. STACK reported 68 bugs in total. The developers promptly fixed 9 of them after we demonstrated how to crash the database server by exploiting these bugs, as described in §6.2.1. We further discovered that Intel's icc and PathScale's pathcc compilers discarded 29 checks, which STACK identified as unstable code (i.e., urgent optimization bugs), and reported these problems to the developers. At the writing of this paper, the strategies for fixing them are still under discussion.

STACK found 26 time bombs (see §6.2.3 for one example); we did not submit patches to fix these time bombs given the developers' hesitation in fixing urgent optimization bugs. STACK also produced 4 bug reports that identified redundant code, which did not need fixing.

algorithm	# reports	# packages
elimination	23,969	2,079
simplification (boolean oracle)	47,040	2,672
simplification (algebra oracle)	871	294

Figure 17: Number of reports generated by each of STACK's algorithms from §3.2 for all Debian Wheezy packages, and the number of packages for which at least one such report was generated.

6.4 Performance

To measure the running time of STACK, we ran it against Kerberos, Postgres, and the Linux kernel (with all modules enabled), using their source code from March 23, 2013. The experiments were conducted on a 64-bit Ubuntu Linux machine with an Intel Core i7-980 3.3 GHz CPU and 24 GB of memory. The processor has 6 cores, and each core has 2 hardware threads.

STACK built and analyzed each package using 12 processes in parallel. We set a timeout of 5 seconds for each query to the solver (including computing the UB condition set as described in §4.5). Figure 16 lists the build time, the analysis time, the number of files, the number of total queries to the solver, and the number of query timeouts. The results show that STACK can finish analyzing a large system within a reasonable amount of time.

We noticed a small number of solver timeouts (less than 0.5%) due to complex reachability conditions, often at the end of a function. STACK would miss unstable code in such cases. To avoid this, one can increase the timeout.

6.5 Prevalence of unstable code

We applied STACK to all 17,432 packages in the Debian Wheezy archive as of March 24, 2013. STACK checked 8,575 of them that contained C/C++ code. Building and analyzing these packages took approximately 150 CPU-days on Intel Xeon E7-8870 2.4 GHz processors.

For 3,471 out of these 8,575 packages, STACK detected at least one instance of unstable code. This suggests that unstable code is a widespread problem.

Figure 17 shows the number of reports generated by each of STACK's algorithms. These results suggest that they are all useful for identifying unstable code.

Each of STACK's reports contains a set of UB conditions that cause the code to be unstable. Figure 18 shows the number of times each kind of UB condition showed up in a report. These numbers confirm that many kinds of undefined behavior lead to unstable code in practice.

UB condition	# reports	# packages
null pointer dereference	59,230	2,800
buffer overflow	5,795	1,064
signed integer overflow	4,364	780
pointer overflow	3,680	614
oversized shift	594	193
aliasing	330	70
overlapping memory copy	227	47
division by zero	226	95
use after free	156	79
other libc (cttz, ctlz)	132	7
absolute value overflow	86	23
use after realloc	22	10

Figure 18: Number of reports that involve each of STACK's UB conditions from Figure 3 for all Debian Wheezy packages, and the number of packages for which at least one such report was generated.

As described in §4.5, STACK computes a minimal set of UB conditions necessary for each instance of unstable code. Most unstable code reports (69,301) were the result of just one UB condition, but there were also 2,579 reports with more than one UB condition, and there were even 4 reports involving eight UB conditions. These numbers confirm that some unstable code is caused by multiple undefined behaviors, which suggests that automatic tools such as STACK are necessary to identify them. Programmers are unlikely to find them by manual inspection.

6.6 Completeness

STACK is able to identify all the unstable code examples described in §2.3. However, it is difficult to know precisely how much unstable code STACK would miss in general. Instead we analyze what kind of unstable code STACK misses. To do so, we collected all examples from Regehr's "undefined behavior consequences contest" winners [40] and Wang et al.'s undefined behavior survey [49] as a benchmark, a total of ten tests from real systems.

STACK identified unstable code in seven out of the ten tests. STACK missed three for the following reasons. As described in §4.6, STACK missed two because we chose not to implement their UB conditions for violations of strict aliasing and uses of uninitialized variables; it would be easy to extend STACK to do so. The other case STACK missed was due to approximate reachability conditions, also mentioned in §4.6.

7 Related work

To the best of our knowledge, we present the first definition and static checker to find unstable code, but we build on several pieces of related work. In particular, earlier surveys [26, 41, 49] and blog posts [29, 39, 40] collect examples of unstable code, which motivated us to tackle this problem. We were also motivated by related techniques that can help with addressing unstable code, which we discuss next.

Testing strategies. Our experience with unstable code shows that in practice it is difficult for programmers to notice certain critical code fragments disappearing from the running system as they are silently discarded by the compiler. Maintaining a comprehensive test suite may help catch "vanished" code in such cases, though doing so often requires a substantial effort to achieve high code coverage through manual test cases. Programmers may also need to prepare a variety of testing environments as unstable code can be hardware- and compiler-dependent.

Automated tools such as KLEE [4] can generate test cases with high coverage using symbolic execution. These tools, however, often fail to model undefined behavior correctly. Thus, they may interpret the program differently from the language standard and miss bugs. Consider a check $x + 100 < x$, where x is a signed integer. KLEE considers $x + 100$ to wrap around given a large x; in other words, the check catches a large x when executing in KLEE, even though gcc discards the check. Therefore, to detect unstable code, these tools need to be augmented with a model of undefined behavior, such as the one we proposed in this paper.

Optimization strategies. We believe that programmers should avoid undefined behavior, and we provide suggestions for fixing unstable code in §6.2. However, overly aggressive compiler optimizations are also responsible for triggering these bugs. Traditionally, compilers focused on producing fast and small code, even at the price of sacrificing security, as shown in §2.2. Compiler writers should rethink optimization strategies for generating secure code.

Consider $x + 100 < x$ with a signed integer x again. The language standard does allow compilers to consider the check to be *false* and discard it. In our experience, however, it is unlikely that the programmer intended the code to be removed. A programmer-friendly compiler could instead generate efficient overflow checking code, for example, by exploiting the overflow flag available on many processors after evaluating $x + 100$. This strategy, also allowed by the language standard, produces more secure code than discarding the check. Alternatively, the compiler could produce warnings when exploiting undefined behavior in a potentially surprising way [19].

Currently, gcc provides several options to alter the compiler's assumptions about undefined behavior, such as

- `-fwrapv`, assuming signed integer wraparound for addition, subtraction, and multiplication;
- `-fno-strict-overflow`, assuming pointer arithmetic wraparound in addition to `-fwrapv`; and
- `-fno-delete-null-pointer-checks` [44], assuming unsafe null pointer dereferences.

These options can help reduce surprising optimizations, at the price of generating slower code. However, they cover an incomplete set of undefined behavior that may cause unstable code (e.g., no options for shift or division). Another downside is that these options are specific to gcc; other compilers may not support them or interpret them in a different way [49].

Checkers. Many existing tools can detect undefined behavior as listed in Figure 3. For example, gcc provides the `-ftrapv` option to insert run-time checks for signed integer overflows [42: §3.18]; IOC [11] (now part of clang's sanitizers [9]) and KINT [50] cover a more complete set of integer errors; Saturn [12] finds null pointer dereferences; several dedicated C interpreters such as kcc [14] and Frama-C [5] perform checks for undefined behavior. See Chen et al.'s survey [6] for a summary.

In complement to these checkers that directly target undefined behavior, STACK finds unstable code that becomes dead due to undefined behavior. In this sense, STACK can be considered as a generalization of Engler et al.'s inconsistency cross-checking framework [12, 16]. STACK, however, supports more expressive assumptions, such as pointer and integer operations.

Language design. Language designers may reconsider whether it is necessary to declare certain constructs as undefined behavior, since reducing undefined behavior in the specification is likely to avoid unstable code. One example is left-shifting a signed 32-bit one by 31 bits. This is undefined behavior [24: §6.5.7], even though the result is consistently `0x80000000` on most modern processors. The committee for the C++ language standard is already considering this change [33].

8 Conclusion

This paper presented the first systematic study of unstable code, an emerging class of system defects that manifest themselves when compilers discard code due to undefined behavior. Our experience shows that unstable code is subtle and often misunderstood by system programmers, that unstable code prevails in systems software, and that many popular compilers already perform unexpected optimizations, leading to misbehaving or vulnerable systems. We introduced a new model for reasoning about unstable code, and developed a static checker, STACK, to help system programmers identify unstable code. We hope that compiler writers will also rethink optimization strategies against unstable code. Finally, we hope this paper encourages language designers to be careful with using undefined behavior in the language specification. All STACK source code is publicly available at http://css.csail.mit.edu/stack/.

Acknowledgments

We thank Haogang Chen, Victor Costan, Li Lu, John Regehr, Jesse Ruderman, Tom Woodfin, the anonymous reviewers, and our shepherd, Shan Lu, for their help and feedback. This research was supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, and by NSF award CNS-1053143.

A Correctness of approximation

As discussed in §3.2, STACK performs an optimization if the corresponding query Q is unsatisfiable. Using an approximate query Q' yields a correct optimization if Q' is weaker than Q (i.e., $Q \rightarrow Q'$): if Q' is unsatisfiable, which enables the optimization, the original query Q must also be unsatisfiable.

To prove the correctness of approximation, it suffices to show that the approximate elimination query (5) is weaker than the original query (3); the simplification queries (6) and (4) are similar. Formally, given code fragment e, it suffices to show the following:

$$R_e(\mathbf{x}) \wedge \Delta(\mathbf{x}) \rightarrow R'_e(\mathbf{x}) \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x}). \tag{8}$$

Proof. Since e's dominators are a subset of the program, the well-defined program assumption over $\mathrm{dom}(e)$ must be weaker than $\Delta(\mathbf{x})$ over the entire program:

$$\Delta(\mathbf{x}) \rightarrow \bigwedge_{d \in \mathrm{dom}(e)} (R_d(\mathbf{x}) \rightarrow \neg U_d(\mathbf{x})). \tag{9}$$

From the definition of $\mathrm{dom}(e)$, if fragment e is reachable, then its dominators must be reachable as well:

$$\forall d \in \mathrm{dom}(e) \colon R_e(\mathbf{x}) \rightarrow R_d(\mathbf{x}). \tag{10}$$

Combining (9) and (10) gives:

$$\Delta(\mathbf{x}) \rightarrow (R_e(\mathbf{x}) \rightarrow \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x})). \tag{11}$$

With $R_e(\mathbf{x})$, we have:

$$R_e(\mathbf{x}) \wedge \Delta(\mathbf{x}) \rightarrow R_e(\mathbf{x}) \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x}). \tag{12}$$

By definition $R_e(\mathbf{x}) \rightarrow R'_e(\mathbf{x})$, so (12) implies (8). □

References

[1] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–348, Hollywood, CA, Oct. 2012.

[2] H.-J. Boehm. Threads cannot be implemented as a library. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 261–268, Chicago, IL, June 2005.

[3] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177, York, UK, Mar. 2009.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[5] G. Canet, P. Cuoq, and B. Monate. A value analysis for C programs. In *Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 123–124, Edmonton, Canada, Sept. 2009.

[6] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, Shanghai, China, July 2011.

[7] Chromium. Issue 12079010: Avoid undefined behavior when checking for pointer wraparound, 2013. https://codereview.chromium.org/12079010/.

[8] A. Cimatti, A. Griggio, and R. Sebastiani. Computing small unsatisfiable cores in satisfiability modulo theories. *Journal of Artificial Intelligence Research*, 40:701–728, 2011.

[9] Clang. *Clang Compiler User's Manual: Controlling Code Generation*, 2013. http://clang.llvm.org/docs/UsersManual.html#controlling-code-generation.

[10] J. Corbet. Fun with NULL pointers, part 1, July 2009. http://lwn.net/Articles/342330/.

[11] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 760–770, Zurich, Switzerland, June 2012.

[12] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 435–445, San Diego, CA, June 2007.

[13] C. R. Dougherty and R. C. Seacord. C compilers may silently discard some wraparound checks. Vulnerability Note VU#162289, US-CERT, 2008. http://www.kb.cert.org/vuls/id/162289.

[14] C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *Proceedings of the 39th ACM Symposium on Principles of Programming Languages (POPL)*, pages 533–544, Philadelphia, PA, Jan. 2012.

[15] C. Ellison and G. Roşu. Defining the undefinedness of C. Technical report, University of Illinois, Apr. 2012. http://hdl.handle.net/2142/30780.

[16] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 57–72, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[17] GCC. Bug 30475 - `assert(int+100 > int)` optimized away, 2007. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=30475.

[18] GCC. Bug 49820 - explicit check for integer negative after abs optimized away, 2011. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=49820.

[19] GCC. Bug 53265 - warn when undefined behavior implies smaller iteration count, 2013. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=53265.

[20] D. Gohman. The nsw story, Nov. 2011. http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-November/045730.html.

[21] IBM. *Power ISA Version 2.06 Revision B, Book I: Power ISA User Instruction Set Architecture*, July 2010.

[22] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference, A–Z*, Jan. 2013.

[23] ISO/IEC JTC1/SC22/WG14. *Rationale for International Standard - Programming Languages - C*, Apr. 2003.

[24] ISO/IEC JTC1/SC22/WG14. *ISO/IEC 9899:2011, Programming languages - C*, Dec. 2011.

[25] B. Jack. Vector rewrite attack: Exploitable NULL pointer vulnerabilities on ARM and XScale architectures. White paper, Juniper Networks, May 2007.

[26] R. Krebbers and F. Wiedijk. Subtleties of the ANSI/ISO C standard. Document N1639, ISO/IEC JTC1/SC22/WG14, Sept. 2012.

[27] T. Lane. Anyone for adding `-fwrapv` to our standard CFLAGS?, Dec. 2005. http://www.postgresql.org/message-id/1689.1134422394@sss.pgh.pa.us.

[28] T. Lane. Re: gcc versus division-by-zero traps, Sept. 2009. http://www.postgresql.org/message-id/19979.1251998812@sss.pgh.pa.us.

[29] C. Lattner. What every C programmer should know about undefined behavior, May 2011. http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html.

[30] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, Palo Alto, CA, Mar. 2004.

[31] Linux kernel. Bug 14287 - ext4: fixpoint divide exception at `ext4_fill_super`, 2009. https://bugzilla.kernel.org/show_bug.cgi?id=14287.

[32] D. MacKenzie, B. Elliston, and A. Demaille. *Autoconf: Creating Automatic Configuration Scripts for version 2.69*. Free Software Foundation, Apr. 2012.

[33] W. M. Miller. C++ standard core language defect reports and accepted issues, issue 1457: Undefined behavior in left-shift, Feb. 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#1457.

[34] B. Momjian. Re: Fix for Win32 division involving `INT_MIN`, June 2006. http://www.postgresql.org/message-id/200606090240.k592eUj23952@candle.pha.pa.us.

[35] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[36] D. Novillo. A propagation engine for GCC. In *Proceedings of the 2005 GCC & GNU Toolchain Developers' Summit*, pages 175–184, Ottawa, Canada, June 2005.

[37] Python. Issue 17016: _sre: avoid relying on pointer overflow, 2013. http://bugs.python.org/issue17016.

[38] S. Ranise, C. Tinelli, and C. Barrett. QF_BV logic, 2013. http://smtlib.cs.uiowa.edu/logics/QF_BV.smt2.

[39] J. Regehr. A guide to undefined behavior in C and C++, July 2010. http://blog.regehr.org/archives/213.

[40] J. Regehr. Undefined behavior consequences contest winners, July 2012. http://blog.regehr.org/archives/767.

[41] R. C. Seacord. Dangerous optimizations and the loss of causality, Feb. 2010. https://www.securecoding.cert.org/confluence/download/attachments/40402999/Dangerous+Optimizations.pdf.

[42] R. M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection for GCC 4.8.0*. Free Software Foundation, 2013.

[43] M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 108–120, Vancouver, Canada, June 2000.

[44] E. Teo. [PATCH] add `-fno-delete-null-pointer-checks` to gcc `CFLAGS`, July 2009. https://lists.ubuntu.com/archives/kernel-team/2009-July/006609.html.

[45] J. Tinnes. Bypassing Linux NULL pointer dereference exploit prevention (`mmap_min_addr`), June 2009. http://blog.cr0.org/2009/06/bypassing-linux-null-pointer.html.

[46] L. Torvalds. Re: [patch] CFS scheduler, -v8, May 2007. https://lkml.org/lkml/2007/5/7/213.

[47] J. Tourrilhes. Invalid compilation without `-fno-strict-aliasing`, Feb. 2003. https://lkml.org/lkml/2003/2/25/270.

[48] P. Tu and D. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 414–423, Barcelona, Spain, July 1995.

[49] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Undefined behavior: What happened to my code? In *Proceedings of the 3rd Asia-Pacific Workshop on Systems*, Seoul, South Korea, July 2012.

[50] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving integer security for systems with Kint. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 163–177, Hollywood, CA, Oct. 2012.

[51] K. Winstein and H. Balakrishnan. Mosh: An interactive remote shell for mobile clients. In *Proceedings of the 2012 USENIX Annual Technical Conference*, pages 177–182, Boston, MA, June 2012.

[52] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 263–278, Seattle, WA, Nov. 2006.

Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions

Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem*
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.

Abstract

Systems software such as OS kernels, embedded systems, and libraries must obey many rules for both correctness and performance. Common examples include "accesses to variable A must be guarded by lock B," "system calls must check user pointers for validity before using them," and "message handlers should free their buffers as quickly as possible to allow greater parallelism." Unfortunately, adherence to these rules is largely unchecked.

This paper attacks this problem by showing how system implementors can use *meta-level compilation* (MC) to write simple, system-specific compiler extensions that automatically check their code for rule violations. By melding domain-specific knowledge with the automatic machinery of compilers, MC brings the benefits of language-level checking and optimizing to the higher, "meta" level of the systems implemented in these languages. This paper demonstrates the effectiveness of the MC approach by applying it to four complex, real systems: Linux, OpenBSD, the Xok exokernel, and the FLASH machine's embedded software. MC extensions found roughly 500 errors in these systems and led to numerous kernel patches. Most extensions were less than a hundred lines of code and written by implementors who had a limited understanding of the systems checked.

1 Introduction

Systems software must obey many rules such as "check user permissions before modifying kernel data structures," "for speed, enforce mutual exclusion with spin locks rather than disabling interrupts," and "message handlers must free their buffer before completing." Code that does not obey these rules can degrade performance or crash the system.

There are several methods to find violations of system rules. A rigorous way is to build an abstract specification of the code and then use model checkers [23, 32] or theorem provers/checkers [2, 11, 25] to check that the specification is internally consistent. When applicable, formal verification finds errors that are difficult to detect by other means. However, specifications are difficult and costly to construct. Further, specifications do not necessarily mirror the code they abstract and, in practice, suffer from missing features and over-simplifications. While recent work has begun attacking these problems [6, 14], it is extremely rare for software to be verified.

The most common method used to detect rule violations is testing. Testing is simpler than verification. It also avoids the mirroring problems of formal verification by working with actual code rather than an abstraction of it. However, testing is dynamic, which has numerous disadvantages. First, the number of execution paths typically grows exponentially with code size. Thorough, precise testing requires writing many test cases to exercise these paths and drive the system into error states. The effort required to create these tests, and the time it takes to run them, scales with the amount of code. As a result, real systems have many paths that are rarely or never hit by testing and errors that manifest themselves only after days of continuous execution. Further, finding the cause of a test failure can be difficult, especially when the effect is a delayed system crash. Finally, testing requires running the tested code, which can create significant practical problems. For example, testing all device drivers in an OS requires acquiring possibly hundreds or thousands of devices and understanding how to thoroughly exercise them.

Another common method to detect rule violations is manual inspection. This method has the strength that it can consider all semantic levels and adapt to ad hoc coding conventions and system rules. Unfortunately, many systems have millions of lines of code

*This research was supported in part by DARPA contract MDA904-98-C-A933 and by a Terman Fellowship.

with deep, complex code paths. Reasoning about a single path can take minutes or sometimes, when dealing with concurrency, hours. Further, the reliability of manual inspection is erratic.

These methods leave implementors in an unfortunate situation. Verification is impractical for most systems. Testing misses many cases and makes diagnosis difficult. Manual inspection is unreliable and tedious. One possible alternative is to use static compiler analysis to find rule violations. Unlike verification, compilers work with the code itself, removing the need to write and maintain a specification. Unlike testing, static analysis can examine all execution paths for errors, even in code that cannot be conveniently executed. Further, a compiler analysis pass reduces the need to construct numerous test cases and scales from a single function to an entire system with little increase in manual effort.

Compilers can be used to enforce systems rules because many rules have a straightforward mapping to program source. Rule violations can be found by checking when source operations do not make sense at an abstract level. For example, ordering rules such as "interrupts must be enabled after being disabled" reduce to observing the order of function calls or idiomatic sequences of statements (in this case, a call to a disable interrupt function must be followed by a re-enable call).

The main barrier to a compiler checking or optimizing at this level is that while it must have a precise understanding of the semantics of its input code, it typically has no idea of the "meta" semantics of the software system this code constructs. Thus, it cannot check many properties inexpressible (or just not expressed) in terms of the underlying language's type system. This leaves an unfortunate dichotomy. Implementors understand the semantics of the system operations they build and use but do not have the mechanisms to check or exploit these semantics automatically. Compilers have the machinery to do so, but their domain ignorance prevents them from exploiting it.

This paper shows how to automatically check systems rules using *meta-level compilation* (MC). MC attacks this problem by making it easy for implementors to extend compilers with lightweight, system-specific checkers and optimizers. Because these extensions can be written by system implementors themselves, they can take into account the ad hoc (sometimes bizarre) semantics of a system. Because they are compiler based, they also get the benefits of automatic static analysis.

In our MC system, implementors write extensions in a high-level state-machine language, *metal*. These extensions are dynamically linked into our extensible compiler, *xg++*, and applied down all flow paths in all functions in the program source input. They use language-based patterns to recognize operations that they care about. Then, when the input code matches these patterns, they detect rule violations by transitioning between states that allow or disallow other operations.

This paper's primary contribution is its demonstration that MC is a general, effective approach for finding system errors. Our most important results are:

1. MC checkers find serious errors in complex, real systems code. We present a series of extensions that found roughly 500 errors in four systems: the Linux 2.3.99 kernel, OpenBSD, the Xok exokernel [16], and the FLASH machine's embedded cache controller code [20]. Many errors were the worst type of systems bugs: those that crash the system, but only after it has been running continuously for days.

2. MC optimizers discover system-level opportunities that are difficult to find with manual inspection. While the main focus of this paper is error checking, MC extensions can also be used for optimization. Section 8 describes three FLASH-specific, MC optimizers that found hundreds of system-level optimization opportunities.

3. MC extensions are simple. The extensions mentioned above are typically less than a hundred lines of code.

A practical result of our experience with MC is that the majority of our extensions were written by programmers who had only a passing familiarity with the systems that they checked. Although writing code that obeys system rules can be quite difficult, these rules are easy to express. Thus, writing checkers for many of them is relatively straightforward.

This paper is laid out as follows. Section 2 discusses related work. Section 3 gives an overview of MC and the system we use to implement it. Section 4 applies the approach to the C `assert` macro and shows that even in such a limited domain, MC provides non-trivial benefits. Section 5 shows how to use MC to enforce ordering constraints such as checking that kernels verify user pointers before using them. Section 6 extends this to global, system-wide constraints. Section 7 is a more detailed case study in how we used MC to check Linux locking and interrupt disabling/re-enabling disciplines. Section 8 describes our FLASH optimizers, and Section 9 concludes.

2 Related Work

We proposed the initial idea of MC in [9] and provided a simple system, `magik` (based on the `lcc` ANSI C compiler [12]), for using it. While the original paper had many examples, it provided no experimental evaluation. This paper provides a more developed view of MC, a significantly easier-to-use and more powerful framework for building extensions, and an experimental demonstration of its effectiveness. Concurrently with this paper, we presented a detailed case study of applying MC to the FLASH system [4]. The 8 compiler extensions presented in that paper discovered 34 errors in FLASH code that could potentially crash the machine, such as message handlers that lost or double freed hardware message buffers and buffer race conditions. This paper's main difference is its demonstration that MC is a general technique by applying it to a variety of systems. Because of this broader scope, it lacks the detail in [4], but finds roughly a factor of ten more errors.

Below, we compare our work to efforts in high-level compilation, verification, and extensible compilers.

Higher-level compilation. Many projects have hard wired application-level information in compilers. These projects include: compiler-directed management of I/O [24]; the ERASER dynamic race detection checker [30]; ParaSoft's Insure++ [19], which can check for Unix system call errors; the use of static analysis to check for security errors in privileged programs [1]; and the GNU compilers' `-Wall` option, which warns about dangerous functions and questionable programming practices. Related to the checkers in this paper, Microsoft has an internal tool for finding a fixed set of coding violations in Windows device drivers [27] such as errors in handling 64-bit code and missing user pointer validity checks.

These projects use compiler support to analyze specific problems, whereas MC explicitly argues for the general use of compilers to check and optimize systems and provides an extensible framework for doing so. This extensibility enables detection of rule violations that are impossible to find without system-specific knowledge.

Systems for finding software errors. Most approaches to statically finding software errors center around either formal verification (as discussed in Section 1) or strong type checking.

Verification uses stronger analysis than MC extensions. However, MC extensions appear to be more generally effective. To the best of our knowledge, verification papers tend to find a small number of errors (typically 0-2), whereas the MC checkers in this paper found hundreds. Verification's lower bug counts seem largely due to the difficulty in writing specifications, which scales with code size. As a consequence, only small pieces of code are verified. In contrast, because MC operates directly on source code, it (like traditional compiler analyses) applies as easily to millions of lines of code as it does to only a few.

Two recent strong-typing systems are the extended static type checking (ESC) project [8] and Intrinsa's PREfix [15]. Both of these systems use stronger analyses than our approach. However, they only check for a fixed set of low-level errors (e.g., buffer overruns and null pointer references). Their lack of extensibility means that, with the exception of ESC's support for finding some class of race conditions, neither system can find the system-level errors that MC can detect.

LCLint [10] statically checks programmer source annotations to detect coding errors and abstraction barrier violations. Like ESC and Intrinsa, LCLint is not extensible, which prevents it from finding the errors that MC can find. Further, the source annotations that LCLint requires scale with code size, significantly increasing the manual effort needed to apply it.

Extensible compilation. There have been a number of "open compiler" systems that allow programmers to add analysis routines, usually modeled as extensions, that traverse the compiler's abstract syntax tree. These include Lord's `ctool` [22], which allows scheme extensions to walk over an abstract syntax tree for C, and Crew's Prolog-based ASTLOG [7], also used for C.

Lamping et al. [21] and Kiczales et al. [17] argue for pushing domain-specific information into compilation. They use meta-object protocols (MOPs) to allow programs to be augmented with a "meta" part that controls the base [17]. Such protocols are typically dynamic and have fairly limited analysis abilities. Shigeru Chiba's `Open C++` [3] provides a static MOP that allows users to extend the compilation process.

The extensions in these systems are mainly limited to syntax-based tree traversal or transformation and do not have data flow information. As a result, they seem to be both less powerful than MC extensions and more difficult to use. Our current, language-based approach is a dramatic improvement over our previous tree-based systems: extensions are 2-4 times smaller, have less bugs, and handle more cases. Further, to the best of our knowledge, `ctool`, ASTLOG, and `Open C++` provide no experimental results, making it difficult to evaluate their effectiveness.

At a lower-level, the ATOM object code modifi-

cation system [31] gives users the ability to modify object code in a clean, simple manner. By focusing on machine code, ATOM can be used in more situations than MC, which requires source code. However, while dynamic testing schemes [13, 30] are well served by object-level modifications, it would be difficult to perform our static checks without the semantic information available in the compiler.

Concurrently with our original work [9], Kiczales et al. [18] proposed "aspect oriented programming" (AOP) as a way of combining code that manages "aspects," such as synchronization, with code that needs them. AOP has the advantage of being integrated within a traditional language framework. It has the disadvantage that aspects have more limited scope than MC extensions, which survey the entire system as well as check rules difficult to enforce with an AOP framework (e.g., preventing kernel code from using floating point). Further, because AOP requires source modifications, retro-fitting it on the systems we check would be non-trivial.

3 Meta-level Compilation

Many systems constraints describe legal orderings of operations or specific contexts in which these operations can or cannot occur. Since the actions relevant to these rules are visible in program source, an MC compiler extension can check them by searching for the corresponding operations and verifying that they obey the given ordering and/or contextual restrictions. Table 1 gives a representative set of rule "templates" that can be checked in this manner along with examples. Many system rules that roughly follow these templates can be checked automatically. For example, an MC extension to enforce the contextual rule, "for speed, if a shared variable is not modified, protect it with read locks," can search for each write-lock critical section, examine all variable uses, and, if no stores occur to protected variables, demote the locks or suggest alternative usage.

3.1 Language Overview

In our implementation of MC, compiler extensions are written in a high-level, state-machine language, *metal* [5]. These extensions are dynamically linked into our extensible compiler, *xg++*(based on the GNU g++ compiler). After *xg++* translates each input function into its internal representation, the extensions are applied down every possible execution path in that function. The state machine part of the language can be viewed as syntactically similar to a "yacc" specification. Typically, SMs use patterns

```
{ #include "linux-includes.h" }
sm check_interrupts {
  // Variables
  // used in patterns
  decl { unsigned } flags;

  // Patterns
  // to specify enable/disable functions.
  pat enable = { sti(); }
             | { restore_flags(flags); } ;
  pat disable = { cli(); };

  // States
  // The first state is the initial state.
  is_enabled: disable ==> is_disabled
     | enable ==> { err("double enable"); }
     ;
  is_disabled: enable ==> is_enabled
     | disable ==> { err("double disable"); }
     // Special pattern that matches when the SM
     // hits the end of any path in this state.
     | $end_of_path$ ==>
        { err("exiting w/intr disabled!"); }
     ;
}
```

Figure 1: A *metal* SM to detect (1) when interrupts disabled using `cli` are not re-enabled using either `sti` or `restore_flags` and (2) duplicate enable/disable calls.

to search for interesting source code features, which, when matched, cause transitions between states. Patterns are written in an extended version of the base language (C++), and can match almost arbitrary language constructs such as declarations, expressions, and statements. Expressing patterns in the base language makes them both flexible and easy to use, since they closely mirror the source constructs they describe.

Figure 1 shows a stripped-down *metal* extension for Linux that checks that disabled interrupts are re-enabled or restored to their initial state upon exiting a function. Interrupts are disabled by calling the `cli()` procedure; they are enabled by calling `sti()` or restored using `restore_flags(flags)`, where the `flags` variable holds the interrupt state before the `cli()` was issued. Conceptually, the extension finds violations by checking that each call to disable interrupts has a matching enable call on all outgoing paths. As refinements, the extension warns of duplicate calls to these functions or non-sequitur calls (e.g., re-enabling without disabling). A more complete version of this checker, described in Section 7, found 82 errors in Linux code.

The extension tracks the interrupt status using

Rule template	Examples
"Never/always do X"	"Do not use floating point in the kernel." (§ 4.3) "Do not allocate large variables on the 6K byte kernel stack." (§ 4.3) "Do not send more than two messages per virtual network lane." "Allocate as much storage as an object needs." (§ 5.2)
"Do X rather than Y"	"Use memory mapped I/O rather than copying." "Avoid globally disabling interrupts."
"Always do X before/after Y"	"Check user pointers before using them in the kernel." (§ 5.1) "Handle operations that can fail (e.g., memory, disk block, virtual interrupt allocation)." (§ 5.2) "Re-enable interrupts after disabling them." (§ 7) "Release locks after acquiring them." (§ 7) "Check user permissions before modifying kernel data structures."
"Never do X before/after Y"	"Do not acquire lock A before B." "Do not use memory that has been freed." (§ 5.2) "Do not (deallocate an object, acquire/release a lock) twice." (§ 5.2 § 7) "Do not increment a module's reference count after calling a function that can sleep." (§ 6.3)
"In situation X, do (not do) Y"	"Protect all variable mutations with write locks." "If a system call fails, reverse all side-effect operations (deallocate memory, disk blocks, pages, unincrement reference counters)." (§ 5.2 § 6.3) "To avoid deadlock, while interrupts are disabled, do not call functions that can sleep." (§ 6.2)
"In situation X, do Y rather than Z"	"If a variable is not modified, protect it with read locks." "If code does not share data with interrupt handlers, then use spin locks rather than the more expensive interrupt disabling." "To save an instruction when setting a message opcode, xor in the new and old opcode rather than using assignment." (§ 8)

Table 1: Sample system rule templates and examples. Checkers for the rule are denoted by section number.

```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head *sh,
                int b_size) {
       struct buffer_head *bh;
       unsigned long flags;

       save_flags(flags);
       cli();
       if ((bh = sh->buffer_pool) == NULL)
              return NULL;
       sh->buffer_pool = bh->b_next;
       bh->b_size = b_size;
       restore_flags(flags);
       return bh;
}
```

Figure 2: Example code from the Linux 2.3.99 Raid 5 driver illustrating a real error caught by the extension. The SM will be applied down both paths in this function. The path ending with a return of `bh` is well formed and will be accepted. The path ending with the return of `NULL` is not, and will get a warning about not re-enabling interrupts.

two states, `is_enabled` and `is_disabled`. SMs start in the state mentioned in the first transition definition (here, `is_enabled`). Each state has a set of rules specifying a pattern, an optional state transition, and an optional action. Actions can be arbitrary C++ code. For a given state, *metal* checks pattern rules in lexical order. If any code matches the specified patterns, *metal* processes this matching code, sets the state to the new state (the token after the `==>` operator), and executes the action. In this example, `is_enabled` has two rules. The first, actionless rule searches for functions that disable interrupts using the `disable` pattern and transitions to the `is_disabled` state. The second rule searches for calls to functions that enable interrupts and gives a warning. Since it does not specify a transition state, the SM remains in the `is_enabled` state. If no pattern matches, the SM remains in the same state and continues down the current code path. The `flags` variable is a wild card that matches any expression of type `unsigned`. When it is matched, *metal* will put the matching expression in `flag`, which can then be used in an action. We use this feature in an extension discussed in Section 4.

To run this SM, it is first compiled with `mcc`, our *metal* compiler. It is then dynamically linked into *xg++* using a compile-time, command-line flag. When run on the Linux "RAID 5" driver buffer allocation code in Figure 2, it is pushed down both paths

in the function. The first path returns `NULL` when the buffer pool is empty (i.e., when the `if` statement fails); the other returns a buffer on successful allocation. The first path fails to re-enable interrupts, and this error [1] is caught and reported by the extension.

One way to get a feel for how costly it would be to manually perform the check our SM does automatically is that even when we showed an experienced Linux programmer the exact error in Figure 2, it took him over 20 minutes to examine a single call chain out of the nine leading to this function. Performing similar analysis for the other hundreds of thousands of lines of driver and kernel code seems impractical.

3.2 Practical issues

Metal SMs can specify whether they should be applied either down all paths (i.e., flow-sensitive) or linearly through the code (i.e., flow-insensitive). A simple implementation of flow-sensitive SMs could take exponential time in some cases. We use aggressive caching to prune redundant code paths where SM instances follow paths that join (e.g., if statements, loops) and reach the join point in the same state. Our caching is based on the fact that a deterministic SM applied to the same input in the same internal state must compute the same result. The system represents the state of an SM as a vector holding the value of its variables. For each node in the input flow-graph, it records the set of states in which it has been visited. If an SM arrives at a node in the same state as a previous instance, the system prunes it.

While caching was originally motivated by speed, perhaps its most important feature is that it provides a clean framework for computing loop "fixed points" transparently. When an SM has exhausted the set of states reachable within the loop (typically with two iterations), *metal* automatically stops traversing the loop. This fixed-point behavior depends on the SM having a finite (and small) number of states. We do not currently enforce this restriction.

The current *xg++* system does not integrate global analysis with the SM framework. Instead, it provides a library of routines to emit client-annotated flow graphs to a file, which can then be read and traversed. Section 6 gives an example of how we used this framework to compute the transitive closure of all possibly-sleeping functions. We are integrating these two passes.

[1] Amusingly, this interrupt disable bug would be masked by an immediate kernel segmentation fault since callers of this function dereference the returned pointer without checking whether the allocation succeeded.

3.3 Caveats

Most of our extensions are checkers rather than verifiers: they find bugs, but do not guarantee their absence. For example, their ignorance of aliases prevents them from asserting that many actions "cannot happen." In general, many compiler problems are undecidable, which places hard limits on the effectiveness of static analysis. Despite these limitations, as our results show, MC extensions are quite effective. We are currently investigating how to turn some classes of checkers into verifiers.

We mainly check systems we did not build. As a result, some rule violations we found might not be bugs because the code could use a non-obvious system feature that works correctly in a specific situation. We countered this danger in two ways. First, we sent our error logs to the system implementors of Linux, FLASH, and Xok for confirmation. However, while we got feedback on many errors, their sheer number meant that many did not receive careful examination. Second, we conservatively did not count many cases that were difficult to reason about. While our results may still contain mis-diagnoses, we would be surprised if these caused more than a few percentage points difference.

Several of our checkers produce a number of false positives (in the worst case, in Section 7, up to three per error). These are due to the limitations of both static analysis and our checkers, which primarily use simple local analyses. Usually these numbers can be reduced significantly by adding some amount of global analysis or system-specific knowledge. In almost all cases, each false positive can be suppressed with a single source annotation. Extensions can provide annotations by supplying a set of reserved functions that clients call to indicate that a specific source-level warning should be suppressed. As a refinement, checkers can detect bogus or erroneous annotations by warning when they are not needed.

Basing our MC system on a C++ compiler has caused difficulties when applying it to Linux and Xok. These systems aggressively assume C's more relaxed type system and use GNU extensions that are illegal in `g++`. Thus, while in theory MC can be applied to a system transparently, we had to modify Xok and Linux to remove GNU C constructs that are illegal in C++. We also modified the `g++` front-end to relax its type checking. To avoid this labor for other systems, we are currently finishing a `gcc`-based implementation of *xg++*. More generally, since the *metal* language has been designed to be shielded from both the underlying language and compiler, we plan to port it other languages and other compilers.

The remainder of this paper describes the extensions we implemented using *metal* and *xg++* and the results of applying the concept of meta-level compilation to real systems.

4 A Simple Meta-language

The C `assert` macro takes a single condition as its argument, checks this condition at runtime, and aborts execution if the condition is false. This macro defines one of the simplest meta-languages possible: it has no state and a single operation. This section shows how MC can help even such simple interfaces by presenting two extensions that check the following two assertion invariants:

1. Assertions should not have non-debugging side-effects. Frequently, `assert` is used only for development and turned off in production code. If an `assert` condition has important side-effects, these will disappear and the program will behave incorrectly.

2. Assertion conditions should not fail. Programmers use assertions to check for conditions that should not happen. Any code path leading to an assertion that causes its boolean expression to fail is probably a bug.

4.1 Checking assertion side-effects

Figure 3 presents a *metal* checker that inspects assertion expressions for side-effects. The directive, "`flow_insensitive`," tells *metal* to apply the extension linearly over input functions rather than down all paths, improving speed and error reporting (since there will be exactly one message per violation). The SM begins in the initial state, `start`, and uses the literal *metal* pattern "`{assert(expr);}`" to find all `assert` uses. [2] On each match, *metal* stores the `assert` expression in the variable, `expr`. It then runs `start`'s action, which uses the *metal* procedure `mgk_expr_recurse` to recursively apply the SM to the expression in `expr` in the `in_assert` state. The `in_assert` state uses *metal*'s generic type "`any`" to match assignments, and pointer increments and decrements of any type. Note that the assignment operator will also detect uses of C's infix operators (e.g., `+=`, `-=`, etc.). The extension matches any function call with any set of arguments using the extended types

[2] Since patterns can match nearly arbitrary C code, it does not matter if `assert` is a function or a macro; we have modified the pre-processor to ignore line and file directives.

```
{ #include <assert.h> }
// Apply SM ignoring control flow
sm Assert flow_insensitive {
  // Match expressions of "any" type
  decl { any } expr, x, y, z;
  // Used in combination to match all
  // calls with any arguments
  decl { any_call } any_fcall;
  decl { any_args } args;

  // Find all assert calls.  Then apply
  // SM to "expr" in state "in_assert."
  start: { assert(expr); } ==>
       { mgk_expr_recurse(expr, in_assert); } ;
  // Find all side-effects
  in_assert:
     // Match all calls
     { any_fcall(args) } ==>
           { err("function call"); }
      // Match any assignment (including
      // the operators +=, -=, etc.)
    | { x = y }  ==> { err("assignment"); }
      // Match all increments and decrements
      // --z and ++z ommited for brevity
    | { z++ } ==> { err("post-increment"); }
    | { z-- } ==> { err("post-decrement"); } ;
}
```

Figure 3: A *metal* SM that warns of side-effects in `assert` uses.

`any_call` and `any_args` in combination. To assist developers in writing extensions, *metal* provides a set of generic types for matching different classes of types (e.g., scalars, pointers, floats), and different programming constructs (e.g., case labels, indirections).

When applied to Xok's ExOS library operating system, this 25 line extension found 16 violations in 199 `assert` uses. Two were false positives triggered by debugging functions. These could be suppressed by wrapping such calls in a differently named, unchecked assertion macro. The remaining fourteen cases were errors in crucial system code that would function incorrectly if the assertion was removed. The underlying cause of these errors was `assert`'s use as shorthand for checking the result of possibly-failing operations such as insertion of page table entries and deallocation of shared memory regions. A typical example is the following snippet from the ExOS "`mmap`" code to insert a page table entry:

```
/* libexos/os/mmap.c:mmap_fault_handler:410 */
assert(_exos_self_insert_pte(0, PG_P|
     PG_U|PG_W, PGROUNDDOWN(va), 0, NULL) == 0);
```

The effect of removing the assert condition (and hence these calls) would be mysterious virtual memory errors.

4.2 Checking assertions statically

Assertions specify conditions that the programmer believes must hold. Without MC, compilers are oblivious to this fact, so `assert` checks can only occur dynamically. With MC, it is possible to find errors by evaluating these conditions statically, thereby quickly and precisely finding errors.

We wrote such an extension on top of *xg++*. At a high level, it uses *xg++*'s dataflow routines to track the values of scalar variables. At each `assert` use, it evaluates the assertion expression against these known set of values. If the expression could fail, it emits a warning. Currently, *xg++* only performs primitive analysis that tracks the set of constant assignments to scalar variables on a given path. The set of possible values for a variable is then just the union of constant assignments to that variable before it is used. If any non-constant assignments occur, the value is considered "unknown." Returning the set of possible values allows the effectiveness of the checker to transparently increase as our analysis in *xg++* becomes more powerful. As a practical refinement, we eliminate a large class of false positives by ignoring assertions of the constant "0" (which always fails) since this is an idiomatic method for programmers to terminate execution in "impossible" situations.

When applied to the FLASH cache coherence code (discussed more in Section 8) the 100 line extension found five errors that could have crashed the system. These errors underscore the value of static evaluation, since they were in code that had been heavily tested for over five years. They had been missed because the length and complexity of typical FLASH code paths caused them to only occur sporadically. This complexity also makes manual detection of errors difficult. On one path, the assignment and the assertion that it violated were 300 lines apart and separated by 20 if statements, 6 else clauses, and 10 conditional compilation directives. Another case beat this by having 21 if statements, 4 else clauses, and 29 conditional compilations! Even given the exact situation that leads to the error, inspecting such paths is mind-numbing.

4.3 Discussion

Library implementations cannot inspect the context in which they are used or how they are invoked. MC can be used to attack these blindnesses. Our first extension used MC to to detect illegal actions in `assert` uses, something that an `assert` implementation cannot otherwise do either dynamically or statically. Our second extension used context knowledge to push dynamically evaluated conditions to compile time. A similar approach can be used to make certain dynamic error checks static or to improve performance by allowing implementations to specialize themselves to a given context, such as a memory allocator that generates specialized inline allocations for constant size allocation requests.

The restriction on side-effects in assertion conditions is a miniature example of a more general pattern of "language subsetting," where systems impose an execution context more restrictive than the base language in which code is written. We have built two other extensions that enforce system-specific execution restrictions. The first warns when kernel code uses floating point. It found one case where a Linux graphics driver assumes that floating point calculations will be evaluated at compile time. Using a compiler other than `gcc` or lower optimization levels could violate this assumption. The second checks for stack overflow. It found 10 places where Linux code allocated variables larger than 3K on the 6K kernel stack, and numerous 1K or larger allocations. Most of these led to patches by kernel maintainers. It also found a similar case in Xok where an innocent looking stack-allocated structure turned out to be over 8K bytes.

In addition to checking, systems can use restriction checkers for optimization by detecting when an application's actions are more limited than the general case. For example, a threads package can use smaller stack sizes than the default if it can derive an upper bound on stack usage.

5 Temporal Orderings

Many system operations must (or must not) happen in sequence. Sequencing rules are well-suited for compiler checking since sequences are frequently encoded as literal procedure calls in code. This allows a *metal* extension to find violations by searching for operations and transitioning to states that allow, disallow, or require other operations. This section discusses two such extensions. The first enforces an "X before Y" rule that system calls properly check application pointers passed to them for validity before using them. The second checks that code obeys a set of ordering rules for memory allocation and deallocation.

5.1 Checking copyin/copyout

Most operating systems guard against application corruption of kernel memory by, in part, using special routines to check system call input pointers and to move data between user and kernel space. We present an MC extension that finds errors in such code by

finding paths where an application pointer is used before passing through such routines. At each system call definition, the extension uses a special *metal* pattern to find every pointer parameter, which it binds to a `tainted` state. (The use of per-variable state differs from the previous checkers that used a single, global state per path.) The only legal operations on a `tainted` variable are being (1) killed by an assignment or (2) passed as an argument to functions expecting tainted inputs (e.g, data movement routines or output functions such as `kprintf`). All other uses will be signaled as an error.

We tailored a version of this checker for the Xok exokernel code. It detects which procedures are system calls using the exokernel naming convention that such routine names begin with the prefix "`sys_`." As a refinement, the checker warns when any non-system-call routines use "paranoid" user-data routines. It examined 187 distinct user pointers in the exokernel proper and device code and found 18 errors. A typical error is this command to issue disk requests:

```
/* from sys/kern/disk.c */
int sys_disk_request (u_int sn, struct Xn_name
         *xn_user, struct buf *reqbp, u_int k) {
  ...
  /* bypass for direct scsi commands */
  if (reqbp->b_flags & B_SCSICMD)
    return sys_disk_scsicmd (sn, k, reqbp);
```

Here, the pointer, `reqbp`, is passed in from user space and dereferenced in the `if` statement without being checked.

This extension also signalled 15 false positives. Four of these were due to a stylized use where non-null pointers were verified using standard routines, but null ones were allowed through (they would be handled correctly by lower levels). Three others were due to kernel backdoors used to let system calls call other system calls with unchecked parameters. The remaining were due to the checker's lack of global analysis and its disallowing of tainted variable copies.

5.2 Checking memory management

Most kernel code uses memory managers based loosely on the C procedures `malloc` and `free`. We present an extension that checks four common rules:

1. Since memory allocation can fail, kernel code must check whether the returned pointer is valid (i.e., not null) before using it.

2. Memory cannot be used after it has been freed.

3. Paths that allocate memory and then abort with an error should typically deallocate this memory before returning.

	Linux		OpenBSD	
Violation	**Bug**	**False**	**Bug**	**False**
No check	79	9	49	2
Error leak	44	49	3	1
Use after Free	7	3	0	0
Underflow	2	0	0	0
Total	132	61	52	3

Table 2: Error counts for Linux and OpenBSD. The checker was applied 4268 times in Linux and 464 times in OpenBSD.

4. The size of allocated memory cannot be less than the size of the object the assigned pointer holds.

Figure 4 shows a stripped-down extension that checks these rules. For space, the size check and most error reporting code is omitted. This extension, like the previous one, associates each variable with a state encoding what operations are legal on it. Pointers to allocated storage can be in exactly one of four states: `unknown`, `null`, `not_null`, or `freed`. A variable is bound to the `unknown` state at every allocation site. When an `unknown` variable is compared to null (e.g., in C, "0") the extension sets the variable's state on the true (null) path to `null` and on the false (non-null) path to `not_null`. When the variable is compared to non-null, these two cases are reversed. The two initial patterns recognize C's check-and-compare allocation idiom and combine these transitions with the initial variable binding. Pointers passed to `free` transition to the `freed` state. As a minor refinement, when variables are overwritten, the extension stops following them by transitioning to the special *metal* state, `stop`.

The checker only allows dereferences of `not_null` pointers. This restriction catches instances when memory is used before being checked, on null paths, or after being freed. It catches double-free errors by warning when `freed` pointers are passed to `free`. It catches cases when error paths do not free allocated memory by warning when any `non_null` or `unchecked` variable reaches a return of a negative integer, which idiomatically signals an error path.

The full version of the checker is 60 lines of code. We get a lot for so little: the extension implements a flow-sensitive compiler analysis pass that checks for rules on all paths and takes into consideration the observations furnished by passing through conditionals. As Table 2 shows, the extension found 132 errors in Linux and 51 errors in OpenBSD. It turned up 61 and 3 false positives respectively, most due to not

handling variable copies, or not detecting when allocated memory would be freed by a cleanup routine.

The most common error was not checking the result of memory allocation: 79 cases in Linux, 49 in OpenBSD. In Linux, the single largest source of these errors was an allocation macro, `CODA_ALLOC`, which was widely used throughout the Coda file system code. It contains the unfortunate code:

```
/* include/linux/coda_linux.h:CODA_ALLOC */
ptr = (cast)vmalloc((unsigned long) size);
...
if (ptr == 0)
  printk("kernel malloc returns 0 at %s:%d\n",
                __FILE__,__LINE__);
memset( ptr, 0, size );
```

While this code prints a helpful message on every failed allocation, the initialization using `memset` will immediately cause a kernel segmentation fault.

The next most common error was not freeing memory on error paths (44 in Linux, 3 in OpenBSD). A typical not-freeing error is given in Figure 5. An idiomatic mistake was to have many exit points from a function, but forgetting to free the memory at all of these points.

The seven use-after-freeing errors could cause nondeterministic bugs if another thread re-allocated the freed memory. The most common case was five cut-and-paste uses of the code:

```
/* drivers/isdn/pcbit:pcbit_init_dev */
kfree(dev);
iounmap((unsigned char*)dev->sh_mem);
release_mem_region(dev->ph_mem, 4096);
```

Here, the memory pointed to by `dev` is freed and then immediately used in two subsequent function calls.

Additionally, the checker discovered two under-allocation errors. These were particularly dangerous, since they could cause memory corruption whenever a routine is used, rather than only failing under high load. One was caused by an apparent typo where the size of the memory needed for a structure of type `struct atm_mpoa_qos` (92 bytes) was computed using the size of a structure of type `struct atm_qos` (84 bytes):

```
/* net/atm/mpc.c:169:atm_mpoa_add_qos */
struct atm_mpoa_qos *entry;
...
entry = kmalloc(sizeof(struct atm_qos),
                                GFP_KERNEL);
```

The other error reversed `kmalloc`'s size and interrupt level arguments, specifying that 7 (the value of `GFP_KERNEL`) bytes of storage to be allocated instead of 16. Currently, both errors are harmless, since the kernel uses a power-of-two memory allocator with a minimum allocation unit of 32 bytes. However, they are latent time bombs if a more space efficient allocator is ever used.

```
sm null_checker {
   decl { scalar } sz;      // match any scalar
   decl { const int } retv; // match const ints
   decl { any_ptr } v1;     // match any ptr
   // 'state' specifies 'v' will have a state
   state decl { any_ptr } v;

   // Associate allocated memory with unknown
   // state until compared to null.
   start, v.all:
       // set v's state on true path to "null",
       // on false path to "not_null"
       { ((v = (any)malloc(sz)) == 0) }
          ==> true=v.null, false=v.not_null
       // vice versa
     | { ((v = (any)malloc(sz)) != 0) }
          ==> true=v.not_null, false=v.null
       // unknown state until observed.
     | { v = (any)malloc(sz) } ==> v.unknown;

   // Allow comparisions on variables in
   // states "unknown", "null", and "not_null."
   v.unknown, v.null, v.not_null:
      { (v == 0) } ==>
          true = v.null, false = v.not_null
    | { (v != 0) } ==>
          true = v.not_null, false = v.null;

   // Catch error path leaks by warning when
   // a non-null, non-freed variable gets to a
   // return of a negative integer.
   v.unknown, v.not_null: { return retv; } ==>
        { if(mgk_int_cst(retv) < 0)
            err("Error path leak!"); };

   // No dereferences of null or unknown ptrs.
   v.null, v.unknown: { *(any *)v } ==>
           { err("Using ptr illegally!"); };

   // Allow free of all non-freed variables.
   v.unknown, v.null, v.not_null:
        { free(v); } ==> v.freed;

   // Check for double free and use after free.
   v.freed:
        { free(v) } ==> { err("Dup free!"); }
      | { v } ==> { err("Use-after-free!"); };

   // Overwriting v's value kills its state
   v.all: { v = v1 } ==> v.ok;
}
```

Figure 4: *Metal* extension that checks that allocated memory is (1) checked before use, (2) not used after a free, (3) not double freed, and (4) always freed on error paths (those returning a negative integer).

```
/* from drivers/char/tea6300.c */
static int tea6300_attach(...) {
  ...
  client = kmalloc(sizeof *client,GFP_KERNEL);
  if (!client)
    return -ENOMEM;
  ...
  tea = kmalloc (sizeof *tea, GFP_KERNEL);
  if (!tea)
    return -ENOMEM;
  ...
  MOD_INC_USE_COUNT;
  ...
}
```

Figure 5: Code with two errors: (1) not freeing memory (`client`) on an error path and (2) (discussed in Section 6) calling `MOD_INC_USE_COUNT` after potentially blocking memory allocation calls.

Check	Local	Global	False Pos
Interrupts	18	42	4
Spin Lock	21	42	4
Module	22	$\sim$ 53	$\sim$ 2
Total	61	$\sim$ 137	$\sim$ 10

Table 3: Results for checking if kernel routines block (1) with interrupts disabled ("Interrupts"), (2) while holding a spin lock ("Spin Lock"), or (3) in a way that causes a module race ("Module"). We divide errors into whether they needed local or global analysis. Local errors were due to direct calls to blocking functions; global errors reached a blocking routine via a multi-level call chain. The global analysis results for Module are marked as approximate since they have not been manually confirmed.

While these checks focus on raw byte memory management, the general extension template can be retrofitted to check similar rules for other, higher-level objects. A modified version of this extension found 15 probable errors in Linux "IRQ" allocation code where allocations were not checked for errors, and IRQ's were not deallocated on error paths.

6 Enforcing Rules Globally

The extensions described thus far have been implemented as local analyses. However, many systems rules are context dependent and apply globally across functions in a given call chain. This section presents two extensions that use *xg++*'s global analysis framework to check the following Linux rules:

1. Kernel code cannot call blocking functions with interrupts disabled or while holding a spin lock. Violating this rule can lead to deadlock [28].

2. A dynamically loaded kernel module cannot call blocking functions until the module's reference count has been properly set. Violating this rule leads to a race condition where the module could be unloaded while still in use [26].

We first describe a global analysis pass that computes a transitive closure of all potentially blocking routines. Then, we discuss how the two extensions use this result.

6.1 Computing blocking routines

We build a list of possibly blocking functions in two passes. The first, local pass, is a *metal* extension that traverses over every kernel routine, marking it if it calls functions known to potentially block. In Linux, blocking functions are primarily (1) kernel memory allocators called without the `GFP_ATOMIC` flag (which specifies not to sleep when the request cannot be fulfilled) or (2) routines to move data to or from user space (these block on a page fault). After processing each routine, the extension calls *xg++* support routines to emit the routine's flow graph to a file. The flow graph contains (1) the routine's annotation (if any) and (2) all procedures the routine calls. After the entire kernel has been processed, each input source file will have a corresponding emitted flow graph file. The second, global pass, uses *xg++* routines to link together all these files into a global call graph for the entire kernel. The global pass then uses *xg++* routines to perform a depth first traversal over this call graph calculating which routines have any path to a potentially blocking function. The output of this pass is a text file containing the names of all functions that could ever call a blocking function. Running the global analysis on the Linux kernel gives roughly 3000 functions that could potentially sleep.

6.2 Checking for blocking deadlock

Linux, like many OSes, uses a combination of interrupt disabling and spin locks for mutual exclusion. Interrupt disabling imposes an implicit rule: a thread running with interrupts disabled cannot block, since if it was the last runnable thread, the system will deadlock. Similarly, because of the implementation of Linux kernel thread scheduling, threads holding spin locks cannot block. Doing so causes deadlock when a sleeping thread holds a spin lock that a thread on

the same CPU is trying to acquire.

Our *metal* extension checks both rules by assuming each routine starts in a "clean" state with interrupts enabled and no locks held. As it traverses each code path, if it hits a statement that disables interrupts, it goes to a `disabled` state; an enable interrupt call returns it to the original state. Similarly, if it hits a function that acquires a spin lock, it traverses to a `locked` state; an unlock call returns it to the clean state. While in either of these states (or their composition), the extension examines all function calls and reports an error if the call is to a function in the list of potentially blocking routines.

Despite the simplicity of these rules, real code violates it in numerous places. The extension found 123 errors in Linux. Of those errors, 79 could lead to deadlock. The remaining 44 were calls to `kmalloc` with interrupts disabled. Possibly motivated by the frequency of this error, the `kmalloc` code checks if it is called with interrupts disabled, and, if so, it prints a warning and re-enables interrupts. In situations where interrupt disabling was used for synchronization, this leads to race conditions. The following code snippet is representative of a typical error (the mistake has been annotated in the source but not fixed):

```
/* drivers/sound/midibuf.c */
save_flags(flags);
cli();
...
while (c < count)
  ...
  for (i = 0; i < n; i++)
    /* BROKE BROKE-CANT DO THIS WITH CLI!! */
    copy_from_user((char *)&tmp_data,
                           &(buf)[c],1);
    QUEUE_BYTE(midi_out_buf[dev], tmp_data);
    c++;
  }
restore_flags(flags);
```

The call to `copy_from_user` can implicitly sleep, but is called after interrupts have been disabled with the call to `cli`.

The local errors seem to be caused by driver implementors not having a clear picture of either (1) the rules they have to follow and (2) that user data movement routines can block. The global errors seem to be caused by the fact that it is often hard to tell if a function can potentially block without tediously tracing through several function calls in different files, or without a considerable amount of *a priori* Linux kernel knowledge.

The checker produced eight false positives. Six were because the global calculation of blocking functions does not check if a called function would reenable interrupts before calling a blocking function. Two others were caused by name conflicts where a file defined and called a function with the same name as a blocking function.

The approach of this section also applies to other operating systems. Another implementor used our system to write an extension for the OpenBSD system that checked if interrupt handling code called a blocking operation. He found one bug where an interrupt handler could call a page allocation routine that in turn called a blocking memory allocator [29].

6.3 Checking module reference counts

Linux allows kernel subsystems to be dynamically loaded and unloaded. Modules have a reference count tracking the number of kernel subsystems using them. Modules increment this count during loading (using `MOD_INC_USE_COUNT`) and decrement it during unloading (using `MOD_DEC_USE_COUNT`). The kernel can unload modules with a zero reference count at any time. A module must protect against being unloaded while sleeping by incrementing its reference count before calling a blocking function. Similarly, during unloading, it cannot block after decrementing its count. Finally, if the module aborts installation after incrementing its reference count, it must decrement the count to restore it to its original value.

Our extension checks for load race conditions by tracking if a potentially blocking function has been called and flagging subsequent `MOD_INC`s. Conversely, it checks for unload race conditions by tracking if a `MOD_DEC` has been performed and flagging subsequent calls to potentially blocking functions. It finds dangling references by emitting an error when a `MOD_INC` has not been reversed along a path that returns a negative integer (which idiomatically signals an error). As Table 3 shows, a local version of the extension that did not use the global list of blocking functions found 22 rule violations, whereas the global version found 53 cases (we have not yet confirmed the global errors).

7 Linux Mutual Exclusion

The complexity of dealing with concurrency leads most of the Linux kernel and its device drivers to follow a localized strategy where critical sections begin and end within the same function body. Despite this stylized use, the size of the code and implementors' imperfect understanding leads to errors. We wrote an extended version of the interrupt checker described in Section 3 to check that each kernel function conforms to the following conditions:

Condition	Applied	Bug	False Pos
Holding lock	~ 5400	29	113 (90)
Double lock	-	1	3
Double unlock	-	1	20 (18)
Intr disabled	~ 5800	44 (43)	63 (54)
Bottom half	~ 180	4	12
Bogus flags	~ 3200	4	49 (24)
Total	-	83 (82)	260 (201)

Table 4: Results of running the Linux synchronization primitives checker on kernel version 2.3.99. The **Applied** column is an estimate of the number of times the check was applied. We skipped twelve warnings that were difficult to classify. The parenthesized numbers show the changes when the two files with the most false positives are ignored.

1. All locks acquired within the function body are released before exiting.
2. No execution paths attempt to lock or unlock the same lock twice.
3. Upon exiting, interrupts are either enabled or restored to their initial state.
4. The "bottom halves" of interrupt handlers are not disabled upon exiting.
5. Interrupt flags are saved before they are restored.

Table 4 shows the results of running the extension on Linux. The "Applied" column is an estimate of the number of times each check was applied. Two device drivers account for a large number of false positives because they use macros that consult runtime state before locking or unlocking. The parenthesized numbers show the changes in the false positive results (over 20%) when these two files are ignored.

The most common bugs are either holding a lock or leaving interrupts disabled on function exit. These bugs often occur when detecting an error condition after which the function returns immediately. For example, the checker found this bug in a device driver for PCMCIA card services

```
/* drivers/pcmcia/cs.c:
            pcmcia_deregister_client */
spin_lock_irqsave(&s->lock, flags);
client = &s->clients;
while ((*client) && ((*client) != handle))
    client = &(*client)->next;
if (*client == NULL)
    /* forgot about &s->lock, flags! */
    return CS_BAD_HANDLE;
```

The checks for Linux locking conventions have resulted in seven kernel patches, including a fix for the error shown above. All seven patches fix cases where a lock is mistakenly held when exiting a function, and six of the seven are in device drivers (the last patch was to an implementation of ipv4 network filters). We have not been able to confirm many of the other potential bugs with kernel or device driver developers, though several strong OS implementors have examined them and consider them to be at least suspicious. Most of the potential bugs are in device drivers and networking code – this is not surprising since much of this code is written by developers throughout the world with varying degrees of familiarity with the Linux kernel.

The false positives mostly come from three sources. Code that intentionally violates the convention for the sake of efficiency or modularity accounts for 90 false positives. For example, sometimes a family of related device drivers will define an interface that breaks the conventions. Another large source of false positives (48) is caused by the fact that our checker only performs local analysis. Some drivers implement their own locking functions using the basic primitives provided by the system. The checker will warn when these functions exit holding a lock or with interrupts disabled, which is exactly what they are supposed to do. Global analysis could eliminate many of these false positives. Finally, the fact that our system does not prune simple, impossible paths accounts for 35 false positives. A typical example of this is when kernel code conditionally acquires a lock, performs an action, and then releases the lock based on the same condition. There are only two possible paths through this code, not the four that our system thinks exist.

The remaining 21 false positives could be eliminated by extending the checker's notion of locking functions and changing our system to prune the false branch of loop conditionals of the form "`for(;;)`."

8 Optimizing FLASH

In addition to checking, MC can be used for optimization. Below, we describe three extensions written to find system-level optimization opportunities in the FLASH machine's cache coherence code [20]. This code must be fast because it implements functionality (cache coherence) that is usually placed in hardware. Eliminating even a single instruction is considered beneficial. Several of the protocols examined here have been aggressively tuned for years due to their use in numerous performance papers as evidence for the effectiveness of software-controlled

Optimization	Number	False Pos	LOC
Buffer Free	11	9	30
Message Length	40	0	32
XOR Opcode	hundreds	~10	400(*)

Table 5: MC-based FLASH optimizer results. **Number** counts how many optimization opportunities were found. The XOR checker is written in an old version of the system — a version written in *metal* would be several factors smaller.

cache coherence. Despite this effort, MC optimizers found hundreds of optimization opportunities, mostly due to the difficulty in manually performing equivalent searches across FLASH's deeply nested paths.

Buffer-free optimization. Each time a FLASH node receives a message, it invokes a customized message protocol handler that determines how to satisfy the request and update the protocol state. Handlers use the incoming message buffer to send outgoing data messages, and must free it before exiting. Handlers can send data messages, which need a buffer, and control messages, which do not. Many handlers send more than one message when responding to a request. To minimize the chance of losing a buffer, implementors are typically conservative and defer buffer freeing until the last handler send, irrespective of whether the last send(s) was a control message and therefore did not need a buffer. Unfortunately, while this strategy simplifies handler code, it increases buffer contention under high load.

Our extension indicates when buffer frees can occur earlier in the code. It traces all sends on each path through the function, and by looking at send arguments, detects if the send (1) needs a buffer and (2) frees its buffer. It gives a suggestion for any path that has an active buffer that ends with a "suffix" of control sends. The extension is 56 lines long, and found 11 instances in a large FLASH protocol, "`dyn_ptr`," where the buffer could be safely freed earlier. Each of these optimizations could be implemented by changing only two lines of code. The extension also produced nine false positives. Most of these were cases where the execution path was too complex to optimize without major code restructuring.

Redundant length assignments. Our second, lower-level optimization extension detects redundant assignments to a message buffer's length field. For speed, when sending multiple messages, implementors set a buffer's message length early in a handler and then try to reuse this setting across multiple messages. Long path lengths make it easy to miss redundant assignments. Our checker detects redundancies by recording the last assignment on every path and warning if there are two assignments of the same constant. It discovered 40 redundant assignments in the FLASH protocol code.

Efficient opcode setting. Message headers must specify the message's opcode (type). Opcode assignment costs two instructions. However, if the handler knows what opcode is currently in a header, it can change the opcode in one instruction by xoring the message header with the xor of the new and current opcode. Our extension detects such cases by computing when a message header, with known opcode, is assigned a new opcode. Both the old and new opcodes must be the same on all incoming paths. The extension determines the initial header value by looking in an automatically-built list of all opcodes a handler might receive. If there is only one possible opcode value, the extension records it and starts in a "known" state. Otherwise, the checker starts in an "unknown" state. It transitions from this state to the "known" state after the first opcode assignment. Each assignment encountered in the known state is annotated with the current opcode value. A second pass then checks every assignment and, if all paths reached it in the known state with the same opcode, emits a warning to the user that xor could be used to save an instruction. This checker found hundreds of such cases.

9 Conclusion

Systems are pervaded with restrictions of what actions programmers must always or never perform, how they must order events, and which actions are legal in a given context. In many cases, these restrictions link together the entire system, creating a fragile, intricate mess. Currently, systems builders obey these restrictions as well as they can. Unfortunately, system complexity makes such obedience difficult to sustain. Programmers make mistakes, and often they have only an approximate understanding of important system restrictions. Such mistakes can easily evade testing, which rarely exercises all cases.

We have shown that many system restrictions can be automatically checked and exploited using meta-level compilation (MC). MC makes it easy for implementors to extend compilers with lightweight system-specific checkers and optimizers. Currently, a system rule must be understood by all implementors. MC allows one implementor, who understands this rule, to write a check that is enforced on everyone's code. This leverage exerts tremendous practical force on the development of complex systems.

Check	Errors	False Positives	Uses	LOC
Side-effects(§ 4.1)	14	2	199	25
Static assert(§ 4.2)	5	0	1759	100
Stack check(§ 4.3)	10+	0	332K	53
User-ptr(§ 5.1)	18	15	187	68
Allocation(§ 5.2)	184	64	4732	60
Block(§ 6.2)	123	8	-	131
Module(§ 6.3)	~75	2	-	133
Mutex(§ 7)	82	201	14K	64
Total	~511	~292	-	669

Table 6: The results of MC-based checkers summarized over all checks. **Error** is the number of errors found, **False Positives** is the number of false positives, **Uses** is the number of times the check was applied, and **LOC** is the number of lines of *metal* code for the extension (including comments and whitespace).

MC is a general approach, scaling from simple cases such as checking assertions up to global strategies for mutual exclusion and deadlock avoidance. We have demonstrated MC's power by using it to check four real, heavily-used, and tested systems. It found bugs in all of them — roughly 500 in all — many of which would be difficult to find with testing or manual inspection. Further, these extensions typically required less than a day and a hundred lines of code to implement. Curiously, writing code to check restrictions is significantly easier than writing code that obeys them. With few exceptions, our extensions were written by programmers who, at best, only had a passing familiarity with the systems to which they were applied. We believe that these results show that the use of meta-level compilation can significantly aid system construction.

10 Acknowledgements

We thank David Dill for many discussions on software checking, Wilson Hsieh for discussions about the initial *metal* language, Mark Heinrich for his willingness to validate our FLASH results. Wallace Huang and Yu Ping Hu verified many of the error messages from the null checker in Section 5. Rusty Russell, Tim Waugh, Alan Cox, and David Miller answered numerous Linux questions and gave feedback on error reports. Additionally, we thank the other generous readers of `linux-kernel` for their feedback and support. We thank Mark Horowitz for his support, and the initial suggestion of looking at FLASH, which led to numerous other interesting directions. Finally, we thank David Dill, Wilson Hsieh, Mark Horowitz, the anonymous reviewers and especially Andrew Myers for valuable feedback on the paper.

References

[1] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing systems*, pages 131–152, Spring 1996.

[2] R. S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 1(43):166–192, January 1996.

[3] S. Chiba. A metaobject protocol for C++. In *OOPSLA 1995 Conference Proceedings Object-oriented programming systems, languages, and applications*, pages 285–299, October 1995.

[4] A. Chou, B. Chelf, D.R. Engler, and M. Heinrich. Using meta-level compilation to check FLASH protocol code. To appear in ASPLOS 2000, November 2000.

[5] A. Chou and D.R. Engler. Metal: A language and system for building lightweight, system-specific software checkers, analyzers and optimizers. Available upon request: acc@cs.stanford.edu, 2000.

[6] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *ICSE 2000*, 2000.

[7] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the First Conference on Domain Specific Languages*, pages 229–242, October 1997.

[8] D.L. Detlefs, R.M. Leino, G. Nelson, and J.B. Saxe. Extended static checking. TR SRC-159, COMPAQ SRC, December 1998.

[9] D.R. Engler. Incorporating application semantics and control into compilation. In *Proceedings of the First Conference on Domain Specific Languages*, October 1997. An extended version "Interface Compilation: Steps toward Compiling Program Interfaces as Languages" was selected to appear in IEEE Transactions on Software Engineering, May/June, 1999, Volume 25, Number 3, p 387–400.

[10] D. Evans, J. Guttag, J. Horning, and Y.M. Tan. Lclint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, December 1994.

[11] R. W. Floyd. *Assigning meanings to programs*, pages 19–32. J.T. Schwartz, Ed. American Mathematical Society, 1967.

[12] C. W. Fraser and D. R. Hanson. *A retargetable C compiler: design and implementation.* Benjamin/Cummings Publishing Co., Redwood City, CA, 1995.

[13] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, December 1992.

[14] G. Holzmann and M. Smith. Software model checking: Extracting verification models from source code. In *Invited Paper. Proc. PSTV/FORTE99 Publ. Kluwer*, 1999.

[15] Intrinsa. A technical introduction to PREfix/Enterprise. Technical report, Intrinsa Corporation, 1998.

[16] M.F. Kaashoek, D.R. Engler, G.R. Ganger, H.M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[17] G. Kiczales, J. des Rivieres, and D.G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, June 1997.

[19] A. Kolawa and A. Hicken. Insure++: A tool to support total quality software. `www.parasoft.com/insure/papers/tech.htm`.

[20] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.

[21] J. Lamping, G. Kiczales, L.H. Rodriguez Jr., and E. Ruf. An architecture for an open compiler. In *Proceedings of the IMSA'92 workshop on reflection and meta-level architectures*, 1992.

[22] T. Lord. Application specific static code checking for C programs: Ctool. In *˜twaddle: A Digital Zine (version 1.0)*, 1997.

[23] K.L. McMillan and J. Schwalbe. Formal verification of the gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 242–51. Tokyo, Japan Inf. Process. Soc., 1991.

[24] T.C. Mowry, A.K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, 1996.

[25] G. Nelson. *Techniques for program verification.* Available as Xerox PARC Research Report CSL-81-10, June, 1981, Stanford University, 1981.

[26] K. Owens. "Please review all modules for unload races". Sent to `linux-kernel@vger.rutgers.edu`. Gives protocol to follow to prevent module unload races., 2000.

[27] R. Rashid. Personal communication. Microsoft's internal tool used to check violations in Windows device drivers., July 2000.

[28] P. Russell (rusty@linuxcare.com). Unreliable guide to hacking the Linux kernel. Distributed with the 2.3.99 Linux RedHat Kernel, 2000.

[29] C.P. Sapuntzakis. Personal communication. Bug in OpenBSD where an interrupt context could call blocking memory allocator, April 2000.

[30] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[31] A. Srivastava and A. Eustace. ATOM — a system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994.

[32] U. Stern and D.L. Dill. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

DOI:10.1145/1646353.1646374

How Coverity built a bug-finding tool, and a business, around the unlimited supply of bugs in software systems.

BY AL BESSEY, KEN BLOCK, BEN CHELF, ANDY CHOU, BRYAN FULTON, SETH HALLEM, CHARLES HENRI-GROS, ASYA KAMSKY, SCOTT MCPEAK, AND DAWSON ENGLER

A Few Billion Lines of Code Later

Using Static Analysis to Find Bugs in the Real World

IN 2002, COVERITY commercialized[3] a research static bug-finding tool.[6,9] Not surprisingly, as academics, our view of commercial realities was not perfectly accurate. However, the problems we encountered were not the obvious ones. Discussions with tool researchers and system builders suggest we were not alone in our naïveté. Here, we document some of the more important examples of what we learned developing and commercializing an industrial-strength bug-finding tool.

We built our tool to find generic errors (such as memory corruption and data races) and system-specific or interface-specific violations (such as violations of function-ordering constraints). The tool, like all static bug finders, leveraged the fact that programming rules often map clearly to source code; thus static inspection can find many of their violations. For example, to check the rule "acquired locks must be released," a checker would look for relevant operations (such as `lock()` and `unlock()`) and inspect the code path after flagging rule disobedience (such as `lock()` with no `unlock()` and double locking).

For those who keep track of such things, checkers in the research system typically traverse program paths (flow-sensitive) in a forward direction, going across function calls (inter-procedural) while keeping track of call-site-specific information (context-sensitive) and toward the end of the effort had some of the support needed to detect when a path was infeasible (path-sensitive).

A glance through the literature reveals many ways to go about static bug finding.[1,2,4,7,8,11] For us, the central religion was results: If it worked, it was good, and if not, not. The ideal: check millions of lines of code with little manual setup and find the maximum number of serious true errors with the minimum number of false reports. As much as possible, we avoided using annotations or specifications to reduce manual labor.

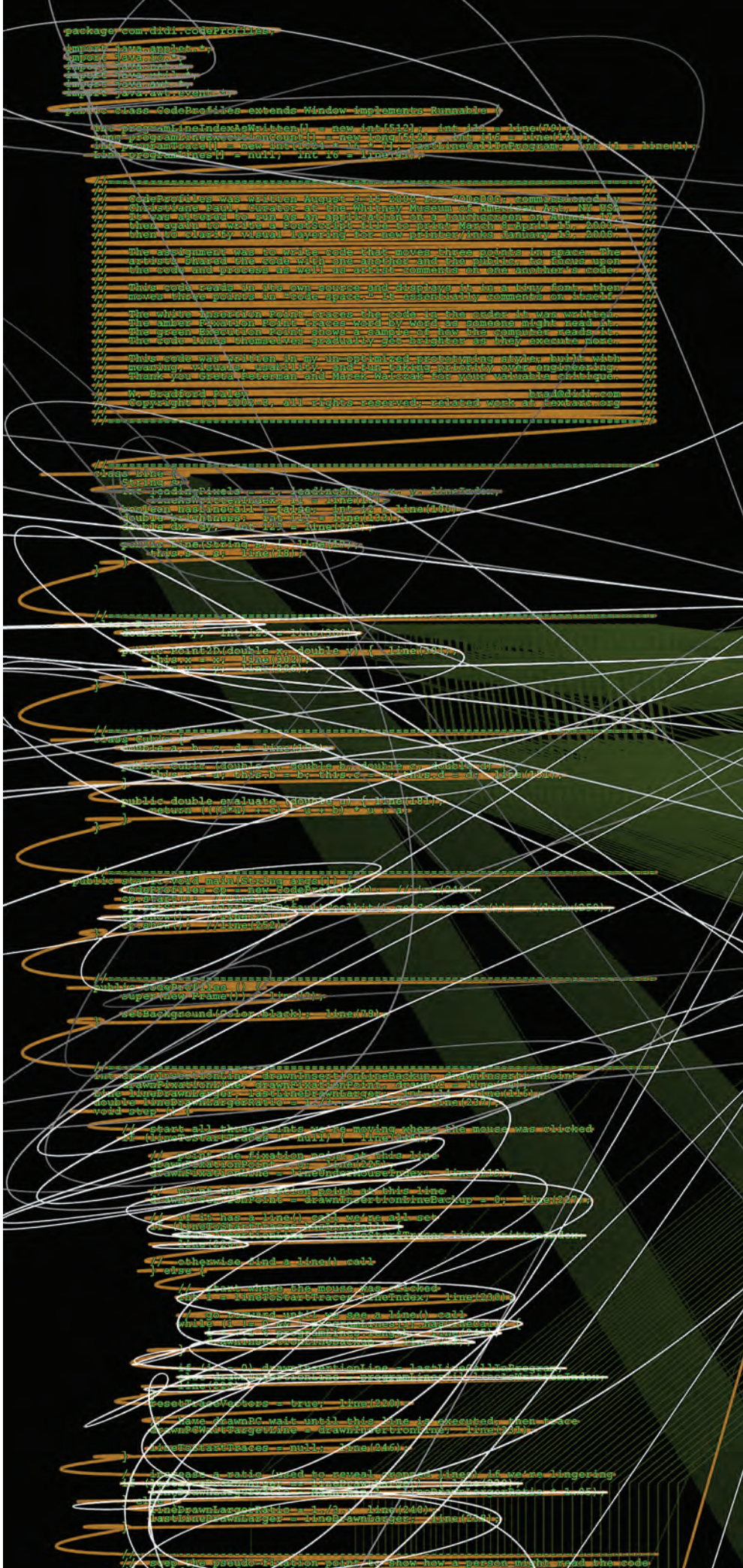
Like the PREfix product,[2] we were also unsound. Our product did not verify the absence of errors but rather tried to find as many of them as possible. Unsoundness let us focus on handling the easiest cases first, scaling up as it proved useful. We could ignore code constructs that led to high rates of false-error messages (false positives) or analysis complexity, in the extreme skipping problematic code entirely (such as assembly statements, functions, or even entire files). Circa 2000, unsoundness was controversial in the research community, though it has since become almost a de facto tool bias for commercial products and many research projects.

Initially, publishing was the main force driving tool development. We would generally devise a set of checkers or analysis tricks, run them over a few

million lines of code (typically Linux), count the bugs, and write everything up. Like other early static-tool researchers, we benefited from what seems an empirical law: Assuming you have a reasonable tool, if you run it over a large, previously unchecked system, you will always find bugs. If you don't, the immediate knee-jerk reaction is that something must be wrong. Misconfiguration? Mistake with macros? Wrong compilation target? If programmers must obey a rule hundreds of times, then without an automatic safety net they cannot avoid mistakes. Thus, even our initial effort with primitive analysis found hundreds of errors.

This is the research context. We now describe the commercial context. Our rough view of the technical challenges of commercialization was that given that the tool would regularly handle "large amounts" of "real" code, we needed only a pretty box; the rest was a business issue. This view was naïve. While we include many examples of unexpected obstacles here, they devolve mainly from consequences of two main dynamics:

First, in the research lab a few people check a few code bases; in reality many check many. The problems that show up when thousands of programmers use a tool to check hundreds (or even thousands) of code bases do not show up when you and your co-authors check only a few. The result of summing many independent random variables? A Gaussian distribution, most of it not on the points you saw and adapted to in the lab. Furthermore, Gaussian distributions have tails. As the number of samples grows, so, too, does the absolute number of points several standard deviations from the mean. The unusual starts to occur with increasing frequency.



W. Bradford Paley's CodeProfiles was originally commissioned for the Whitney Museum of American Art's "CODeDOC" Exhibition and later included in MoMA's "Design and the Elastic Mind" exhibition. CodeProfiles explores the space of code itself; the program reads its source into memory, traces three points as they once moved through that space, then prints itself on the page.

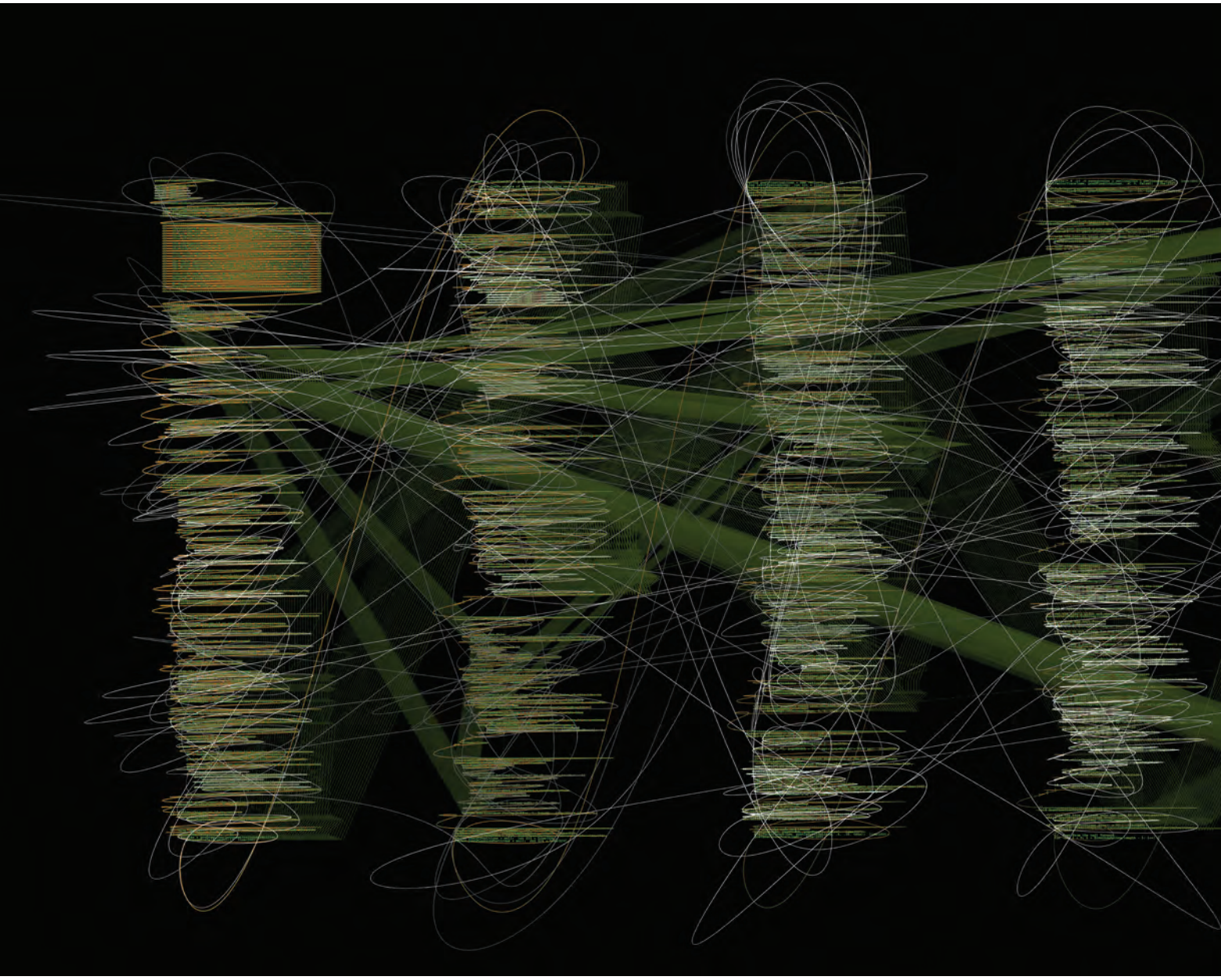
"CODEPROFILES" BY W. BRADFORD PALEY. HTTP://DIDI.COM

For code, these features include problematic idioms, the types of false positives encountered, the distance of a dialect from a language standard, and the way the build works. For developers, variations appear in raw ability, knowledge, the amount they care about bugs, false positives, and the types of both. A given company won't deviate in all these features but, given the number of features to choose from, often includes at least one weird oddity. Weird is not good. Tools want expected. Expected you can tune a tool to handle; surprise interacts badly with tuning assumptions.

Second, in the lab the user's values, knowledge, and incentives are those of the tool builder, since the user and the builder are the same person. Deployment leads to severe fission; users often have little understanding of the tool and little interest in helping develop it (for reasons ranging from simple skepticism to perverse reward incentives) and typically label any error message they find confusing as false. A tool that works well under these constraints looks very different from one tool builders design for themselves.

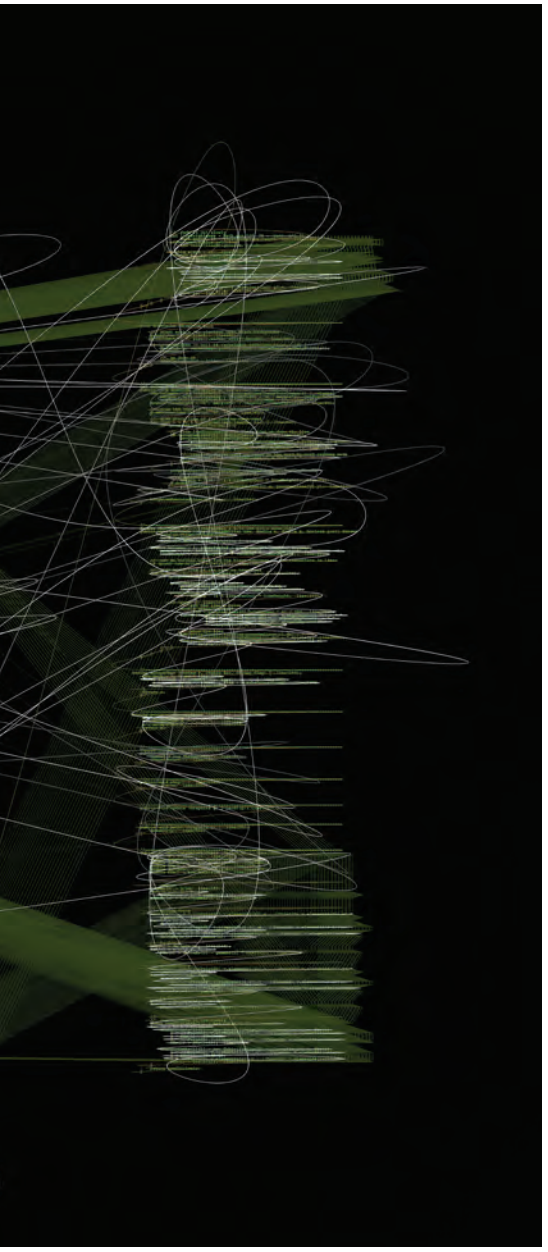
However, for every user who lacks the understanding or motivation one might hope for, another is eager to understand how it all works (or perhaps already does), willing to help even beyond what one might consider reasonable. Such champions make sales as easily as their antithesis blocks them. However, since their main requirements tend to be technical (the tool must work) the reader likely sees how to make them happy, so we rarely discuss them here.

Most of our lessons come from two different styles of use: the initial trial of the tool and how the company uses the tool after buying it. The trial is a pre-sale demonstration that attempts to show that the tool works well on a potential customer's code. We generally ship a salesperson and an engineer to the customer's site. The engineer configures the tool and runs it over a given code base and presents results soon after. Initially, the checking run would happen



in the morning, and the results meeting would follow in the afternoon; as code size at trials grows it's not uncommon to split them across two (or more) days.

Sending people to a trial dramatically raises the incremental cost of each sale. However, it gives the non-trivial benefit of letting us educate customers (so they do not label serious, true bugs as false positives) and do real-time, ad hoc workarounds of weird customer system setups.



"CODEPROFILES" BY W. BRADFORD PALEY

The trial structure is a harsh test for any tool, and there is little time. The checked system is large (millions of lines of code, with 20–30MLOC a possibility). The code and its build system are both difficult to understand. However, the tool must routinely go from never seeing the system previously to getting good bugs in a few hours. Since we present results almost immediately after the checking run, the bugs must be good with few false positives; there is no time to cherry pick them.

Furthermore, the error messages must be clear enough that the sales engineer (who didn't build the checked system or the tool) can diagnose and explain them in real time in response to "What about this one?" questions.

The most common usage model for the product has companies run it as part of their nightly build. Thus, most require that checking runs complete in 12 hours, though those with larger code bases (10+MLOC) grudgingly accept 24 hours. A tool that cannot analyze at least 1,400 lines of code per minute makes it difficult to meet these targets. During a checking run, error messages are put in a database for subsequent triaging, where users label them as true errors or false positives. We spend significant effort designing the system so these labels are automatically reapplied if the error message they refer to comes up on subsequent runs, despite code-dilating edits or analysis-changing bug-fixes to checkers.

As of this writing (December 2009), approximately 700 customers have licensed the Coverity Static Analysis product, with somewhat more than a billion lines of code among them. We estimate that since its creation the tool has analyzed several billion lines of code, some more difficult than others.

Caveats. Drawing lessons from a single data point has obvious problems. Our product's requirements roughly form a "least common denominator" set needed by any tool that uses non-trivial analysis to check large amounts of code across many organizations; the tool must find and parse the code, and users must be able to understand error messages. Further, there are many ways to handle the problems we have encountered, and our way may not be the best one. We discuss our methods more for specificity than as a claim of solution.

Finally, while we have had success as a static-tools company, these are small steps. We are tiny compared to mature technology companies. Here, too, we have tried to limit our discussion to conditions likely to be true in a larger setting.

Laws of Bug Finding

The fundamental law of bug finding is No Check = No Bug. If the tool can't check a system, file, code path, or given property, then it won't find bugs in it. Assuming a reasonable tool, the first order bound on bug counts is just how much code can be shoved through the tool. Ten times more code is 10 times more bugs.

We imagined this law was as simple a statement of fact as we needed. Unfortunately, two seemingly vacuous corollaries place harsh first-order bounds on bug counts:

Law: You can't check code you don't see. It seems too trite to note that checking code requires first finding it... until you try to do so consistently on many large code bases. Probably the most reliable way to check a system is to grab its code during the build process; the build system knows exactly which files are included in the system and how to compile them. This seems like a simple task. Unfortunately, it's often difficult to understand what an ad hoc, homegrown build system is doing well enough to extract this information, a difficulty compounded by the near-universal absolute edict: "No, you can't touch that." By default, companies refuse to let an external force modify anything; you cannot modify their compiler path, their broken makefiles (if they have any), or in any way write or reconfigure anything other than your own temporary files. Which is fine, since if you need to modify it, you most likely won't understand it.

Further, for isolation, companies often insist on setting up a test machine for you to use. As a result, not infrequently the build you are given to check does not work in the first place, which you would get blamed for if you had touched anything.

Our approach in the initial months of commercialization in 2002 was a low-tech, read-only replay of the build commands: run `make`, record its output in a file, and rewrite the invocations to their compiler (such as gcc) to instead call our checking tool, then rerun everything. Easy and simple. This approach worked perfectly in the lab and for a small number of our earliest customers. We then had the fol-

lowing conversation with a potential customer:

"How do we run your tool?"

"Just type '`make`' and we'll rewrite its output."

"What's '`make`'? We use `ClearCase`."

"Uh, What's `ClearCase`?"

This turned out to be a chasm we couldn't cross. (Strictly speaking, the customer used '`ClearMake`,' but the superficial similarities in name are entirely unhelpful at the technical level.) We skipped that company and went to a few others. They exposed other problems with our method, which we papered over with 90% hacks. None seemed so troublesome as to force us to rethink the approach—at least until we got the following support call from a large customer:

"Why is it when I run your tool, I have to reinstall my Linux distribution from CD?"

This was indeed a puzzling question. Some poking around exposed the following chain of events: the company's `make` used a novel format to print out the absolute path of the directory in which the compiler ran; our script misparsed this path, producing the empty string that we gave as the destination to the Unix "`cd`" (change directory) command, causing it to change to the top level of the system; it ran "`rm -rf *`" (recursive delete) during compilation to clean up temporary files; and the build process ran as root. Summing these points produces the removal of all files on the system.

The right approach, which we have used for the past seven years, kicks off the build process and intercepts every system call it invokes. As a result, we can see everything needed for checking, including the exact executables invoked, their command lines, the directory they run in, and the version of the compiler (needed for compiler-bug workarounds). This control makes it easy to grab and precisely check all source code, to the extent of automatically changing the language dialect on a per-file basis.

To invoke our tool users need only call it with their build command as an argument:

```
cov-build <build command>
```

We thought this approach was bulletproof. Unfortunately, as the astute reader has noted, it requires a command prompt. Soon after implementing it we went to a large company, so large it had a hyperspecialized build engineer, who engaged in the following dialogue:

A misunderstood explanation means the error is ignored or, worse, transmuted into a false positive.

"How do I run your tool?"

"Oh, it's easy. Just type '`cov-build`' before your build command."

"Build command? I just push this [GUI] button..."

Social vs. technical. The social restriction that you cannot change anything, no matter how broken it may be, forces ugly workarounds. A representative example is: Build interposition on Windows requires running the compiler in the debugger. Unfortunately, doing so causes a very popular windows C++ compiler—Visual Studio C++ .NET 2003—to prematurely exit with a bizarre error message. After some high-stress fussing, it turns out that the compiler has a use-after-free bug, hit when code used a Microsoft-specific C language extension (certain invocations of its `#using` directive). The compiler runs fine in normal use; when it reads the freed memory, the original contents are still there, so everything works. However, when run with the debugger, the compiler switches to using a "debug malloc," which on each `free` call sets the freed memory contents to a garbage value. The subsequent read returns this value, and the compiler blows up with a fatal error. The sufficiently perverse reader can no doubt guess the "solution."[a]

Law: You can't check code you can't parse. Checking code deeply requires understanding the code's semantics. The most basic requirement is that you parse it. Parsing is considered a solved problem. Unfortunately, this view is naïve, rooted in the widely believed myth that programming languages exist.

The C language does not exist; neither does Java, C++, and C#. While a language may exist as an abstract idea, and even have a pile of paper (a standard) purporting to define it, a standard is not a compiler. What language do people write code in? The character strings accepted by their compiler. Further, they equate compilation with certification. A file their compiler does

a Immediately after process startup our tool writes 0 to the memory location of the "in debugger" variable that the compiler checks to decide whether to use the debug `malloc`.

not reject has been certified as "C code" no matter how blatantly illegal its contents may be to a language scholar. Fed this illegal not-C code, a tool's C front-end will reject it. This problem is the tool's problem.

Compounding it (and others) the person responsible for running the tool is often not the one punished if the checked code breaks. (This person also often doesn't understand the checked code or how the tool works.) In particular, since our tool often runs as part of the nightly build, the build engineer managing this process is often in charge of ensuring the tool runs correctly. Many build engineers have a single concrete metric of success: that all tools terminate with successful exit codes. They see Coverity's tool as just another speed bump in the list of things they must get through. Guess how receptive they are to fixing code the "official" compiler accepted but the tool rejected with a parse error? This lack of interest generally extends to any aspect of the tool for which they are responsible.

Many (all?) compilers diverge from the standard. Compilers have bugs. Or are very old. Written by people who misunderstand the specification (not just for C++). Or have numerous extensions. The mere presence of these divergences causes the code they allow to appear. If a compiler accepts construct X, then given enough programmers and code, eventually X is typed, not rejected, then encased in the code base, where the static tool will, not helpfully, flag it as a parse error.

The tool can't simply ignore divergent code, since significant markets are awash in it. For example, one enormous software company once viewed conformance as a competitive disadvantage, since it would let others make tools usable in lieu of its own. Embedded software companies make great tool customers, given the bug aversion of their customers; users don't like it if their cars (or even their toasters) crash. Unfortunately, the space constraints in such systems and their tight coupling to hardware have led to an astonishing oeuvre of enthusiastically used compiler extensions.

Finally, in safety-critical software systems, changing the compiler often requires costly re-certification. Thus, we routinely see the use of decades-old compilers. While the languages these compilers accept have interesting features, strong concordance with a modern language standard is not one of them. Age begets new problems. Realistically, diagnosing a compiler's divergences requires having a copy of the compiler. How do you purchase a license for a compiler 20 versions old? Or whose company has gone out of business? Not through normal channels. We have literally resorted to buying copies off eBay.

This dynamic shows up in a softer way with non-safety-critical systems; the larger the code base, the more the sales force is rewarded for a sale, skewing sales toward such systems. Large code bases take a while to build and often get tied to the compiler used when they were born, skewing the average age of the compilers whose languages we must accept.

If divergence-induced parse errors are isolated events scattered here and there, then they don't matter. An unsound tool can skip them. Unfortunately, failure often isn't modular. In a sad, too-common story line, some crucial, purportedly "C" header file contains a blatantly illegal non-C construct. It gets included by all files. The no-longer-potential customer is treated to a constant stream of parse errors as your compiler rips through the customer's source files, rejecting each in turn. The customer's derisive stance is, "Deep source code analysis? Your tool can't even compile code. How can it find bugs?" It may find this event so amusing that it tells many friends.

Tiny set of bad snippets seen in header files. One of the first examples we encountered of illegal-construct-in-key-header file came up at a large networking company

```
// "redefinition of parameter 'a'"
void foo(int a, int a);
```

The programmer names `foo`'s first formal parameter a and, in a form of lexical locality, the second as well. Harmless. But any conformant compiler will reject this code. Our tool certainly did. This is not helpful; compiling no files means finding no bugs, and people don't need your tool for that. And, because its compiler accepted it, the potential customer blamed us.

Here's an opposite, less-harmless case where the programmer is trying to make two different things the same

```
typedef char int;
```

("Useless type name in empty declaration.")

And one where readability trumps the language spec

```
unsigned x = 0xdead _ beef;
```

("Invalid suffix '_beef' on integer constant.")

From the embedded space, creating a label that takes no space

```
void x;
```

("Storage size of 'x' is not known.")

Another embedded example that controls where the space comes from

```
unsigned x @ "text";
```

("Stray '@' in program.")

A more advanced case of a nonstandard construct is

```
Int16 ErrSetJump(ErrJumpBuf buf)
 = { 0x4E40 + 15, 0xA085; }
```

It treats the hexadecimal values of machine-code instructions as program source.

The award for most widely used extension should, perhaps, go to Microsoft support for precompiled headers. Among the most nettlesome troubles is that the compiler skips all the text before an inclusion of a precompiled header. The implication of this behavior is that the following code can be compiled without complaint:

```
I can put whatever I want here.
It doesn't have to compile.
If your compiler gives an error,
   it sucks.
#include  <some-precompiled-
   header.h>
```

Microsoft's on-the-fly header fabrication makes things worse.

Assembly is the most consistently troublesome construct. It's already non-portable, so compilers seem to almost deliberately use weird syntax, making it difficult to handle in a general way. Unfortunately, if a programmer uses assembly it's probably to write a widely used function, and if the programmer does it, the most likely place to put it is in a widely used

header file. Here are two ways (out of many) to issue a `mov` instruction

```
// First way
foo() {
      _ _asm mov eax, eab
      mov eax, eab;
}

// Second way
#pragma asm
_ _asm [ mov eax, eab mov
eax, eab ]
#pragma end _asm
```

The only thing shared in addition to mov is the lack of common textual keys that can be used to elide them.

We have thus far discussed only C, a simple language; C++ compilers diverge to an even worse degree, and we go to great lengths to support them. On the other hand, C# and Java have been easier, since we analyze the bytecode they compile to rather than their source.

How to parse not-C with a C front-end. OK, so programmers use extensions. How difficult is it to solve this problem? Coverity has a full-time team of some of its sharpest engineers to firefight this banal, technically uninteresting problem as their sole job. They're never done.[b]

We first tried to make the problem someone else's problem by using the Edison Design Group (EDG) C/C++ front-end to parse code.[5] EDG has worked on how to parse real C code since 1989 and is the de facto industry standard front-end. Anyone deciding to not build a homegrown front-end will almost certainly license from EDG. All those who do build a homegrown front-end will almost certainly wish they did license EDG after a few experiences with real code. EDG aims not just for mere feature compatibility but for version-specific bug compatibility across a range of compilers. Its front-end probably resides near the limit of what a profitable company can do in terms of front-end gyrations.

Unfortunately, the creativity of compiler writers means that despite two decades of work EDG still regularly meets defeat when trying to parse real-world large code bases.[c] Thus, our next step is for each supported compiler, we write a set of "transformers" that mangle its personal language into something closer to what EDG can parse. The most common transformation simply rips out the offending construct. As one measure of how much C does not exist, the table here counts the lines of transformer code needed to make the languages accepted by 18 widely used compilers look vaguely like C. A line of transformer code was almost always written only when we were burned to a degree that was difficult to work around. Adding each new compiler to our list of "supported" compilers almost always requires writing some kind of transformer. Unfortunately, we sometimes need a deeper view of semantics so are forced to hack EDG directly. This method is a last resort. Still, at last count (as of early 2009) there were more than 406(!) places in the front-end where we had an `#ifdef COVERITY` to handle a specific, unanticipated construct.

EDG is widely used as a compiler front-end. One might think that for customers using EDG-based compilers we would be in great shape. Unfortunately, this is not necessarily the case. Even ignoring the fact that compilers based on EDG often modify EDG in idiosyncratic ways, there is no single "EDG front-end" but rather many versions and possible configurations that often accept a slightly different language variant than the (often newer) version we use. As a Sisyphean twist, assume we cannot work around and report an incompatibility. If EDG then considers the problem important enough to fix, it will roll it together with other patches into a new version.

So, to get our own fix, we must upgrade the version we use, often causing divergence from other unupgraded EDG compiler front-ends, and more issues ensue.

Social versus technical. Can we get customer source code? Almost always, no. Despite nondisclosure agreements, even for parse errors and preprocessed code, though perhaps because we are viewed as too small to sue to recoup damages. As a result, our sales engineers must type problems in reports from memory. This works as well as you might expect. It's worse for performance problems, which often show up only in large-code settings. But one shouldn't complain, since classified systems make things even worse. Can we send someone onsite to look at the code? No. You listen to recited syntax on the phone.

Bugs

Do bugs matter? Companies buy bug-finding tools because they see bugs as bad. However, not everyone agrees that bugs matter. The following event has occurred during numerous trials. The tool finds a clear, ugly error (memory corruption or use-after-free) in important code, and the interaction with the customer goes like thus:

"So?"

"Isn't that bad? What happens if you hit it?"

"Oh, it'll crash. We'll get a call." [Shrug.]

If developers don't feel pain, they often don't care. Indifference can arise from lack of accountability; if QA cannot reproduce a bug, then there is no blame. Other times, it's just odd:

"Is this a bug?"

"I'm just the security guy."

"That's not a bug; it's in third-party code."

"A leak? Don't know. The author left years ago..."

No, your tool is broken; that is not a bug. Given enough code, any bug-

b Anecdotally, the dynamic memory-checking tool Purify[10] had an analogous struggle at the machine-code level, where Purify's developers expended significant resources reverse engineering the various activation-record layouts used by different compilers.

c Coverity won the dubious honor of being the single largest source of EDG bug reports after only three years of use.

Lines of code per transformer for 18 common compilers we support.

160 QNX	280 HP-UX	285 picc.cpp
294 sun.java.cpp	384 st.cpp	334 cosmic.cpp
421 intel.cpp	457 sun.cpp	603 iccmsa.cpp
629 bcc.cpp	673 diab.cpp	756 xlc.cpp
912 ARM	914 GNU	1294 Microsoft
1425 keil.cpp	1848 cw.cpp	1665 Metrowerks

finding tool will uncover some weird examples. Given enough coders, you'll see the same thing. The following utterances were culled from trial meetings:

Upon seeing an error report saying the following loop body was dead code

```
foo(i = 1; i < 0; i++)
  ... deadcode ...
```

"No, that's a false positive; a loop executes at least once."

For this memory corruption error (32-bit machine)

```
int a[2], b;
memset(a, 0, 12);
```

"No, I meant to do that; they are next to each other."

For this use-after-free

```
free(foo);
foo->bar = ...;
```

"No, that's OK; there is no malloc call between the free and use."

As a final example, a buffer overflow checker flagged a bunch of errors of the form

```
unsigned p[4];
...
p[4] = 1;
```

"No, ANSI lets you write 1 past the end of the array."

After heated argument, the programmer said, "We'll have to agree to disagree." We could agree about the disagreement, though we couldn't quite comprehend it. The (subtle?) interplay between 0-based offsets and buffer sizes seems to come up every few months.

While programmers are not often so egregiously mistaken, the general trend holds; a not-understood bug report is commonly labeled a false positive, rather than spurring the programmer to delve deeper. The result? We have completely abandoned some analyses that might generate difficult-to-understand reports.

How to handle cluelessness. You cannot often argue with people who are sufficiently confused about technical matters; they think you are the one who doesn't get it. They also tend to get emotional. Arguing reliably kills sales. What to do? One trick is to try to organize a large meeting so their peers do the work for you. The more people in the room, the more likely there is someone very smart and respected and cares (about bugs and about the given code), can diagnose an error (to counter arguments it's a false positive), has been burned by a similar error, loses his/her bonus for errors, or is in another group (another potential sale).

...it's not uncommon for tool improvement to be viewed as "bad" or at least a problem.

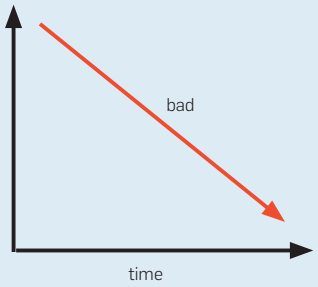
Further, a larger results meeting increases the probability that anyone laid off at a later date attended it and saw how your tool worked. True story: A networking company agreed to buy the Coverity product, and one week later laid off 110 people (not because of us). Good or bad? For the fired people it clearly wasn't a happy day. However, it had a surprising result for us at a business level; when these people were hired at other companies some suggested bringing the tool in for a trial, resulting in four sales.

What happens when you can't fix all the bugs? If you think bugs are bad enough to buy a bug-finding tool, you will fix them. Not quite. A rough heuristic is that fewer than 1,000 bugs, then fix them. More? The baseline is to record the current bugs, don't fix them but do fix any new bugs. Many companies have independently come up with this practice, which is more rational than it seems. Having a lot of bugs usually requires a lot of code. Much of it won't have changed in a long time. A reasonable, conservative heuristic is if you haven't touched code in years, don't modify it (even for a bug fix) to avoid causing any breakage.

A surprising consequence is it's not uncommon for tool improvement to be viewed as "bad" or at least a problem. Pretend you are a manager. For anything bad you can measure, you want it to diminish over time. This means you are improving something and get a bonus.

You may not understand technical issues that well, and your boss certainly doesn't understand them. Thus, you want a simple graph that looks like Figure 1; no manager gets a bonus for Figure 2. Representative story: At company X, version 2.4 of the tool found approximately 2,400 errors, and over time the company fixed about 1,200 of them. Then it upgraded to version 3.6. Suddenly there were 3,600 errors. The manager was furious for two reasons: One, we "undid" all the work his people

Figure 1. Bugs down over time = manager bonus.



had done, and two, how could we have missed them the first time?

How do upgrades happen when more bugs is no good? Companies independently settle on a small number of upgrade models:

Never. Guarantees "improvement";

Never before a release (where it would be most crucial). Counterintuitively happens most often in companies that believe the tool helps with release quality in that they use it to "gate" the release;

Never before a meeting. This is at least socially rational;

Upgrade, then roll back. Seems to happen at least once at large companies; and

Upgrade only checkers where they fix most errors. Common checkers include use-after-free, memory corruption, (sometimes) locking, and (sometimes) checkers that flag code contradictions.

Do missed errors matter? If people don't fix all the bugs, do missed errors (false negatives) matter? Of course not; they are invisible. Well, not always. Common cases: Potential customers intentionally introduced bugs into the system, asking "Why didn't you find it?" Many check if you find important past bugs. The easiest sale is to a group whose code you are checking that was horribly burned by a specific bug last week, and you find it. If you don't find it? No matter the hundreds of other bugs that may be the next important bug.

Here is an open secret known to bug finders: The set of bugs found by tool A is rarely a superset of another tool B, even if A is much better than B. Thus, the discussion gets pushed from "A is better than B" to "A finds some things, B finds some things" and does not help the case of A.

Adding bugs can be a problem; losing already inspected bugs is always a problem, even if you replace them with many more new errors. While users know in theory that the tool is "not a verifier," it's very different when the tool demonstrates this limitation, good and hard, by losing a few hundred known errors after an upgrade.

The easiest way to lose bugs is to add just one to your tool. A bug that causes false negatives is easy to miss. One such bug in how our early research tool's internal representation handled array references meant the analysis ignored most array uses for more than nine months. In our commercial product, blatant situations like this are prevented through detailed unit testing, but uncovering the effect of subtle bugs is still difficult because customer source code is complex and not available.

Churn

Users really want the same result from run to run. Even if they changed their code base. Even if they upgraded the tool. Their model of error messages? Compiler warnings. Classic determinism states: the same input + same function = same result. What users want: different input (modified code base) + different function (tool version) = same result. As a result, we find upgrades to be a constant headache. Analysis changes can easily cause the set of defects found to shift. The new-speak term we use internally is "churn." A big change from academia is that we spend considerable time and energy worrying about churn when modifying checkers. We try to cap churn at less than 5% per release. This goal means large classes of analysis tricks are disallowed since they cannot obviously guarantee minimal effect on the bugs found. Randomization is verboten, a tragedy given that it provides simple, elegant solutions to many of the exponential problems we encounter. Timeouts are also bad and sometimes used as a last resort but never encouraged.

Myth: More analysis is always good. While nondeterministic analysis might cause problems, it seems that adding more deterministic analysis is always good. Bring on path sensitivity! Theorem proving! SAT solvers! Unfortunately, no.

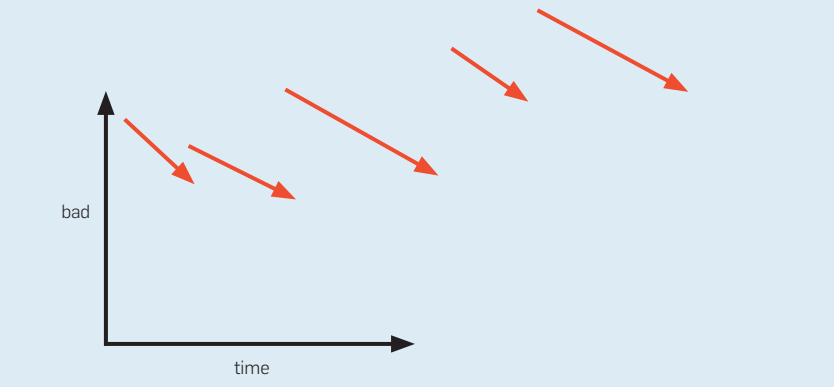
At the most basic level, errors found with little analysis are often better than errors found with deeper tricks. A good error is probable, a true error, easy to diagnose; best is difficult to misdiagnose. As the number of analysis steps increases, so, too, does the chance of analysis mistake, user confusion, or the perceived improbability of event sequence. No analysis equals no mistake.

Further, explaining errors is often more difficult than finding them. A misunderstood explanation means the error is ignored or, worse, transmuted into a false positive. The heuristic we follow: Whenever a checker calls a complicated analysis subroutine, we have to explain what that routine did to the user, and the user will then have to (correctly) manually replicate that tricky thing in his/her head.

Sophisticated analysis is not easy to explain or redo manually. Compounding the problem, users often lack a strong grasp on how compilers work. A representative user quote is "'Static' analysis'? What's the performance overhead?"

The end result? Since the analysis that suppresses false positives is invisible (it removes error messages rather than generates them) its sophistication has scaled far beyond what our research

Figure 2. No bonus.



system did. On the other hand, the commercial Coverity product, despite its improvements, lags behind the research system in some ways because it had to drop checkers or techniques that demand too much sophistication on the part of the user. As an example, for many years we gave up on checkers that flagged concurrency errors; while finding such errors was not too difficult, explaining them to many users was. (The PREfix system also avoided reporting races for similar reasons though is now supported by Coverity.)

No bug is too foolish to check for. Given enough code, developers will write almost anything you can think of. Further, completely foolish errors can be some of the most serious; it's difficult to be extravagantly nonsensical in a harmless way. We've found many errors over the years. One of the absolute best was the following in the X Window System:

```
if(getuid() != 0 && geteuid == 0) {
 ErrorF("only root");
 exit(1);
  }
```

It allowed any local user to get root access[d] and generated enormous press coverage, including a mention on Fox news (the Web site). The checker was written by Scott McPeak as a quick hack to get himself familiar with the system. It made it into the product not because of a perceived need but because there was no reason not to put it in. Fortunately.

False Positives

False positives do matter. In our experience, more than 30% easily cause problems. People ignore the tool. True bugs get lost in the false. A vicious cycle starts where low trust causes complex bugs to be labeled false positives, leading to yet lower trust. We have seen this cycle triggered even for true errors. If people don't understand an error, they label it false. And done once, induction makes the (n+1)th time easier. We initially thought false positives could be eliminated through technology. Because of this dynamic we no longer think so.

We've spent considerable technical effort to achieve low false-positive rates in our static analysis product. We aim for below 20% for "stable" checkers. When forced to choose between more bugs or fewer false positives we typically choose the latter.

Talking about "false positive rate" is simplistic since false positives are not all equal. The initial reports matter inordinately; if the first *N* reports are false positives (*N* = 3?), people tend to utter variants on "This tool sucks." Furthermore, you never want an embarrassing false positive. A stupid false positive implies the tool is stupid. ("It's not even smart enough to figure *that* out?") This technical mistake can cause social problems. An expensive tool needs someone with power within a company or organization to champion it. Such people often have at least one enemy. You don't want to provide ammunition that would embarrass the tool champion internally; a false positive that fits in a punchline is really bad.

Conclusion

While we've focused on some of the less-pleasant experiences in the commercialization of bug-finding products, two positive experiences trump them all. First, selling a static tool has become dramatically easier in recent years. There has been a seismic shift in terms of the average programmer "getting it." When you say you have a static bug-finding tool, the response is no longer "Huh?" or "Lint? Yuck." This shift seems due to static bug finders being in wider use, giving rise to nice networking effects. The person you talk to likely knows someone using such a tool, has a competitor that uses it, or has been in a company that used it.

Moreover, while seemingly vacuous tautologies have had a negative effect on technical development, a nice balancing empirical tautology holds that bug finding is worthwhile for anyone with an effective tool. If you can find code, and the checked system is big enough, and you can compile (enough of) it, then you will always find serious errors. This appears to be a law. We encourage readers to exploit it.

Acknowledgments

We thank Paul Twohey, Cristian Cadar, and especially Philip Guo for their helpful, last-minute proofreading. The experience covered here was the work of many. We thank all who helped build the tool and company to its current state, especially the sales engineers, support engineers, and services engineers who took the product into complex environments and were often the first to bear the brunt of problems. Without them there would be no company to document. We especially thank all the customers who tolerated the tool during its transition from research quality to production quality and the numerous champions whose insightful feedback helped us focus on what mattered. C

References

1. Ball, T. and Rajamani, S.K. Automatically validating temporal safety properties of interfaces. In *Proceedings of the Eighth international SPIN Workshop on Model Checking of Software* (Toronto, Ontario, Canada). M. Dwyer, Ed. Springer-Verlag, New York, 2001, 103–122.
2. Bush, W., Pincus, J., and Sielaff, D. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience 30*, 7 (June 2000), 775–802.
3. Coverity static analysis; http://www.coverity.com
4. Das, M., Lerner, S., and Seigle, M. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany, June 17–19). ACM Press, New York, 2002, 57–68.
5. Edison Design Group. EDG C compiler front-end; http://www.edg.com
6. Engler, D., Chelf, B., Chou, A., and Hallem, S. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Conference on Operating System Design & Implementation* (San Diego, Oct. 22–25). USENIX Association, Berkeley, CA, 2000, 1–1.
7. Flanagan, C., Leino, K.M., Lillibridge, M., Nelson, G., Saxe, J.B., and Stata, R. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Berlin, Germany, June 17–19). ACM Press, New York, 2002, 234–245.
8. Foster, J.S., Terauchi, T., and Aiken, A. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany, June 17–19). ACM Press, New York, 2002, 1–12.
9. Hallem, S., Chelf, B., Xie, Y., and Engler, D. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Berlin, Germany, June 17–19). ACM Press, New York, 2002, 69–82.
10. Hastings, R. and Joyce, B. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 USENIX Conference* (Berkeley, CA, Jan. 20–24). USENIX Association, Berkeley, CA, 1992, 125–138.
11. Xie, Y. and Aiken, A. Context- and path-sensitive memory leak detection. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal, Sept. 5–9). ACM Press, New York, 2005, 115–125.

Al Bessey, **Ken Block**, **Ben Chelf**, **Andy Chou**, **Bryan Fulton**, **Seth Hallem**, **Charles Henri-Gros**, **Asya Kamsky**, and **Scott McPeak** are current or former employees of Coverity, Inc., a software company based in San Francisco, CA.; http://www.coverity.com

Dawson Engler (engler@stanford.edu) is an associate professor in the Department of Computer Science and Electrical Engineering at Stanford University, Stanford, CA, and technical advisor to Coverity, Inc., San Francisco, CA.

d The tautological check `geteuid == 0` was intended to be `geteuid() == 0`. In its current form, it compares the address of `geteuid` to 0; given that the function exists, its address is never 0.

Superoptimizer -- A Look at the Smallest Program

Henry Massalin

Department of Computer Science
Columbia University
New York, NY 10027

Abstract

Given an instruction set, the superoptimizer finds the shortest program to compute a function. Startling programs have been generated, many of them engaging in convoluted bit-fiddling bearing little resemblance to the source programs which defined the functions. The key idea in the superoptimizer is a probabilistic test that makes exhaustive searches practical for programs of useful size. The search space is defined by the processor's instruction set, which may include the whole set, but it is typically restricted to a subset. By constraining the instructions and observing the effect on the output program, one can gain insight into the design of instruction sets. In addition, superoptimized programs may be used by peephole optimizers to improve the quality of generated code, or by assembly language programmers to improve manually written code.

1. Introduction

The search for the optimal algorithm to compute a function is one of the fundamental problems in computer science. In contrast to theoretical studies of optimal algorithms, practical applications motivated the design, implementation, and use of the superoptimizer. Instead of proving upper or lower bounds for abstract algorithms, the superoptimizer finds the shortest program in the program space defined by the instruction set of commercial machines, such the Motorola 68000 or Intel 8086.

The functions to be optimized are specified with programs written using the target machine's instruction set. Therefore, the input to the superoptimizer is a machine language program. The output is another program, which may be shorter. Since both programs run on the same processor, with a well-defined environment, we can establish their equivalence.

A probabilistic test and a method for pruning the search tree makes the superoptimizer a practical tool for programs of limited size (about 13 machine instructions).

In section 2, we describe an interesting example to illustrate the superoptimizer approach. The design and algorithms used in the superoptimizer are detailed in section 3. We discuss the applications and limitations of the superoptimizer in section 4. In section 5, we compare the superoptimizer with related work. The conclusion in section 6 is followed by a list of interesting minimal programs in appendix I.

2. An Interesting Example

We begin with an example to show what superoptimized code looks like. The instruction set used here, as in most of the paper, is Motorola's 68020 instruction set. Our example is the *signum* function, defined by the following program:

```
signum(x)
int     x;
{
        if(x > 0)       return 1;
        else if(x < 0)  return -1;
        else            return 0;
}
```

This function compiles to 9 instructions occupying 18 bytes of memory on the SUN-3 C compiler. Most programmers when asked to write this function in assembly language would use comparison instructions and conditional jumps to decide in what range the argument lies. Typically, this takes 8 68020 instructions, although clever programmers can do it in 6.

It turns out that by exploiting various properties of two's complement arithmetic one can write *signum* in four instructions! This is what superoptimizer found when fed the compiled machine code for the signum function as input:

```
(x in d0)
add.l   d0,d0   |add d0 to itself
subx.l d1,d1    |subtract (d1 + Carry) from d1
negx.l d0       |put (0 - d0 - Carry) into d0
addx.l d1,d1    |add (d1 + Carry) to d1
(signum(x) in d1)  (4 instructions)
```

Like a typical superoptimized program, the logic is really convoluted. One of the first things that comes to mind is "where are the conditional jumps?". As we will see later, many functions that would normally be written with conditional jumps are optimized into short programs without them. This can result in significant speedups for certain pipelined machines that execute conditional jumps slowly.

Let us see how it works. The "add.l d0, d0" instruction doubles the contents of register d0, but more importantly, the sign bit is now in the carry flag. The "subx.l d1, d1" instruction computes "d1-d1-carry --> d1". Regardless of the initial value of d1, d1-d1-carry is -carry. Thus d1 is -1 if d0 was negative and 0 otherwise. Besides negating, "negx.l d0" will set the carry flag if and only if d0 was nonzero. Finally, "addx.l d1, d1" doubles d1 and adds the carry. Now if d0 was negative, d1 is -1 and carry is set, so d1+d1+carry is -1, if d0 was 0, d1 is 0 and carry is clear, so d0+d0+carry is 0, if d0 was positive, d1 is 0 and carry is set, so d1+d1+carry is 1.

3. Superoptimizer Internals

Superoptimizer takes a program written in machine language as the input source. It finds the shortest program that computes the same function as the source program by doing an exhaustive search over all possible programs. The search space is defined by choosing a subset of the machine's instruction set, and the op-codes of these instructions are stored in a table. Superoptimizer consults this table and generates all combinations of these instructions, first of length 1, then of length 2, and so on. Each of these generated programs is tested, and if found to match the function of the source program, superoptimizer prints the program and halts.

Two methods are used to reduce the search time. The first is a fast probabilistic test for determining the the equivalence of two programs. The second is a method for pruning the search space while maintaining the guarantee of optimality. These two methods will now be discussed, but first a boolean-logic equivalence test will be explained, which was the first test proceedure implemented, because it finds use in the tree pruning method.

3.1. Boolean Test

The most important part of superoptimizer is the routine that determines whether two pieces of code computes the same function. The first version of superoptimizer used what we call the *boolean program verifier*. The idea was to express the function output in terms of boolean-logic operations on the input argument. Once this is done, two programs are equivalent if their boolean expressions matches minterm for minterm.

In practice, some instructions such as *add* and *mul* have boolean expressions with on the order of 2^31 minterms. Various methods had been devised to reduce the memory requirements, but it took too long to compute the boolean expressions for every program generated. The initial version of superoptimizer tested about 40 programs per second, and this allowed programs of up to 3 instructions to be generated in reasonable time.

3.2. Probabilistict Test

The idea behind the probabilistic test is simple: run the machine code for the program being tested a few times with some set of inputs and check whether the outputs match those of the source program. The idea here is that most programs will fail this simple test, and a full program verification test will be done only for the few programs that this test fails to catch. Running through a few carefully chosen test vectors takes very little time. Currently, superoptimizer can test 50000 programs per second and the exhaustive search approach becomes practical.

The test vectors are chosen (manually) to maximize the probability that a random program will fail on the first or second test. For example, the test vectors for the *signum* function included -1000, 0 and 456 as the first three vectors. This quickly eliminates programs that return the same answer regardless of argument, answers of the same sign, as well as programs that return their argument. Following these vectors, all the numbers from -1024 to 1024 were tested.

It was found in practice that a program has a very low probability of passing this execution test and failing the boolean verification test. This fact proves very useful since most programs of interest have boolean expressions that are too large to fit in memory. We can dispense with the boolean test and manually inspect the generated programs for correctness, without having to analyze a large number of wrong programs. This manual check is not difficult since the programs are small (about 4 to 13 instructions). Currently, superoptimizer runs without the boolean check, and the author has yet to find an incorrect program.

One problem introduced by the probabilistic execution test is machine dependency. The test works only if the instruction set being searched can be executed on the machine running the superoptimizer. In other words, if we wish to change the instruction set, we would have to port the superoptimizer to the new machine. This port is not too difficult since the current version of superoptimizer is rather short (about 300 lines of 68020 assembly code), however it does require that one translate it into the target assembly code.

3.3. Pruning

In order to further reduce the search time, we filter out instruction sequences that are known not to occur in any optimal program. Any sequence of instructions that has the same effect on the machine state as a shorter sequence cannot be part of an optimal program, because if it were, you can get a shorter program by substituting the shorter sequence, and therefore the program was not optimal. Typical sequences include the obviously silly "move X,Y; move X,Y" and "move X,Y; move Y,X", "and X,Y; move Z,Y" in which the MOVE destroys the result of the AND, "and #0,X" which does the same thing as "clr X", and "and X,Y; <any> Z,W; and.l X,Y" where the second AND is superfluous.

This filtering is done with N-dimensional bit tables, where N is the length of the longest sequence we wish to filter. Each instruction in the sequence we wish to test indexes one dimension of the bit table, and a lookup value of '1' causes the program to be rejected as non-optimal (and also as incorrect, since it is the same as a shorter program, and superoptimizer has already checked all shorter programs).

There are two ways that these bit tables can be filled. A human can tell the bit table maker program to exclude all "move X,Y; move Y,X" sequences. The program then scans all instructions in all dimensions of the bit matrix and sets the values accordingly. One can also run superoptimizer with the boolean test, and have it find the equivalences on its own.

4. Applications and Limitations

4.1. Current Limitations

Even with the probabilistic test, the exhaustive search still grows exponentially with the number of instructions in the generated program. The current version of superoptimizer has generated programs 12 instructions long in several hours running time on a 16MHz 68020 computer. Therefore, the superoptimizer has limited usefulness as a code generator for a compiler.

Another difficulty concerns pointers. A pointer can point anywhere in memory and so to model a pointer in terms of boolean expressions one needs to take all of memory into account. Even on a 256-byte machine, there are 2^(2^(256*8)) possible minterms, and these are just too many. We have explored the probabilistic test approach for pointers, but the results have been inconclusive.

Currently, we have only the 68020 version of the superoptimizer running the probabilistic test, so the instruction sets are restricted to subsets of the 68020 set. The machine-independent version of superoptimizer is limited to very short programs.

4.2. Applications

Because of the pointer problem, superoptimizer works best when the instruction set is constrained to register-register operations. Even so, it can be used to analyze instruction sets. Some of the programs in appendix I were tried on the Western Electric WE32000 microprocessor and in every case the resulting program was longer

than the 68020 programs. The reason for this was found to be the lack of an add-with-carry instruction and the fact that the flags are set according to the 32 bit result, even for byte sized operands. The National Semiconductor NS32032 was also found to suffer from flag problems. Here the difficulty is that extra instructions are needed to test the outcome of an operation because few instructions set the flags.

Another use would be in the design of RISC architectures. One can try various instruction sets simply by coding their function in terms of boolean expressions and seeing what superoptimizer comes up with. A particular instruction may be omitted if superoptimizer finds a short equivalent sequence of other instructions.

The superoptimizer may be very useful in optimizing little tasks that often confront a compiler. An example is finding the optimal program that multiplies by a particular constant for use in accessing arrays and such. Some examples of multiplication by constants can be found in I.6.

Another useful feature of superoptimizer is the identity tables containing the equivalent program sequences found. These programs may be extracted and used to increase the power of a conventional peephole optimizer.

In practice, the best use of superoptimizer has been as an aid to the assembly language programmer. An experienced programmer can use superoptimizer to come up with nifty equivalent sequences for small sections of his code, while retaining the overall logical flow that makes a program maintainable. This method has been used by the author (along with another program that optimizes code emulating state machines) to write the C library function *printf* in only 500 bytes.

5. Comparison with Related Work

The most commonly used optimization techniques are those that attempt to improve the code that a compiler produces. Examples are peephole optimizers and data-flow analysis. Peephole optimizers [2] are table driven pattern matchers that operate on the assembly language code produced by the compiler. Every time a sequence of instructions is matched by one of the tables, a smaller and faster replacement sequence is used.

Data-flow analysis [1] is a technique applied during the semantic and code generation phases of the compilation process. It improves code in several ways. First, it eliminates redundant computations (common sub-expression elimination). Second, it moves expressions within a loop whose values do not depend on the loop variable to outside the loop (loop invariance). Third, (also in a loop) it converts expressions of the form 'K * loop-index' into the equivalent arithmetic progression 'TMP = TMP + K' (strength reduction).

These methods are general. They work regardless of the machine-specific details such as the representation of an integer. However, usually the result is not optimal in either space or speed. Superoptimizer depends on the instruction set, however, the code is guaranteed to be optimal in space and it does a very good job in speed as well.

Krumme and Ackley [4] have written a code generator for the DEC-10 computer that is based on exhaustive search. Their method translates each interior node of an expression tree into several viable instruction sequences. These sequences are then pieced together to form a set of translations for the entire expression. This set is then searched to find the cheapest alternative.

In their method, there is a one to one correspondence between the instructions in the translation and the original expression. For example, if there's an add in the expression, there will also be an add somewhere in the generated code. Superoptimizer has a more global view of the problem. It 'translates' one sequence of instructions into another completely different sequence. On the other hand, superoptimizer can't translate large programs.

The two approaches can be seen as complementing each other. Superoptimizer can be used to prepare the code generation tables used in Krumme and Ackley's method. Their method can also be incorporated into superoptimizer to increase the size of programs that can be handled. Superoptimizer can generate several short equivalent sequences for small fragments of the source program, and then Krumme and Ackley's method would be used to piece these together and find a short overall sequence.

Kessler [3] has written a code optimization tool, which translates sequences of instructions into one single instruction. The superoptimizer can be seen as a more general tool with broader applications, since it can transform programs of many instructions to another one of several instructions. However, Kessler's optimizer works regardless of program size, and therefore can be easily used to optimize compiled code. Another difference is that he uses template matching, while superoptimizer relies on exhaustive search.

6. Conclusion

We have taken a practical approach to the search for the optimal program. We have found that the shortest programs are surprising, often containing sequences of instructions that one would not expect to see side by side. The *signum* function is an example of this, and the *min* and *max* functions given in section I.3 contain a beautiful combination of the logical *and* and the arithmetic *add*.

Exhaustive search is justified by these results, and a probabilistic test allows programs of practical size to be produced. Although results are limited to a dozen instructions, those found are already useful. Many examples of these can be found in Appendix I.

One of the most interesting results is not the programs themselves, but a better understanding of the interrelations between arithmetic and logical instructions. Similar ideas seem to come up consistently in the superoptimized programs. These include the sequence '*add.l d1,d1; subx.l d1,d1*' that extracts the sign of a number in the *signum* and *abs* functions and the sequence '*sub.l d1,d0; and.l d2,d0; add.l d1,d0*' that selects one of two values depending on a third in the *min* and *max* functions.

In the future, we hope to explore these ideas further, and compile a list of useful arithmetic-logical idioms that can be concatenated to form optimal or near-optimal programs.

Appendix

I. More Interesting Results

I.1. *SIGNUM* Function

The *signum* function has been defined in section 2. Given the 68000 instruction set, four is the minimum number of instructions to compute *signum*. Interestingly, three suffice on the 8086.

```
(x in ax)
cwd     (sign extends register ax into dx)
neg  ax
adc  dx,dx
(signum(x) in dx)
```

I.2. Absolute Value Function

Find the absolute value of a number, excluding conditional jumps from the instruction set.

```
(x in d0)
move.l   d0,d1
add.l    d1,d1
subx.l   d1,d1
eor.l    d1,d0
sub.l    d1,d0
(abs(x) in d0)
```

Notice that although it is longer than the classical method (test; jump-if-positive; negate), it has no jumps! This might actually be faster than the classical method on some pipelined machines where jumps are expensive.

I.3. Max and Min

This program finds the maximum of the unsigned numbers in d0 and d1 and returns the answer in d0. The comments on the right show what's in the various registers during execution and is similar to the boolean expression checker's method of analysis.

```
(d0=X, d1=Y) |Flag,Reg|If d1>d0   |If d1<=d0
sub.l    d1,d0|(C,d0) =|(1, X-Y)  |(0, X-Y)
subx.l   d2,d2|(C,d2) =|(1,11..11)|(0,0...0)
or.l     d2,d0|(C,d0) =|(1,11..11)|(0,X-Y)
addx.l   d1,d0|d0 =    |Y         |X
(d0 = max(X, Y))
```

This program finds the minimum of the unsigned numbers in d0 and d1 and returns the answer in d0.

```
(d0=X, d1=Y) |Flag,Reg|If d1>d0   |If d1<=d0
sub.l    d1,d0|(C,d0) =|(1, X-Y)  |(0, X-Y)
subx.l   d2,d2|d2 =    |111...111 |000...000
and.l    d2,d0|d0 =    |X-Y       |0
add.l    d1,d0|d0 =    |X         |Y
(d0 = min(X, Y))
```

Simultaneous min and max.

```
(d0=X, d1=Y) |Flag,Reg|If d1>d0   |If d1<=d0
sub.l    d1,d0|(C,d0) =|(1, X-Y)  |(0, X-Y)
subx.l   d2,d2|d2 =    |111...111 |000...000
and.l    d0,d2|d2 =    |X-Y       |0
eor.l    d2,d0|d0 =    |0         |X-Y
add.l    d1,d0|d0 =    |Y         |X
add.l    d2,d1|d1 =    |X         |Y
(d0 = max(X, Y), d1 = min(X, Y))
```

I.4. Logical Tests

Here are some logical tests that yield true/false answers. Sequences such as these have immediate application in a compiler to improve execution speed. Shown here are the tests for zero and non-zero.

```
Suitable for BASIC   Suitable for C, PASCAL

d0 =  0 if d0 == 0   d0 = 0 if d0 == 0
   = -1 if d0 != 0      = 1 if d0 != 0
neg.l    d0          neg.l     d0
subx.l   d0,d0       subx.l    d0,d0
                     neg.l     d0


d0 = -1 if d0 == 0   d0 =  1 if d0 == 0
   =  0 if d0 != 0      =  0 if d0 != 0
neg.l    d0          neg.l   d0
subx.l   d0,d0       subx.l  d0,d0
not.l    d0          addq.l  1,d0
```

By prepending 'move.l A,d0; sub.l B,d0' to the above one can construct tests for A == B and A != B.

I.5. Decimal to Binary

This piece converts a 8 digit BCD number stored in d0, one digit to a nibble, to binary with the result also in d0. It is the longest sequence ever generated by superoptimizer, and was actually done in three stages. The idea that this was even possible came while generating sequences to multiply by 10. At first I had superoptimizer compute the 2 digit BCD to binary conversion function '*((d0 & 0xF0) >> 4) * 10 + (d0 & 0x0F)*'. This came out surprisingly short:

```
(2 digit BCD number in d0)
move.b   d0,d1
and.b    #$F0,d1
lsr.b    #3,d1
sub.b    d1,d0
sub.b    d1,d0
sub.b    d1,d0
(binary equivalent in d0)
```

What is actually being computed is

```
ans = d0 - 3 * ((d0 & 0xF0)/8)
```

Representing the contents of d0 as (H:L) where H is the upper nibble and L is the lower nibble we get

```
d0 = 16 * H + L,    d0 & 0xF0 = 16*H
ans = (16*H+L) - 3 * (16*H/8)
    = 16*H+L - 6*H
    = 10*H + L
```

which is the 2 digit BCD to binary function. Encouraged by this result, superoptimizer was put to the task of computing first the 4 digit BCD to binary function and then the 8 digit BCD to binary function. Here is the 8 digit converter:

```
(8 digit BCD number in d0)
move.l   d0,d1                  *
and.l    #$F0F0F0F0, d1         *
lsr.l    #3,d1                  *
sub.l    d1,d0                  *
sub.l    d1,d0                  *
sub.l    d1,d0                  *
move.l   d0,d1                  +
and.l    #$FF00FF00,d1          +
lsr.l    #1,d1                  +
sub.l    d1,d0                  +
lsr.l    #2,d1                  +
sub.l    d1,d0                  +
lsr.l    #3,d1                  +
add.l    d1,d0                  +
move.l   d0,d1                  -
swap     d1                     -
mulu     #$D8f0,d1              -
sub.l    d1,d0                  -
(binary equivalent in d0)
```

What is most amazing is the first section (marked by * alongside the program) It looks exactly like the 2 digit BCD to binary function. This section computes 4 simultaneous 2 digit BCD to binary functions on adjacent pairs of nibbles and deposits the answer back into the byte occupied by those nibbles. The second part (marked by +) computes two simultaneous 2-byte base 100 to binary conversion functions. Finally, the third part computes the function 'high-word-of-d0 * 10000 + low-word-of-d0' to complete the conversion.

I.6. Multiplication by Constants

During a two week period, superoptimizer was used to find minimal programs that multiply by constants. A sampling of these programs is included in this section.

An interesting observation is that the average program size increases as the multiplication constant increases, but it increases very slowly. The average size of programs that multiply by small numbers (less than 40) is 5 instructions, most programs that multiply by numbers in the hundreds are 6 to 7 instructions long, and programs that multiply by thousands are between 7 and 8 instructions long.

```
d0 *= 29                    d0 *= 39
move.l   d0,d1              move.l   d0,d1
lsl.l    #4,d0              lsl.l    #2,d0
sub.l    d1,d0              add.l    d1,d0
add.l    d0,d0              lsl.l    #3,d0
sub.l    d1,d0              sub.l    d1,d0
```

```
                                d0 *= 625
                               move.l  d0,d1
d0 *= 156                      lsl.l   #2,d0
move.l  d0,d1                  add.l   d1,d0
lsl.l   #2,d1                  lsl.l   #3,d0
add.l   d1,d0                  sub.l   d1,d0
lsl.l   #5,d0                  lsl.l   #4,d0
sub.l   d1,d0                  add.l   d1,d0
```

I.7. Division by Constants

Division turns out to be difficult to optimize. A general divide by constant that works for all 32-bit arguments is too long to realize any time gain over the divide instruction, and is certainly not shorter. Additionally, there doesn't seem to be any nifty arithmetic-logical operations that simplify the process. The generated programs just multiply by the reciprocal of the constant. Since we do an exhaustive search, this negative result can be seen as a confirmation of the inherent high cost of divisions for the instruction sets considered.

The following programs were generated in an attempt to gain insight into binary to BCD algorithms, another area where superoptimizer has had little success. Note that even with the restricted argument range, these are much longer than the multiply programs.

```
d0 = trunc(d0/10)          for d0 = 0..99
move.b d0,d1
add.b  d0,d0 |d0 = 10 * x
lsr.b  #1,d1 |d1 = .1 * x
add.b  d1,d0 |d0 = 10.1 * x
lsr.b  #3,d0 |d0 = .0101 * x
add.b  d1,d0 |d0 = .1101 * x
lsr.b  #3,d0 |d0 = .0001101 * x

d0 = trunc(d0/100)         for d0 = 0..9999
move.w d0,d1
lsr.w  #1,d1 |d1 =   .1 * x
add.w  d0,d0 |d0 = 10 * x
add.w  d0,d1 |d1 = 10.1 * x
lsr.w  #5,d0 |d0 =   .0001 * x
add.w  d1,d0 |d0 = 10.1001 * x
lsr.w  #8,d1 |note: you can't lsr.w #10,d1
lsr.w  #2,d1 |d1 =   .00000000101 * x
sub.w  d1,d0 |d0 = 10.10001111011
lsr.w  #8,d0 |d0 = .0000001010001111011 * x
```

References

[1] Aho, A.V., Sethi, R, Ullman, J.D.
Compilers Principles, Techniques, and Tools.
Addison Wesley, 1986.

[2] Davidson, J.W. and Fraser, C.W.
Automatic Generation of Peephole Optimizations.
In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 111-116.
ACM/SIGPLAN, June, 1984.

[3] Kessler, P.B.
Discovering Machine-Specific Code Improvements.
In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 249-254.
ACM/SIGPLAN, June, 1986.

[4] Krumme, D.W. and Ackley, D.H.
A Practical Method for Code Generation Based On Exhaustive Search.
In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 185-196.
ACM/SIGPLAN, June, 1982.

Macho: Programming With Man Pages

Anthony Cozzie, Murph Finnicum, and Samuel T. King
University of Illinois

Abstract

Despite years of work on programming languages, programming is still slow and error-prone. In this paper we describe Macho, a system which combines a natural language parser, a database of code, and an automated debugger to write simple programs from natural language and examples of their correct execution. Adding examples to natural language makes it easier for Macho to actually generate a correct program, because it can test its candidate solutions and fix simple errors. Macho is able to synthesize basic versions of six out of nine small coreutils from short natural language descriptions based on their man pages and sample runs.

1 Introduction

Programming is hard. Because computers can only execute simple instructions, the programmer must spell out the application's behavior in excruciating detail. Because computers slavishly follow their instructions, any trivial error will result in a crash or, worse, a security exploit. Together they make computer code difficult and time consuming to write, read, and debug.

Programmers write software the same way they do everything else: by imitating other people. The first response to a new problem is often to google it, and ideally find code snippets or examples of library calls. The programmer then combines these chunks of code, writes some test cases, and makes small changes to the program until its output is correct for the inputs he has considered.

Software engineering researchers have developed techniques to help automate each of these parts of the programming process. Code search tools scan through databases of source code to find code samples related to programmer queries. For example, SNIFF [2] uses source code comments to help find snippets of code, and Prospector [4] finds library calls that convert from one language type to another. Automated debugging tools not only help find problems [6], but sometimes even suggest solutions [7]. For example, recent work by Weimer *et al.* [5], describes how to use genetic programming algorithms to modify buggy source code automatically until the modified programs pass a set of test cases.

Although these techniques do save time, the programmer is still responsible for selecting code snippets, arranging them into a program, and debugging the result. In this paper we describe Macho, a system that generates simple Java programs from a combination of natural language, examples (unit tests), and a large repository of Java source code (mostly from Sourceforge projects). It contains four subsystems: a natural language parser that maps English into database queries, a large database that maps programmer abstractions to snippets of Java code, a stitcher that combines code snippets in "reasonable" ways, and an automated debugger that tests the resulting candidate programs against the examples and makes simple fixes automatically.

Because database search and automated debugging are still hard problems with immature tools, Macho's abilities are correspondingly basic. Our current version of Macho was able to synthesize simple versions (no options, one or two arguments) of various Unix core utilities from simple natural language specifications and examples of correct behavior, including versions of ls, pwd, cat, cp, sort, and grep. Macho was unable to generate correct solutions for wget, head, and uniq. Macho is still under construction, but it has already provided us with several interesting results.

Macho is a remarkably simple attack on an extraordinarily difficult task. Natural language understanding is considered one of the hardest problems in Artificial Intelligence with a huge body of current research. Generalizing from examples is similarly difficult. And even once a computer system "understands" the problem it still must actually write suitable Java code.

Our key insight is that natural language and examples have considerable synergy. Macho has a fighting chance

to generate correct programs because each component can partially correct for the mistakes of the others. For example, a database query will return many possible results, most of which will be incorrect, but by leveraging the type system the stitcher can eliminate many unlikely solutions. Even more importantly, the test cases allow Macho to partially detour around the difficult problem of natural language processing. Modern machine learning techniques provide probabilistic answers, whether the question is the meaning of a piece of natural language or the best sample function in the database to use. Backed by its automated debugger, Macho can afford to try multiple solutions.

In addition, combining examples and natural language greatly reduces their ambiguity: the set of programs that satisfies both the natural language and the test cases is much smaller than the sets that satisfy each input individually, although there are some exceptions: Macho found it surprisingly easy to synthesize cat from a unit test using the empty files it used for generating ls. However, we found that most of the time a program that passed even one reasonable test case would be correct. Together natural language and examples form a fairly concrete specification.

2 Architecture

Macho's workflow mirrors a human programmer. It maps the natural language to implied computation, maps those abstractions to concrete Java code, combines the code chunks into a candidate solution, and finally debugs the resulting program. The goal of each subsystem is therefore to minimize the amount of brute force and thereby synthesize the largest possible programs.

2.1 Natural Language Parser

Our natural language parsing subsystem attempts to extract implied chunks of computation and the data flow between them from the words and phrases it receives, and encode that knowledge for the database. Usually the structure of the sentence can be directly transformed to requested computation: verbs imply action, nouns imply objects, and two nouns linked by a preposition imply some sort of conversion code. This mapping is conceptually similar to previous work [1], but Macho's database "understands" a much larger number of concepts, including abbreviations. In order to handle these more varied sentences, we began with an off-the-shelf system provided by the University of Illinois Cognitive Computation group to tag individual words with their part of speech (noun, verb, adjective, etc.) and to split sentences apart into smaller phrases.

Our main problem was fixing the errors of the parser, which was trained on a standard corpus of newspaper articles, not jargon filled man pages. For example, 'file' is usually a verb, like "the SEC filed charges against Enron today." and print is often a noun, e.g., "Their foul prints will not soon be cleansed from the financial system.". These kinds of errors were quite common.

To help detect what words were intended to act as actions, we build a graph of prepositions linking the objects in a sentence together into a tree. A traversal of this tree reveals the relationship between the nouns at its leaves. When we find words that are not linked to the rest of the sentence by this graph, we can guess that they are misclassified verbs. The parser also provides some hints as to likely control flow. For example, plural adjective or adverbial phrases often imply a filter operation that is implemented as an if statement. The description of grep contains 'lines matching a pattern' which implies only some lines will be used.

2.2 Database

As the subsystem that maps natural language abstractions to concrete Java code, the database is the engine that powers Macho. When the database can suggest reasonable code chunks, the stitching can usually find a correct solution, but when the database fails the space of candidate programs is simply too large to succeed by flailing randomly.

Our original plan was to use Google Code, but we almost immediately dismissed it as completely inadequate. Google Code indexes a huge number of files, but it appears to only perform keyword search on the raw text of the source files, which we found to be inadequate for our problem. Instead, we developed our own database for Macho.

Our first step was to obtain a data set of about 200,000 Java files from open source projects and compile them using a special version of javac that we modified to emit abstract syntax trees. We compiled rather than parsed because we wanted exact global locations for each function call, and because we didn't want to reuse broken code. Since open source programmers are not exactly paragons of code maintenance, only about half of our source files compiled successfully.

Our database returns candidate methods based on input and output variables, e.g. the query $directory \rightarrow files$ would return all functions called with an input variable named directory and assigned to a variable named files. This nicely captured the different abstractions that different programmers used to represent code, which is important because functions have only one name. The problem with this approach is that many things aren't usually implemented as functions. Higher level concepts

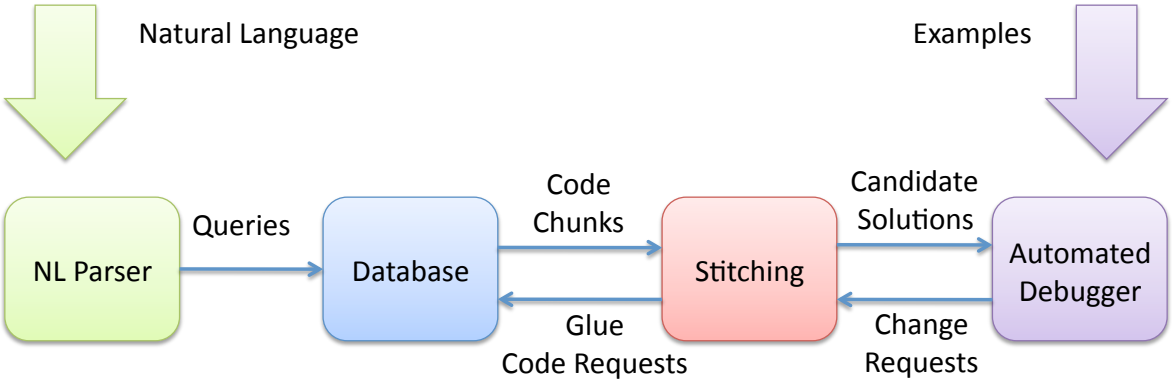


Figure 1: Macho workflow

like ignore, first, or adjacent usually appear as operations or even control flow. Often they have no input variables or are only tagged in the comments.

2.3 Stitching

Macho's stitching subsystem combines results from database queries into candidate programs. Its main guide is the type system; two expressions can be linked by a variable if the output type of one matches the input type of the other. If the types don't match, the stitcher will query the database for common chunks of code that were used to convert between those types.

Macho also generates a small amount of control flow. If statements are generated only from hints by the natural language parser and the synthesizer. Map loops are generated when suggested by the type system. Macho tries to limit control flow generation because it swiftly increases the solution space; an upstream chunk may be placed in any block above the downstream chunk.

The most difficult part of stitching is keeping track of the data flow between expressions in the presence of control flow. The natural language gives a great deal of information for how information is supposed to flow from one chunk to another; previous natural language programming systems generated code without any search at all.

2.4 Automated debugger

Macho's automated debugging subsystem attempts to debug candidate programs. This type of automated debugging is potentially extremely difficult, but many of the automatically generated candidate programs will have utterly obvious errors that can be fixed easily. The primary difference between stitching and automated debugging is that debugging is dynamic rather than static and has access to the behavior of the program. Currently the automated debugger runs the candidate in a sandbox and performs a diff between the output of the candidate and the unit test and classifies the candidate into one of five simple cases: exception thrown (try to insert an if block around the offending statement), a superset of correct output (insert if blocks around the offending print), garbage (try the next program), a subset of correct output (try adding a few prints), or, in the best case, correct output (declare victory).

These components have synergy beyond simply correcting mistakes. For example, our automated debugger leverages the database to suggest changes to buggy programs. When it is faced with a potential solution for ls which incorrectly prints hidden files, the debugger queries the database for commonly used functions of *java.io.File* which could be used in an if statement to restrict the obstreperous print. This simple probabilistic model allows it to try the *isHidden* method even though it is not used elsewhere in the candidate solution.

Although the automated debugging seems superficially simple, it actually solves a very difficult problem of library combination. Macho's database finds candidate functions entirely by name, which may be unrelated to their purpose. Running the code allows the debugger to eliminate these imposter functions.

3 Evaluation

Objectively evaluating Macho is very difficult. There is no standard test suite where we can benchmark our results against other systems, and using the language from the man pages directly is almost impossible. Consider the byzantine man page description for wget:

> GNU Wget is a free utility for non-interactive download of files from the Web. It supports HTTP, HTTPS, and FTP protocols, as well as retrieval through HTTP proxies.

Program	Result	Input	Notes
pwd	success	Print the current working directory.	Difficult as there is no input.
pwd	success	Print the user directory.	CWD = “user.dir” in Java.
pwd	success	Print the current directory.	Abbreviation!
pwd	fail	Print the working directory.	Breaks NLP for arcane reasons.
pwd	fail	Show the current working directory.	Database entries for show are mostly graphics.
cat	success	Print the lines of a file.	Vanilla.
cat	success	Read a file.	Print is synthesized.
cat	fail	Display the contents of a file.	Database entries for contents are mostly graphics.
cat	fail	Print a file	Solutions print the file name.
sort	success	Sort the lines of a file.	Print is synthesized.
sort	success	Sort a file by line.	
sort	fail	Sort a file.	Insufficiently precise specification.
sort	fail	Sort the contents of a file	Database entries for contents are mostly graphics.
grep	success	Print the lines in a file matching a pattern.	Solutions using both JavaLib and GNU regexes.
grep	fail	Find a pattern in the lines of a file.	Correct except for if statement linking test and print.
grep	fail	Search file for a pattern.	Poor resiliency for function names.
ls	success	Print the names of files in a directory. Sort the names.	
ls	success	Print the contents of a folder. Sort the names.	
ls	fail	Print the names of the entries in a directory.	Entries to names fails.
ls	fail	Print the files in a directory.	Does not synthesize sort.
cp	success	Copy src file to dest file.	Programmer abbreviation!
cp	success	Copy file to file.	Ugly but Macho needs to know there are two inputs.
cp	fail	Duplicate file to file.	No candidate in database.
wget	fail	Download file.	Candidates have extra functionality.
wget	fail	Open network connection. Download file.	Macho can’t create buffer transfer loop.
head	fail	Print the first ten lines of a file.	’First’ is incomprehensible.
uniq	fail	Print a file. Ignore adjacent lines.	’Ignore’ and ’adjacent’ don’t map to libraries.
perl	fail	The answer to life, the universe, and everything.	Seems to work, but it’s still running.

Figure 2: Macho’s results for generating select core utils. This figure shows the results for pwd, cat, sort, grep, ls, cp, wget, head, and uniq, and the natural language input we used for each of these programs.

Giving out partial credit is also difficult. Some of Macho’s solutions are very close but not byte identical, but automatically determining whether or not an output is sufficiently close to the test case is approximately as hard as generating the program, an artificial version of the Dunning-Kruger effect. Under these circumstances we decided to try to pick an interesting set of natural language inputs right on the border of Macho’s capabilities and use our best judgement when the test cases were “close”.

Macho succeeded in generating simple versions of six out of nine coreutils - pwd, cat, sort, grep, cp, and ls - and failed to synthesize wget, head, and uniq. For each core utility, we targeted its default behavior: no options and the minimum number of arguments possible. Since we had the programs available anyway, we used them to generate our unit tests. All of the programs had only one short test and the results are shown in Figure 2.

4 Lessons Learned

4.1 The Database is King

Although most of the programs Macho writes are 10-15 lines or less, there are a *lot* of potential 10-line Java programs. Brute force really does not get very far - the ability of the database to select reasonable pieces from the natural language heuristics is absolutely critical. In general, when the stitching failed, it was often reasonable to think of a hack, or a simple fix, or just let it run a little longer, but when the database failed Macho had no hope of ever generating a correct solution. Improving Macho will require a superior database above everything else.

4.2 Pure NLP is Bad

Programming with natural language is generally considered a bad idea because specifying details gradually mutates the natural language into a wordy version of Visual Basic. Consider a natural language spec for ls:

```
Take the path "/home/zerocool/"
If the path is a file, print
it.  Otherwise get the list of
files in the directory.  Sort the
result alphabetically.  Go over
the result from the beginning to
the end:  If the current element's
filename does not begin with ".",
print it.
```

which is our best guess for the input required for Pegasus [3]; it is obvious why most programmers would

prefer to use Python instead. Instead, a Macho programmer can specify the basic task very simply:

```
Print the names of files in a
directory.  Sort the names.
```

Even an almost trivial program like this leaves many details unspecified: should the sort be alphabetically by filename, size, file extension, or date? Should the program print the full path, the relative path, or just the name of the files? Does "files" include subdirectories or hidden files? All of these questions are easily cleared up by an example of correct operation. Such examples not only have a higher information density than tedious pages of pseudocode or UML, but they also reduce the workload of the programmer by allowing him to think about one case at a time, rather than all possible cases. In other words, examples allow a user to be concrete without being formal.

4.3 Interactive Programming is the Answer

A traditional programmer must write code that satisfies all possible inputs his program will encounter, while a Macho Programmer can consider each input individually. Macho therefore not only saves the programmer the work of writing code but also frees the programmer from difficult formal reasoning.

Ideally, however, the programmer would only be required to verify, not generate, concrete values. In this rosy scenario the programmer would input natural language and the system would offer a set of alternatives. The programmer could then reject incorrect cases, or suggest modifications, until eventually a correct program is negotiated. This is important because programming is not simply the act of transferring a mental vision into machine code. In reality, the requirements are fuzzy. Some things are more important than others, and still others can be waived or changed if they are difficult to implement. Interactive programming allows the programmer to take the path of least resistance to a satisfactory program.

Of course, this also requires considerably more accurate program synthesis from pure natural language, as well as much better understanding of general concepts, which no one really knows how to do at the moment.

5 Conclusions

In this paper we have discussed Macho, a system that synthesizes programs from a combination of natural language, unit tests, and a large database of source code samples. A few of our technical findings are that the natural language can give implicit hints about the control flow in a program, variable names contain useful information about the functionality of code, and the automatic debugger can use the database to add new code to a candidate solution.

Macho is a simple proof of concept system, not yet directly useful for most programmers, but it can still synthesize basic versions of six small coreutils. By improving the source code database we believe that Macho can be a practical system for helping programmers.

References

[1] A. W. Biermann and B. W. Ballard. Toward natural language computation. *Comput. Linguist.*, 6(2):71–86, 1980.

[2] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for Java using free-form queries. In *FASE '09: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, pages 385–400, Berlin, Heidelberg, 2009. Springer-Verlag.

[3] R. Knöll and M. Mezini. Pegasus: first steps toward a naturalistic programming language. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 542–559, New York, NY, USA, 2006. ACM.

[4] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI '05: Proceedings of the 2005 Conference on Programming Language Design and Implementation*, pages 48–61, New York, NY, USA, 2005. ACM.

[5] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[6] A. Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–267, London, UK, 1999. Springer-Verlag.

[7] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 272–281, New York, NY, USA, 2006. ACM.

Yesterday, my program worked. Today, it does not. Why?

Andreas Zeller

Universität Passau
Lehrstuhl für Software-Systeme
Innstraße 33, D-94032 Passau, Germany
zeller@acm.org

Abstract. Imagine some program and a number of changes. If none of these changes is applied ("yesterday"), the program works. If all changes are applied ("today"), the program does not work. Which change is responsible for the failure? We present an efficient algorithm that determines the minimal set of failure-inducing changes. Our *delta debugging* prototype tracked down a single failure-inducing change from 178,000 changed GDB lines within a few hours.

1 A True Story

The GDB people have done it again. The new release 4.17 of the GNU debugger [6] brings several new features, languages, and platforms, but for some reason, it no longer integrates properly with my graphical front-end DDD [10]: the arguments specified within DDD are not passed to the debugged program. Something has changed within GDB such that it no longer works for me. Something? Between the 4.16 and 4.17 releases, no less than 178,000 lines have changed. How can I isolate the change that caused the failure and make GDB work again?

The GDB example is an instance of the "worked yesterday, not today" problem: after applying a set of changes, the program no longer works as it should. In finding the cause of the regression, the *differences* between the old and the new configuration (that is, the changes applied) can provide a good starting point. We call this technique *delta debugging*—determining the causes for program behavior by looking at the differences (the *deltas*).

Delta debugging works the better the *smaller* the differences are. Unfortunately, already one programmer can produce so many changes in a day such that the differences are too large for a human to trace—let alone differences between entire releases. In general, conventional debugging strategies lead to faster results.

However, delta debugging becomes an alternative when the differences can be *narrowed down automatically.* Ness and Ngo [5] present a method used at Cray research for compiler development. Their so-called *regression containment* is activated when the automated regression test fails. The method takes ordered changes from a configuration management archive and applies the changes, one after the other, to a configuration until its regression test fails. This narrows the search space from a set of changes to a single change, which can be isolated temporarily in order to continue development on a working configuration.

[0] To appear in *Proc. Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7)*, Toulouse, France, September 1999.

Regression containment is an effective delta debugging technique in some settings, including the one at Cray research. But there are several scenarios where linear search is not sufficient:

Interference. There may be not one single change responsible for a failure, but a *combination of several changes:* each individual change works fine on its own, but applying the entire set causes a failure. This frequently happens when merging the products of parallel development—and causes enormous debugging work.

Inconsistency. In parallel development, there may be *inconsistent configurations*—combinations of changes that do not result in a testable program. Such configurations must be identified and handled properly.

Granularity. A single logical change may affect several hundred or even thousand lines of code, but only a few lines may be responsible for the failure. Thus, one needs facilities to *break changes into smaller chunks*—a problem which becomes evident in the GDB example.

In this paper, we present automated delta debugging techniques that generalize regression containment such that interference, inconsistencies, and granularity problems are dealt with in an effective and practical manner. In particular, our dd^+ algorithm

- detects arbitrary interferences of changes in linear time
- detects individual failure-inducing changes in logarithmic time
- handles inconsistencies effectively to support fine-granular changes.

We begin with a few definitions required to present the basic *dd* algorithm. We show how its extension dd^+ handles inconsistencies from fine-granular changes. Two real-life case studies using our WYNOT prototype[1] highlight the practical issues; in particular, we reveal how the GDB failure was eventually resolved automatically. We close with discussions of future and related work, where we recommend delta debugging as standard operating procedure after any failing regression test.

2 Configurations, Tests, and Failures

We first discuss what we mean by configurations, tests, and failures. Our view of a *configuration* is the broadest possible:

Definition 1 (Configuration). *Let* $\mathcal{C} = \{\Delta_1, \Delta_2, \ldots, \Delta_n\}$ *be the set of all possible changes* Δ_i*. A change set* $c \subseteq \mathcal{C}$ *is called a* configuration.

A configuration is constructed by applying changes to a *baseline.*

Definition 2 (Baseline). *An empty configuration* $c = \emptyset$ *is called a* baseline.

Note that we do not impose any constraints on how changes may be combined; in particular, we do not assume that changes are ordered. Thus, in the worst case, there are 2^n possible configurations for n changes.

To determine whether a failure occurs in a configuration, we assume a *testing function.* According to the POSIX 1003.3 standard for testing frameworks [3], we distinguish three outcomes:

[1] WYNOT = "Worked Yesterday, NOt Today"

– The test succeeds (PASS, written here as ✔)
– The test has produced the failure it was indented to capture (FAIL, ✘)
– The test produced indeterminate results (UNRESOLVED, ?).[2]

Definition 3 (Test). *The function test* : $2^{\mathcal{C}} \rightarrow \{✘, ✔, ?\}$ *determines for a configuration* $c \in \mathcal{C}$ *whether some given failure occurs (✘) or not (✔) or whether the test is unresolved (?).*

In practice, *test* would construct the configuration from the given changes, run a regression test on it and return the test outcome.[3]

Let us now model our initial scenario. We have some configuration "yesterday" that works fine and some configuration "today" that fails. For simplicity, we only consider the changes present "today", but not "yesterday". Thus, we model the "yesterday" configuration as baseline and the "today" configuration as set of all possible changes.

Axiom 1 (Worked yesterday, not today). $test(\emptyset) = ✔$ *("yesterday") and* $test(\mathcal{C}) = ✘$ *("today") hold.*

What do we mean by changes that cause a failure? We are looking for a specific change set—those changes that make the program fail by including them in a configuration. We call such changes *failure-inducing.*

Definition 4 (Failure-inducing change set). *A change set* $c \subseteq \mathcal{C}$ *is* failure-inducing *if*

$$\forall c' \left(c \subseteq c' \subseteq \mathcal{C} \rightarrow test(c') \neq ✔ \right)$$

holds.

The set of all changes $\mathcal{C}$ is failure-inducing by definition. However, we are more interested in finding the *minimal* failure-inducing subset of $\mathcal{C}$, such that removing any of the changes will make the program work again:

Definition 5 (Minimal failure-inducing set). *A failure-inducing change set* $B \subseteq \mathcal{C}$ *is* minimal *if*

$$\forall c \subset B \left(test(c) \neq ✘ \right)$$

holds.

And exactly *this* is our goal: *For a configuration* C*, to find a minimal failure-inducing change set.*

3 Configuration Properties

If every change combination produced arbitrary test results, we would have no choice but to test all 2^n configurations. In practice, this is almost never the case. Instead, configurations fulfill one or more specific *properties* that allow us to devise much more efficient search algorithms.

[2] POSIX 1003.3 also lists UNTESTED and UNSUPPORTED outcomes, which are of no relevance here.

[3] A single test case may take time. Recompilation and re-execution of a program may be a matter of several minutes, if not hours. This time can be considerably reduced by smart recompilation techniques [7] or caching derived objects [4].

The first useful property is *monotony:* once a change causes a failure, any configuration that includes this change fails as well.

Definition 6 (Monotony). *A configuration $\mathcal{C}$ is* monotone *if*

$$\forall c \subseteq \mathcal{C}\ \big(test(c) = \text{✘} \rightarrow \forall c' \supseteq c\ (test(c') \neq \text{✔})\big) \tag{1}$$

holds.

Why is monotony so useful? Because once we know a change set does *not* cause a failure, so do all subsets:

Corollary 1. *Let $\mathcal{C}$ be a monotone configuration. Then,*

$$\forall c \subseteq \mathcal{C}\ \big(test(c) = \text{✔} \rightarrow \forall\, c' \subseteq c(test(c') \neq \text{✘})\big) \tag{2}$$

holds.

Proof. By contradiction. For all configurations $c \subseteq \mathcal{C}$ with $test(c) = \text{✔}$, assume that $\exists c' \subseteq c\big(test(c') = \text{✘}\big)$ holds. Then, definition 6 implies $test(c) \neq \text{✔}$, which is not the case.

Another useful property is *unambiguity:* a failure is caused by only one change set (and not independently by two disjoint ones). This is mostly a matter of economy: once we have detected a failure-inducing change set, we do not want to search the complement for more failure-inducing change sets.

Definition 7 (Unambiguity). *A configuration $\mathcal{C}$ is* unambiguous *if*

$$\forall c_1, c_2 \subseteq \mathcal{C}\ \big(test(c_1) = \text{✘} \wedge test(c_2) = \text{✘} \rightarrow test(c_1 \cap c_2) \neq \text{✔}\big) \tag{3}$$

holds.

The third useful property is *consistency:* every subset of a configuration returns an determinate test result. This means that applying any combination of changes results in a testable configuration.

Definition 8 (Consistency). *A configuration $\mathcal{C}$ is* consistent *if*

$$\forall c \subseteq \mathcal{C}\ \big(test(c) \neq \text{?}\big)$$

holds.

If a configuration does not fulfill a specific property, there are chances that one of its *subsets* fulfills them. This is the basic idea of the *divide-and-conquer* algorithms presented below.

4 Finding Failure-Inducing Changes

For presentation purposes, we begin with the simplest case: a configuration c that is monotone, unambiguous, and consistent. (These constraints will be relaxed bit by bit in the following sections.) For such a configuration, we can design an efficient algorithm

based on *binary search* to find a minimal set of failure-inducing changes. If c contains only one change, this change is failure-inducing by definition. Otherwise, we *partition* c into two subsets c_1 and c_2 and test each of them. This gives us three possible outcomes:

Found in c_1. The test of c_1 fails—c_1 contains a failure-inducing change.
Found in c_2. The test of c_2 fails—c_2 contains a failure-inducing change.
Interference. Both tests pass. Since we know that testing $c = c_1 \cup c_2$ fails, the failure must be induced by the combination of some change set in c_1 and some change set in c_2.

In the first two cases, we can simply continue the search in the failing subset, as illustrated in Table 1. Each line of the diagram shows a configuration. A number i stands for an included change Δ_i; a dot stands for an excluded change. Change 7 is the one that causes the failure—and it is found in just a few steps.

Step	c_i	Configuration								*test*	
1	c_1	1	2	3	4	.	.	.	.	✔	
2	c_2	.	.	.	.	5	6	7	8	✘	
3	c_1	.	.	.	.	5	6	.	.	✔	
4	c_2	.	.	.	.	.	.	7	8	✘	
5	c_1	.	.	.	.	.	.	7	.	✘	7 is found
Result		.	.	.	.	.	.	7	.		

Table 1. Searching a single failure-inducing change

But what happens in case of interference? In this case, we must search in *both halves*—with all changes in the other half remaining applied, respectively. This variant is illustrated in Table 2. The failure occurs only if the two changes 3 and 6 are applied together. Step 3 illustrates how changes 5–7 remain applied while searching through 1–4; in step 6, changes 1–4 remain applied while searching in 5–7.[4]

Step	c_i	Configuration								*test*	
1	c_1	1	2	3	4	.	.	.	.	✔	
2	c_2	.	.	.	.	5	6	7	8	✔	
3	c_1	1	2	.	.	5	6	7	8	✔	
4	c_2	.	.	3	4	5	6	7	8	✘	
5	c_1	.	.	3	.	5	6	7	8	✘	3 is found
6	c_1	1	2	3	4	5	6	.	.	✘	
7	c_1	1	2	3	4	5	.	.	.	✔	6 is found
Result		.	.	3	.	.	6	.	.		

Table 2. Searching two failure-inducing changes

We can now formalize the search algorithm. The function $dd(c)$ returns all failure-inducing changes in c; we use a set r to denote the changes that remain applied.

[4] Delta debugging is not restricted to programs alone. On this LaTeX document, 14 iterations of manual delta debugging had to be applied until Table 2 eventually re-appeared on the same page as its reference.

Algorithm 1 (Automated delta debugging). The *automated delta debugging algorithm* $dd(c)$ is

$$dd(c) = dd_2(c, \emptyset) \quad \text{where}$$

$$dd_2(c, r) = \text{let } c_1, c_2 \subseteq c \text{ with } c_1 \cup c_2 = c,\ c_1 \cap c_2 = \emptyset,\ |c_1| \approx |c_2| \approx |c|/2$$

$$\text{in } \begin{cases} c & \text{if } |c| = 1 \text{ (“found”)} \\ dd_2(c_1, r) & \text{else if } \mathit{test}(c_1 \cup r) = \text{✘} \text{ (“in } c_1\text{”)} \\ dd_2(c_2, r) & \text{else if } \mathit{test}(c_2 \cup r) = \text{✘} \text{ (“in } c_2\text{”)} \\ dd_2(c_1, c_2 \cup r) \cup dd_2(c_2, c_1 \cup r) & \text{otherwise (“interference”)} \end{cases}$$

The recursion invariant (and thus precondition) for dd_2 is $\mathit{test}(r) = \text{✔} \wedge \mathit{test}(c \cup r) = \text{✘}$.

The basic properties of *dd* are discussed and proven in [9]. In particular, we show that $dd(c)$ returns a minimal set of failure-inducing changes in c if c is monotone, unambiguous, and consistent.

Since *dd* is a divide-and-conquer algorithm with constant time requirement at each invocation, *dd*'s time complexity is at worst linear. This is illustrated in Table 3, where only the combination of *all* changes is failure-inducing, and where *dd* requires less than two tests per change to find them. If there is only one failure-inducing change to be found, *dd* even has logarithmic complexity, as illustrated in Table 1.

Step	c_i	Configuration								*test*	
1	c_1	1	2	3	4	.	.	.	.	✔	
2	c_2	.	.	.	.	5	6	7	8	✔	
3	c_1	1	2	.	.	5	6	7	8	✔	
4	c_2	.	.	3	4	5	6	7	8	✔	
5	c_1	1	.	3	4	5	6	7	8	✔	2 is found
6	c_2	.	2	3	4	5	6	7	8	✔	1 is found
7	c_1	1	2	3	.	5	6	7	8	✔	4 is found
8	c_2	1	2	.	4	5	6	7	8	✔	3 is found
9	c_1	1	2	3	4	5	6	.	.	✔	
10	c_2	1	2	3	4	.	.	7	8	✔	
11	c_1	1	2	3	4	5	.	7	8	✔	6 is found
12	c_2	1	2	3	4	.	6	7	8	✔	5 is found
13	c_1	1	2	3	4	5	6	7	.	✔	8 is found
14	c_2	1	2	3	4	5	6	.	8	✔	7 is found
Result		1	2	3	4	5	6	7	8		

Table 3. Searching eight failure-inducing changes

Let us now recall the properties *dd* requires from configurations: monotony, unambiguity, and consistency. How does *dd* behave when c is *not* monotone or when it is ambiguous? In case of interference, *dd* still returns a failure-inducing change set, although it may not be minimal. But maybe surprisingly, a single failure-inducing change (and hence a minimal set) is found even for non-monotone or ambiguous configurations:

– If a configuration is ambiguous, multiple failure-inducing changes may occur; *dd* returns one of them. (After undoing this change set, re-run *dd* to find the next one.)

– If a configuration is not monotone, then we can devise "undoing" changes that, when applied to a previously failing configuration c, cause c to pass the test again. But still, today's configuration is failing; hence, there must be *another* failure-inducing change that is not undone and that can be found by *dd*.

5 Handling Inconsistency

The most important practical problem in delta debugging is *inconsistent configurations*. When combining changes in an arbitrary way, such as done by *dd*, it is likely that several resulting configurations are inconsistent—the outcome of the test cannot be determined. Here are some of the reasons why this may happen:

Integration failure. A change cannot be applied. It may require earlier changes that are not included in the configuration. It may also be in conflict with another change and a third conflict-resolving change is missing.

Construction failure. Although all changes can be applied, the resulting program has syntactical or semantical errors, such that construction fails.

Execution failure. The program does not execute correctly; the test outcome is unresolved.

Since it is improbable that all configurations tested by *dd* have been checked for inconsistencies beforehand, tests may well outcome unresolved during a *dd* run. Thus, *dd* must be extended to deal with inconsistent configurations.

Let us begin with the worst case: after splitting up c into subsets, all tests are unresolved—ignorance is complete. How we increase our chances to get a resolved test? We know two configurations that are consistent: $\emptyset$ ("yesterday") and C ("today"). By applying *less* changes to "yesterday's" configuration, we increase the chances that the resulting configuration is consistent—the difference to "yesterday" is smaller. Likewise, we can remove less changes from "today's" configuration and decrease the difference to "today".

In order to apply less changes, we can partition c into a *larger number of subsets*. The more subsets we have, the smaller they are, and the bigger are our chances to get a consistent configuration—until each subset contains only one change, which gives us the best chance to get a consistent configuration. The disadvantage, of course, is that more subsets means more testing.

To extend the basic *dd* algorithm to work on an arbitrary number n of subsets $c_1, \ldots, c_n$, we must distinguish the following cases:

Found. If testing any c_i fails, then c_i contains a failure-inducing subset. This is just as in *dd*.

Interference. If testing any c_i passes and its *complement* $\bar{c}_i$ passes as well, then the change sets c_i and $\bar{c}_i$ form an interference, just as in *dd*.

Preference. If testing any c_i is unresolved, and testing $\bar{c}_i$ passes, then c_i contains a failure-inducing subset and is *preferred*. In the following test cases, $\bar{c}_i$ must remain applied to promote consistency.

As a preference example, consider Table 4. In Step 1, testing c_1 turns out unresolved, but its complement $\bar{c}_1 = c_2$ passes the test in Step 2. Consequently, c_2 cannot contain a bug-inducing change set, but c_1 can—possibly in interference with c_2, which is why c_2 remains applied in the following test cases.

Step	c_i	Configuration								*test*	
1	c_1	1	2	3	4	.	.	.	.	?	Testing c_1, c_2
2	c_2	.	.	.	.	5	6	7	8	✔	$\Rightarrow$ Prefer c_1
3	c_1	1	2	.	.	5	6	7	8	...	

Table 4. Preference

Try again. In all other cases, we repeat the process with $2n$ subsets—resulting with twice as many tests, but increased chances for consistency.

As a "try again" example, consider Table 5. Change 8 is failure-inducing, and changes 2, 3 and 7 imply each other—that is, they only can be applied as a whole. Note how the test is repeated first with $n = 2$, then with $n = 4$ subsets.

Step	c_i	Configuration								*test*	
1	$c_1 = \bar{c}_2$	1	2	3	4	.	.	.	.	?	Testing c_1, c_2
2	$c_2 = \bar{c}_1$	.	.	.	.	5	6	7	8	?	$\Rightarrow$ Try again
3	c_1	1	2	.	.	.	.	.	.	?	Testing $c_1, \ldots, c_4$
4	c_2	.	.	3	4	.	.	.	.	?	
5	c_3	.	.	.	.	5	6	.	.	✔	
6	c_4	.	.	.	.	.	.	7	8	?	
7	$\bar{c}_1$	.	.	3	4	5	6	7	8	?	Testing complements
8	$\bar{c}_2$	1	2	.	.	5	6	7	8	?	
9	$\bar{c}_3$	1	2	3	4	.	.	7	8	✘	
10	$\bar{c}_4$	1	2	3	4	5	6	.	.	?	$\Rightarrow$ Try again

Table 5. Searching failure-inducing changes with inconsistencies

In each new run, we can do a little *optimizing:* all c_i that passed the test can be excluded from c, since they cannot be failure-inducing. Likewise, all c_i whose complements $\bar{c}_i$ failed the test can remain applied in following tests. In our example, this applies to changes 5 and 6, such that we can continue with $n = 6$ subsets. After testing each change individually, we finally find the failure-inducing change, as shown in Table 6.

Step	c_i	Configuration								*test*	
11	c_1	1	.	.	.	5	6	.	.	✔	Testing $c_1, \ldots, c_6$
12	c_2	.	2	.	.	5	6	.	.	?	
13	c_3	.	.	3	.	5	6	.	.	?	
14	c_4	.	.	.	4	5	6	.	.	✔	
15	c_5	.	.	.	.	5	6	7	.	?	
16	c_6	.	.	.	.	5	6	.	8	✘	8 is found
Result		.	.	.	.	.	.	.	8		

Table 6. Searching failure-inducing changes with inconsistencies (continued)

Note that at this stage, changes 1, 4, 5 and 6 have already been identified as *not* failure-inducing, since their respective tests passed. If the failure had not been induced by change 8, but by 2, 3, or 7, we would have found it simply by excluding all other changes.

To summarize, here is the formal definition of the extended dd^+ algorithm:

Algorithm 2 (Delta debugging with unresolved test cases).
The *extended delta debugging algorithm* $dd^+(c)$ is

$$dd^+(c) = dd_3(c, \emptyset, 2) \quad \text{where}$$

$$dd_3(c, r, n) =$$

let $c_1, \ldots, c_n \subseteq c$ such that $\bigcup c_i = c$, all c_i are pairwise disjoint,
and $\forall c_i\ (|c_i| \approx |c|/n)$;

let $\bar{c}_i = c - (c_i \cup r)$, $t_i = test(c_i \cup r)$, $\bar{t}_i = test(\bar{c}_i \cup r)$,
$c' = c \cap \bigcap\{\bar{c}_i \mid \bar{t}_i = \text{✘}\}$, $r' = r \cup \bigcup\{c_i \mid t_i = \text{✔}\}$, $n' = \min(|c'|, 2n)$,
$d_i = dd_3(c_i, \bar{c}_i \cup r, 2)$, and $\bar{d}_i = dd_3(\bar{c}_i, c_i \cup r, 2)$

$$\text{in} \begin{cases} c & \text{if } |c| = 1 \text{ (“found”)} \\ dd_3(c_i, r, 2) & \text{else if } t_i = \text{✘} \text{ for some } i \text{ (“found in } c_i\text{”)} \\ d_i \cup \bar{d}_i & \text{else if } t_i = \text{✔} \wedge \bar{t}_i = \text{✔} \text{ for some } i \text{ (“interference”)} \\ d_i & \text{else if } t_i = \text{?} \wedge \bar{t}_i = \text{✔} \text{ for some } i \text{ (“preference”)} \\ dd_3(c', r', n') & \text{else if } n < |c| \text{ (“try again”)} \\ c' & \text{otherwise (“nothing left”)} \end{cases}$$

The recursion invariant for dd_3 is $test(r) \neq \text{✘} \wedge test(c \cup r) \neq \text{✔} \wedge n \leq |c|$.

Apart its extensions for unresolved test cases, the dd_3 function is identical to dd_2 with an initial value of $n = 2$. Like dd, dd^+ has linear time complexity (but requires twice as many tests).

Eventually, dd^+ finds a minimal set of failure-inducing changes, provided that they are *safe*—that is, they can either be applied to the baseline or removed from today's configuration without causing an inconsistency. If this condition is not met, the set returned by dd^+ may not be minimal, depending on the nature of inconsistencies encountered. But at least, all changes that are safe and not failure-inducing are guaranteed to be excluded.[5]

6 Avoiding Inconsistency

In practice, we can significantly reduce the risk of inconsistencies by relying on specific *knowledge* about the nature of the changes. There are two ways to influence the dd^+ algorithm:

[5] True minimality can only be achieved by testing all 2^n configurations. Consider a hypothetic set of changes where only three configurations are consistent: yesterday's, today's, and one arbitrary configuration. Only by trying all combinations can we find this third configuration; inconsistency has no specific properties like monotony that allow for more effective methods.

Grouping Related Changes. Reconsider the changes 2, 3, and 7 of Table 5. If we had some indication that the changes imply each other, we could keep them in a common subset as long as possible, thereby reducing the number of unresolved test cases. To determine whether changes are related, one can use

- *process criteria,* such as common change dates or sources,
- *location criteria,* such as the affected file or directory,
- *lexical criteria,* such as common referencing of identifiers,
- *syntactic criteria,* such as common syntactic entities (functions, modules) affected by the change,
- *semantic criteria,* such as common program statements affected by the changed control flow or changed data flow.

For instance, it may prove useful to group changes together that all affect a specific function (*syntactic criteria*) or that occurred at a common date (*process criteria*).

Predicting Test Outcomes. If we have *evidence* that specific configurations will be inconsistent, we can *predict* their test outcomes as unresolved instead of carrying out the test. In Table 5, if we knew about the implications, then only 5 out of 16 tests would actually be carried out.

Predicting test outcomes is especially useful if we can impose an *ordering* on the changes. Consider Table 7, where each change Δ_i implies all "earlier" changes $\Delta_1, \ldots, \Delta_{i-1}$. Given this knowledge, we can predict the test outcomes of steps 2 and 4; only three tests would actually carried out to find the failure-inducing change.

Step	c_i	Configuration								*test*	
1	c_1	1	2	3	4	.	.	.	.	✔	
2	c_2	.	.	.	.	5	6	7	8	(**?**)	predicted outcome
3	c_1	1	2	3	4	5	6	.	.	✔	
4	c_2	1	2	3	4	.	.	7	8	(**?**)	predicted outcome
5	c_1	1	2	3	4	5	6	7	.	✘	7 is found
Result		.	.	.	.	.	.	7	.		

Table 7. Searching failure-inducing changes in a total order

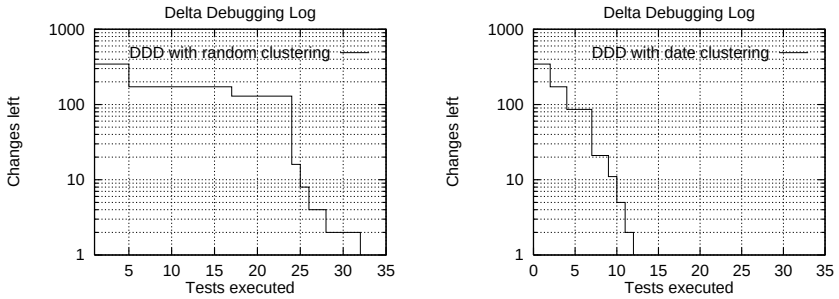
We see that when changes can be ordered, predicting test outcomes makes dd^+ act like a binary search algorithm.

Both grouping and predicting will be used in two case studies, presented below.

7 First Case Study: DDD 3.1.2 Dumps Core

DDD 3.1.2, released in December, 1998, exhibited a nasty behavioral change: When invoked with a the name of a non-existing file, DDD 3.1.2 dumped core, while its predecessor DDD 3.1.1 simply gave an error message. We wanted to find the cause of this failure by using WYNOT.

The DDD configuration management archive lists 116 logical changes between the 3.1.1 and 3.1.2 releases. These changes were split into 344 textual changes to the DDD source.



(a) with random clustering (b) with date clustering

Table 8. Searching a failure-inducing change in DDD

In a first attempt, we ignored any knowledge about the nature or ordering of the changes; changes were ordered and partitioned at random. Table 8(a) shows the result of the resulting WYNOT run. After test #4, WYNOT has reduced the number of remaining changes to 172. The next tests turn out unresolved, so WYNOT gradually increases the number of subsets; at test #16, WYNOT starts using 8 subsets, each containing 16 changes. At test #23, the 7th subset fails, and only its 16 changes remain. Eventually, test #31 determines the failure-inducing change.

We then wanted to know whether knowledge from the configuration management archive would improve performance. We used the following *process criteria:*

1. Changes were grouped according to the date they were applied.
2. Each change implied all earlier changes. If a configuration would not satisfy this requirement, its test outcome would be predicted as unresolved.

As shown in Table 8(b), this resulted in a binary search with very few inconsistencies. After only 12 test runs and 58 minutes[6], the failure-inducing change was found:

```
diff -r1.30 -r1.30.4.1 ddd/gdbinit.C
295,296c296
<   string classpath =
<       getenv("CLASSPATH") != 0 ? getenv("CLASSPATH") : ".";
---
>   string classpath = source_view->class_path();
```

When called with an argument that is not a file name, DDD 3.1.1 checks whether it is a Java class; so DDD consults its environment for the class lookup path. As an "improvement", DDD 3.1.2 uses a dedicated method for this purpose. Unfortunately, the `source_view` pointer used is initialized only later, resulting in a core dump. This problem has been fixed in the current DDD release.

8 Second Case Study: GDB 4.17 does not Integrate

Let us now face greater challenges. As motivated in Section 1, we wanted to track down a failure in 178,000 changed GDB lines. In contrast to the DDD setting from

[6] All times were measured on a Linux PC with a 200 MHz AMD K6 processor.

Section 7, we had no configuration management archive from which to take ordered logical changes.

The 178,000 lines were automatically grouped into 8721 textual changes in the GDB source, with any two textual changes separated by at least two unchanged lines ("context"). The average reconstruction time after applying a change turned out to be 370 seconds. This means that we could run 233 tests in 24 hours or 8721 changes individually in 37 days.

Again, we first ignored any knowledge about the nature of the changes. The result of this WYNOT run is shown in Table 9(a). Most of the first 457 tests turn out unresolved, so WYNOT gradually increases the number of subsets, reducing the number of remaining changes. At test #458, each subset contains only 36 changes, and it is one of these subsets that turns out to be failure-inducing. After this breakthrough, the remaining 12 tests determine a single failure-inducing change.

Running the 470 tests still took 48 hours. Once more, we decided to improve performance. Since process criteria were not available, we used *location criteria* and *lexical criteria* to group changes:

1. At top-level, changes were grouped according to directories. This was motivated by the observation that several GDB directories contain a separate library whose interface remains more or less consistent across changes.
2. Within one directory, changes were grouped according to common files. The idea was to identify compilation units whose interface was consistent with both "yesterday's" and "today's" version.
3. Within a file, changes were grouped according to common usage of identifiers. This way, we could keep changes together that operated on common variables or functions.

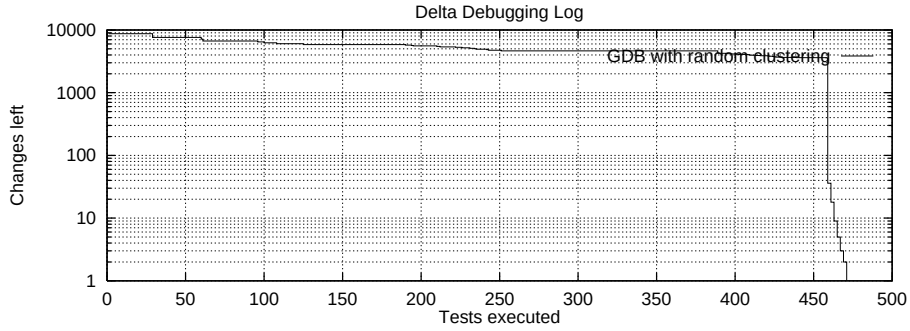
Finally, we added a *failure resolution loop:* After a failing construction, WYNOT scans the error messages for identifiers, adds all changes that reference these identifiers and tries again. This is repeated until construction is possible, or until there are no more changes to add.

The result of this WYNOT run is shown in Table 9(b). At first, WYNOT split the changes according to their directories. After 9 tests with various directory combinations, WYNOT has a breakthrough: the failure-inducing change is to be found in one specific directory. Only 2547 changes are left.

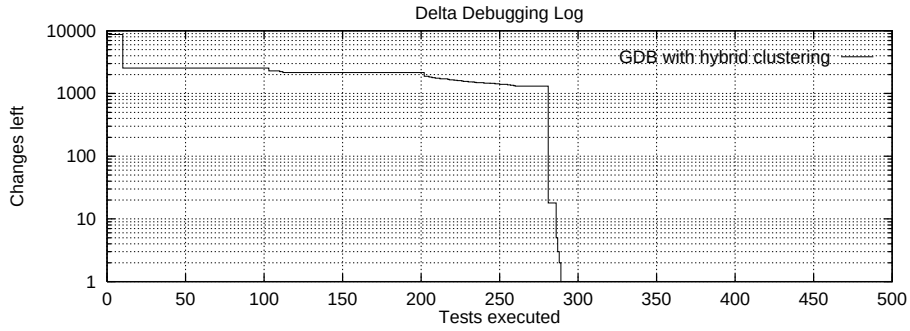
A long period without significant success follows; WYNOT partitions changes into an increasing number of subsets. The second breakthrough occurs at test #280, where each subset contains only 18 changes and where WYNOT narrows down the number of changes to a subset of two files only. The end comes at test #289, after a total of 20 hours. We see that the lexical criteria reduced the number of tests by 38% and the total running time by more than 50%.

In both cases, WYNOT broke down the 178,000 lines down to the same one-line change line that, being applied, causes DDD to malfunction:

```
diff -r gdb-4.16/gdb/infcmd.c gdb-4.17/gdb/infcmd.c
1239c1278
< "Set arguments to give program being debugged when it is started.\n\
---
> "Set argument list to give program being debugged when it is started.\n\
```



(a) with random clustering



(b) with hybrid clustering

Table 9. Searching a failure-inducing change in GDB

This change in a string constant from `arguments` to `argument list` was responsible for GDB 4.17 not interoperating with DDD. Given the command `show args`, GDB 4.16 replies

```
Arguments to give program being debugged when it is started is "a b c"
```

but GDB 4.17 issues a slightly different (and grammatically correct) text:

```
Argument list to give program being debugged when it is started is "a b c"
```

which could not be parsed by DDD! To solve the problem here and now, we simply reversed the GDB change; eventually, DDD was upgraded to make it work with the new GDB version, too.

9 Related Work

There is only one other work on automated delta debugging we have found: the paper on *regression containment* by Ness and Ngo [5], presented in Section 1.[7] Ness and Ngo use simple linear and binary search to identify a single failure-inducing change. Their goal, however, lies not in debugging, but in *isolating* (i.e. removing) the failure-inducing

[7] Ness and Ngo cite no related work, so we assume they found none either.

change such that development of the product is not delayed by resolving the failure. The existence of a configuration management archive with totally ordered changes is assumed; issues like interference, inconsistencies, granularity, or non-monotony are neither handled nor discussed.

Consequently, the failure-inducing change in GDB from Section 8 would not be found at all since there is no configuration management archive from which to take logical changes; in the DDD setting from Section 7, the logical change would be found, but could not have been broken down into this small chunk.

10 Conclusions and Future Work

Delta debugging resolves regression causes automatically and effectively. If configuration information is available, delta debugging is easy; otherwise, there are effective techniques that indicate change dependencies. Although resource-intensive, delta debugging requires no manual intervention and thus saves valuable developer time.

We recommend that delta debugging be an integrated part of regression testing; each time a regression test fails, a delta debugging program should be started to resolve the regression cause. The algorithms presented in this paper provide successful delta debugging solutions that handle difficult details such as interferences, inconsistencies, and granularity.

Our future work will concentrate on avoiding inconsistencies by exploiting domain knowledge. Most simple configuration management archives enforce that each change implies all earlier changes; we want to use full-fledged constraint systems instead [11]. Another issue is to use *syntactic criteria* in order to group changes by affected functions and modules. The most complicated, but most promising approach are *semantic criteria:* Given a change and a program, we can determine a *slice* of the program where program execution may be altered by applying the change. Such slices have been successfully used for semantics-preserving program integration [2] as well as for determining whether a regression test is required after applying a specific change [1]. The basic idea is to determine two *program dependency graphs* (PDGs)—one for "yesterday's" and one for "today's" configuration. Then, for each change c and each PDG, we determine the forward slice from the nodes affected by c. We can then group changes by the *common nodes* contained in their respective slices; two changes with disjoint slices end up in different partitions.

Besides consistency issues, we want to use *code coverage* tools in order to exclude changes to code that is never executed. The intertwining of changes to construction commands, system models, and actual source code must be handled, possibly by multi-version system models [8]. Further case studies will validate the effectiveness of all these measures, as of delta debugging in general.

Acknowledgments. Carsten Schulz contributed significantly to the current WYNOT implementation. The first delta debugging prototype was implemented by Ulrike Heuer. Jens Krinke, Christian Lindig, Kerstin Reese, Torsten Robschink, Gregor Snelting, and Paul Strooper provided valuable comments on earlier revisions of this paper.

Further information on delta debugging, including the full WYNOT implementation, is available at `http://www.fmi.uni-passau.de/st/wynot/`.

References

1. BINKLEY, D. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering 23*, 8 (Aug. 1997), 498–516.
2. BINKLEY, D., HORWITZ, S., AND REPS, T. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology 4*, 1 (Jan. 1995), 3–35.
3. IEEE. *Test Methods for Measuring Conformance to POSIX*. New York, 1991. ANSI/IEEE Standard 1003.3-1991. ISO/IEC Standard 13210-1994.
4. LEBLANG, D. B. The CM challenge: Configuration management that works. In *Configuration Management*, W. F. Tichy, Ed., vol. 2 of *Trends in Software*. John Wiley & Sons, Chichester, UK, 1994, ch. 1, pp. 1–37.
5. NESS, B., AND NGO, V. Regression containment through source code isolation. In *Proceedings of the 21st Annual International Computer Software & Applications Conference (COMPSAC '97)* (Washington, D.C., Aug. 1997), IEEE Computer Society Press, pp. 616–621.
6. STALLMAN, R. M., AND PESCH, R. H. *Debugging with GDB*, 5th ed. Free Software Foundation, Inc., Apr. 1998. Distributed with GDB 4.17.
7. TICHY, W. F. Smart recompilation. *ACM Transactions on Software Engineering and Methodology 8*, 3 (July 1986), 273–291.
8. ZELLER, A. Versioning system models through description logic. In *Proc. 8th Symposium on System Configuration Management* (Brussels, Belgium, July 1998), B. Magnusson, Ed., vol. 1349 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 127–132.
9. ZELLER, A. Yesterday, my program worked. Today, it does not. Why? Computer Science Report 99-01, Technical University of Braunschweig, Germany, Feb. 1999.
10. ZELLER, A., AND LÜTKEHAUS, D. DDD—A free graphical front-end for UNIX debuggers. *ACM SIGPLAN Notices 31*, 1 (Jan. 1996), 22–27.
11. ZELLER, A., AND SNELTING, G. Unified versioning through feature logic. *ACM Transactions on Software Engineering and Methodology 6*, 4 (Oct. 1997), 398–441.

Backwards-Compatible Array Bounds Checking for C with Very Low Overhead*

Dinakar Dhurjati and Vikram Adve
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Ave., Urbana, IL 61801
{dhurjati,vadve}@cs.uiuc.edu

ABSTRACT

The problem of enforcing correct usage of array and pointer references in C and C++ programs remains unsolved. The approach proposed by Jones and Kelly (extended by Ruwase and Lam) is the only one we know of that does not require significant manual changes to programs, but it has extremely high overheads of 5x-6x and 11x–12x in the two versions. In this paper, we describe a collection of techniques that dramatically reduce the overhead of this approach, by exploiting a fine-grain partitioning of memory called Automatic Pool Allocation. Together, these techniques bring the average overhead checks down to only 12% for a set of benchmarks (but 69% for one case). We show that the memory partitioning is key to bringing down this overhead. We also show that our technique successfully detects all buffer overrun violations in a test suite modeling reported violations in some important real-world programs.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.2.5 [**Software**]: Software Engineering—*Testing and Debugging*

General Terms

Reliability, Security, Languages

Keywords

compilers, array bounds checking, programming languages, region management, automatic pool allocation.

1. INTRODUCTION

This paper addresses the problem of enforcing correct usage of array and pointer references in C and C++ programs. This remains an unsolved problem despite a long history of work on detecting array bounds violations or buffer overruns, because the best existing solutions to date are either far too expensive for use in deployed production code *or* raise serious practical difficulties for use in real-world development situations.

The fundamental difficulty of bounds checking in C and C++ is the need to track, at run-time, the intended target object of each pointer value (called the *intended referent* by Jones and Kelly [10]). Unlike safe languages like Java, pointer arithmetic in C and C++ allows a pointer to be computed into the middle of an array or string object and used later to further index into the object. Because such intermediate pointers can be saved into arbitrary data structures in memory and passed via function calls, checking the later indexing operations requires tracking the intended referent of the pointer through in-memory data structures and function calls. The compiler must transform the program to perform this tracking, and this has proved a very difficult problem.

More specifically, there are three broad classes of solutions:

- *Use an expanded pointer representation ("fat pointers") to record information about the intended referent with each pointer*: This approach allows efficient lookup of the pointer but the non-standard pointer representation is incompatible with external, unchecked code, e.g. precompiled libraries. The difficulties of solving this problem in existing legacy code makes this approach largely impractical by itself. The challenges involved are described in more detail in Section 6.

- *Store the metadata separately from the pointer but use a map (e.g., a hash table) from pointers to metadata*: This reduces but does not eliminate the compatibility problems of fat pointers, because checked pointers possibly modified by an external library must have their metadata updated at a library call. Furthermore, this adds a potentially high cost for searching the maps for the referent on loads and stores through pointers.

- *Store only the address ranges of live objects and ensure that intermediate pointer arithmetic never crosses out of the original object into another valid object* [10]: This approach, attributed to Jones and Kelly, stores the address ranges in a global table (organized as a splay tree) and looks up the table (or the splay tree) for the intended referent before every pointer arithmetic operation. This eliminates the incompatibilities

*This work is supported in part by the NSF Embedded Systems program (award CCR-02-09202), the NSF Next Generation Software Program (award CNS 04-06351), and an NSF CAREER award (EIA-0093426).

ICSE'06, May 20–28, 2006, Shanghai, China.

caused by associating metadata with pointers themselves, but current solutions based on this approach have even higher overhead than the previous two approaches. Jones and Kelly [10] report overheads of 5x-6x for most programs. Ruwase and Lam [17] extend the Jones and Kelly approach to support a larger class of C programs, but report slowdowns of a factor of 11x–12x if enforcing bounds for all objects, and of 1.6x–2x for several significant programs even if only enforcing bounds for strings. These overheads are far too high for use in "production code" (i.e., finished code deployed to end-users), which is important if bounds checks are to be used as a security mechanism (not just for debugging). For brevity, we refer to these two approaches as JK and JKRL in this paper.

Note that compile-time checking of array bounds violations via static analysis is not sufficient by itself because it is usually only successful at proving correctness of a fraction (usually small) of array and pointer references [2, 6, 7, 8, 19]. Therefore, such static checking techniques are primarily useful to reduce the number of run-time checks.

An acceptable solution for production code would be one that has no compatibility problems (like the Jones-Kelly approach and its extension), but has overhead low enough for production use. A state-of-the-art static checking algorithm can and should be used to reduce the overhead but we view that as reducing overhead by some constant fraction, for any of the run-time techniques. *The discussion above shows that none of the three current run-time checking approaches come close to providing such an acceptable solution*, with or without static checking.

In this paper, we describe a method that dramatically reduces the run-time overhead of Jones and Kelly's "referent table" method with the Ruwase-Lam extension, to the point that we believe it can be used in production code (and static checking and other static optimizations could reduce the overhead even further). We propose two key improvements to the approach:

1. We exploit a compile-time transformation called Automatic Pool Allocation to greatly reduce the cost of the referent lookups by partitioning the global splay tree into many small trees, while ensuring that the tree to search is known at compile-time. The transformation also safely eliminates many scalar objects from the splay trees, making the trees even smaller.

2. We exploit a common feature of modern operating systems to eliminate explicit run-time checks on loads and stores (which are a major source of additional overhead in the Ruwase-Lam extension). This technique also eliminates a practical complication of Jones and Kelly, namely, the need for one byte of padding on objects and on function parameters, which compromises compatibility with external libraries.

We also describe a few compile-time optimizations (some novel and some obvious) that reduce the sizes of the splay trees, sometimes greatly, or reduce the number of referent lookups. As discussed in Section 3.4, our approach preserves compatibility with external libraries (the main benefits of the JK and JKRL methods) and detects all errors detected by those methods except for references that use pointers cast from integers.

Automatic Pool Allocation uses a pointer analysis to create fine-grain, often short-lived, logical partitions ("pools") of memory objects [13]. By maintaining a separate splay tree for each pool, we greatly reduce the typical size of the trees at each query, and hence the expected cost of the tree lookup. Furthermore, unlike some arbitrary partitioning of memory objects, the properties of pool allocation provide three additional benefits. First, the *target pool* for each pointer variable or pointer field is unique and known at compile-time, and therefore does not have to be found (tracked or searched for) at run-time. Second, because pool allocation often creates type-homogeneous pools, it is possible at run-time to check whether a particular allocation is a single element and avoid entering those objects in the search trees. Finally, we believe that segregating objects by data structure has a tendency to separate frequently searched data from other data, making search trees more efficient (we have not evaluated this hypothesis but it would be interesting to do so).

We evaluate the net overhead of our approach for a collection of benchmarks and three operating system daemons. Our technique works "out-of-the-box" for all these programs, with no manual changes. We find that the average overhead is only about 12% across the benchmarks (and negligible for the daemons), although it is 69% in one case. We also used the Zitser's [23] suite of programs modeling buffer overrun violations reported in several widely used programs — 4 in `sendmail`, 3 in `wu-ftpd`, and 4 in `bind` — and found that our technique successfully detects all these violations. Overall, we believe we have achieved the twin goals that are needed for practical use of array bounds checking in production runs, even for legacy applications: overhead typically low enough for production use, and a fully automatic technique requiring no manual changes.

The next section provides a brief summary of Automatic Pool Allocation and the pointer analysis on which it is based. Section 3 briefly describes the Jones-Kelly algorithm with the Ruwase-Lam extension, and then describes how we maintain and query the referent object maps on a per-pool basis. It also describes three optimizations to reduce the number or cost of referent object queries. Section 5 describes our experimental evaluation and results. Section 6 compares our work with previous work on array bounds checking, and Section 7 concludes with a summary and a brief discussion of possible future work.

2. BACKGROUND: AUTOMATIC POOL ALLOCATION

Automatic Pool Allocation [13] is a fully automatic compile-time transformation that partitions memory into pools corresponding to a compile-time partitioning of objects computed by a pointer analysis. It tries to create pools that are as fine-grained and short-lived as possible. It merges all the target objects of a pointer into a single pool, thus ensuring that there is a unique pool corresponding to each pointer.

We assume that the results of the pointer analysis are represented as a points-to graph. Each node in this graph represents a set of memory objects created at run-time, and two distinct nodes represent disjoint sets of objects. We associate additional attributes with each node; the ones relevant to this work are a type, τ, a flag A indicating whether any of the objects at the node are ever indexed as an array,

and an array of fields, F, one for each possible field of the type τ. τ is either a (program-defined) scalar, array, record or function type, or $\perp$ representing an unknown type. $\perp$ is used when the objects represented by a node are of multiple incompatible types, which most often happens because a pointer value is actually *used* as two different types (cast operations are ignored), but can also happen due to imprecision in pointer analysis. Scalar types and $\perp$ have a single field, record types have a field for each element of the record, array types are treated as their element type (i.e. array indexing is ignored), and functions do not have fields.

We also assume that the compiler has computed a call graph for the program. In our work, we use the call graph implicitly provided by the pointer analysis, via the targets of each function pointer variable.

Given a program containing explicit `malloc` and `free` operations and a points-to graph for the program, Automatic Pool Allocation transforms the program to segregate heap objects into distinct pools. Pools are represented in the code by pool descriptor variables. Calls to `malloc` and `free` are rewritten to call new functions `poolalloc` and `poolfree`, passing in the appropriate pool descriptor. By default, pool allocation creates a distinct pool for each points-to graph node representing heap objects in the program; this choice is necessary for the current work as explained later. For a points-to graph node with $\tau \neq \perp$, the pool created will only hold objects of type τ (or arrays thereof), i.e., the pools will be *type homogeneous* with a known type.

In order to minimize the lifetime of pool instances at runtime, pool allocation examines each function and identifies points-to graph nodes whose lifetime is contained within the function, i.e., the objects are not reachable via pointers after the function returns. This is a simple escape analysis on the points-to graph. The pool descriptor for such a node is created on function entry and destroyed on function exit so that a new pool instance is created every time the function is called. For other nodes, the pool descriptor must outlive the current function so pool allocation adds new arguments to the function to pass in the pool descriptor from the caller. Finally, pool allocation rewrites each function call to pass any pool descriptors needed by any of the potential callees. Ensuring backwards-compatibility of the pool allocation transformation in the presence of external libraries is discussed later in Section 3.4.

We have shown previously that Automatic Pool Allocation can significantly improve memory hierarchy performance for a wide range of programs and does not noticeably hurt performance in other cases [13]. It's compilation times are quite low (less than 3 seconds for programs up to 200K lines of code), and are a small fraction of the time taken by GCC to compile the same programs.

3. RUNTIME CHECKING WITH EFFICIENT REFERENT LOOKUP

3.1 The Jones-Kelly Algorithm and Ruwase-Lam Extension

Jones and Kelly rely on, and strictly enforce, three properties of ANSI C in their approach: (1) Every pointer value at run-time is derived from the address of a unique object, which may be a declared variable or memory returned by a single heap allocation, and must only be used to access that object. Jones and Kelly refer to this as the *intended referent* of a pointer. (2) Any arithmetic on a pointer value must ensure that the source and result pointers point into the same object, or at most one byte past the end of the object (the latter value may be used for comparisions, e.g., in loop termination, but not for loads and stores). (3) Because of the potential for type-converting pointer casts, it is not feasible in general to distinguish distinct arrays within a single allocated object defined above, e.g., two array fields in a `struct` type, and the Jones-Kelly technique does not attempt to do so.

Jones and Kelly maintain a table describing all allocated objects in the program and update this table on `malloc/free` operations and on function entry/exit. To avoid recording the intended referent for each pointer (this is the key to backwards compatibility), they check property (2) strictly on every pointer arithmetic operation, which ensures that a computed pointer value always points within the range of its intended referent. Therefore, the intended referent can be found by searching the table of allocated objects.

More specifically, they insert the following checks (ignoring any later optimization) on each arithmetic operation involving a pointer value:

JK1. check the source pointer is not the invalid value (-2);

JK2. find the referent object for the source pointer value using the table;

JK3. check that the result pointer value is within the bounds of this referent object plus the extra byte. If the result pointer exceeds the bounds, the result -2 is returned to mark the pointer value as invalid.

JK4. Finally, on any load or store, perform checks [JK1-JK3] but JK3 checks the source pointer itself.

Assuming any dereference of the invalid value (-2) is disallowed by the operating system, the last run-time check (JK4) before loads and stores is strictly not necessary for bounds checking. It is, however, a useful check to detect some (but not all) dereferences of pointers to freed memory and pointer cast errors. The most expensive part of these checks is step (JK2), finding the referent object by searching the table. They use a data structure called a splay tree to record the valid object ranges (which must be disjoint). Given a pointer value, they search this tree to find an object whose range contains that value.

If no valid range is found for a given pointer value, the pointer must have been derived from an object allocated by some uninstrumented part of the program, e.g., an external library, or by pointer arithmetic in such a part of the program (since no legal pointer can ever be used to compute an illegal one). Such pointers values cannot be checked and therefore step (JK3) is skipped, i.e., any array bound violations may not be detected.

One complication in their work is that, because a computed pointer may point to the byte after the end of its referent object, the compiler must insert padding of one-byte (or more) between any two objects to distinguish a pointer to the "extra" byte of the first object from a pointer to the second object. They modify the compiler and the `malloc` library to add this extra byte on all allocated objects. Objects can also be passed as function parameters,

however, and inserting padding between two adjacent parameters could cause the memory layout of parameters to differ in checked and unchecked code. To avoid this potential incompatibility, they do not pad parameters to any function call if the call may invoke an unchecked function and do not pad formal parameters in any function that may be called from unchecked code. In the presence of indirect calls via function pointers, the compiler must be conservative about identifying such functions.

A more serious difficulty observed by Ruwase and Lam is that rule (2) above is violated by many C programs (60% of the programs in their experiments), and hence is too strict for practical use. The key problem is that some programs may compute illegal intermediate values via pointer arithmetic but never use them. For example, in the sequence {`q = p+12; r = q-8; N = *r;`}, the value q may be out-of-bounds while r is within bounds for the same object as $*p$. Jones and Kelly would reject such a program at `q = p+12` because the correct referent cannot be identified later (q may point into an arbitrary neighboring object).

Ruwase and Lam extend the JK algorithm essentially by tracking the intended referent of pointers explicitly but only in the case where a pointer moves out of bounds of its intended referent. For every such out-of-bounds pointer, they allocate an object called the OOB (Out-Of-Bounds) object to hold some metadata for the pointer. The pointer itself is modified to point to the OOB object, and the addresses of live OOB objects are also entered into a hash table. This hash table is checked only before accessing the OOB object to ensure it is a valid OOB object address. The OOB object includes the actual pointer value itself plus the address of the intended referent (saved when the pointer first goes out of bounds). All further arithmetic on the pointer is performed on the value in the OOB object. If the pointer value comes back within bounds, the original pointer is restored to its current value and the OOB object is deallocated.

The extra operations required in the Ruwase-Lam extension are: (1) to allocate and initialize an OOB object when a pointer first goes out-of-bounds; (2) on any pointer arithmetic operation, if the pointer value does not have a valid referent *and* cannot be identified as an unchecked object, search the OOB hash table to see if it points to an OOB object, and if so, perform the operation on the value in the OOB object; (3) When an object is deallocated (implicitly at the end of a program scope or explicitly via `free` operation), scan the OOB object hash table to deallocate any OOB objects corresponding to the referent object that is being deallocated.

The first two operations add extra overhead only for out-of-bounds pointers (which would have caused the program to halt with a run-time error in the JK scheme). The third operation is required even in the case of strictly correct program behavior allowed by J-K. Perhaps more importantly, step JK4 of Jones-Kelley, is now necessary for bounds checking since dereferencing OOB objects is disallowed. In particular, if we wish to combine this approach with other techniques for detecting *all* dereferences to freed memory ([20, 4]) or all pointer cast errors ([15, 5]), we would still need to perform JK4 (or a variant which checks that OOB objects are never dereferenced).

3.2 Our Approach

Our approach is based on the Jones-Kelley algorithm with the RL extension, but with two key improvements that greatly reduce the run-time overhead in practice and makes the approach useful in production level systems. In fact, the improvements are dramatic enough that we are even able to use our system for checking all array operations (not just strings), and still achieve much lower overheads than the JK or RL approaches (even compared with the RL approach applied only to strings). The two improvements are: (1) Exploiting Automatic Pool Allocation [13] for much faster searches for referent objects; and (2) An extra level of indirection in the RL approach for OOB pointers that eliminates the need for run-time checks on most loads and stores.

The Jones-Kelley approach, and in turn Ruwase-Lam extension, rely on one splay data structure for the entire heap. Every memory object (except for a few stack objects whose address is not taken) is entered in this big data structure. This data structure is looked up for almost every access to memory or pointer arithmetic operation. For a program with large number of memory objects, the size of the data structure could be very large, making the lookups quite expensive.

The main idea behind our first improvement is to exploit the partitioning of memory created by Automatic Pool Allocation to reduce the size of the splay tree data structures used for each search operation. Instead of using one large splay tree for the entire program, we maintain one splay tree per pool. The size of each individual splay tree is likely to be much smaller than the combined one. Since the complexity of searching the splay tree for uniform accesses is amortized $O(log_2 n)$ (and better for non-uniform accesses), the lookup for each pointer access is likely to be much faster than in the JK or RL approaches.

A key property that makes this approach feasible is that the pool descriptor for each pointer is known at compile-time. Without this, we would have to maintain a run-time mapping from pointers to pools, which would introduce a significant extra cost as well as the same compatibility problems as previous techniques that maintain metadata on pointers.

3.2.1 Algorithm

The steps taken by the compiler in our approach are as follows:

1. First, pool-allocate the program. Let `Pools` be the map computed by the transformation giving the pool descriptor for each pointer variable.

2. For every pointer arithmetic operation in the original program, `p = q + c`, insert a run-time check to test that `p` and `q` have the same referent. We use the function `getreferent(PoolDescriptor *PD, void *p)` to look up the intended referent of a pointer, p. The pool descriptor, `PD`, identifies which splay tree to lookup. For the instruction `p = q + c`, we compute p, then invoke getreferent(Pools[q], q), and finally check that p has the same referent as q using the function call `boundscheck(Referrent *r, void *p)`.

3. The correct pool descriptor for a pointer q may not be known either if the value `q` is obtained from an integer-to-pointer cast or from unchecked code (e.g, as a result of a call to an external function). The latter case is discussed in Section 3.4, below. The two

```
f() {
  A = malloc(...)
  ...
  while(..) {
    ...
    A[i] = ...
  }
}
```

```
f() {
  PoolDescriptor PD
  A = poolalloc(&PD,...)
  ...
  while(..) {
    ...
   Atmp =  getreferent(&PD, A);
   boundscheck(Atmp, A+i);
  }
}
```

Figure 1: Sample code before and after bounds checking instrumentation

cases can be distinguished via the flags on the target points-to graph node: the former case results in a **U** (Unknown) flag while the latter results in a missing **C** (complete) flag, i.e., the node is marked incomplete. In the former case, the pointer may actually point to an object allocated in the main program, i.e., which has a valid entry in the splay tree of some pool, but we do not know which pool at compile-time. We do not check pointer arithmetic on such pointers. This is weaker than Jones-Kelly as it might allow bound violations on such pointers to go undetected.

3.2.2 Handling Non-Heap Data

The original pool allocation transformation only created pools to hold heap-allocated data. We would like to create partitions of globals and stack objects as well, to avoid using large, combined splay trees for those objects. The pointer analysis underlying pool allocation includes points-to graph nodes for all memory objects, including global and stack objects. In our previous work on memory safety, we have extended pool allocation so that it assigns pool descriptors to all global and stack objects as well, *without changing how the objects are allocated*. Pool allocation already created pool descriptors for points-to graph nodes that include heap objects as well as global or stack objects. We only had to modify it to also create "dummy" pool descriptors for nodes that included only global or stack objects. The transformation automatically ensures that the objects are created in the appropriate function (e.g., in `main` if the node includes any globals). We call these "dummy" pool descriptors because no heap allocation actually occurs using them: they simply provide a logical handle to a compiler-chosen subset of memory objects.

For the current work, we have to record each object in the splay tree for the corresponding pool. We do this in `main` for global objects and in the appropriate function for stack-allocated variables (many local variables are promoted to registers and do not need to be stack-allocated or recorded). The bounds checks for operations on pointers to such pools are unchanged.

3.3 Handling Out-Of-Bounds Pointers

The Ruwase-Lam extension to handle OOB pointers requires expensive checks on all loads/stores in the program (before any elimination of redundant checks). In this work, we propose a novel approach to handle out of bounds values (in user-level programs) without requiring checks on any individual loads or stores.

Whenever any pointer arithmetic computes an address outside of the intended referent, we create a new OOB object and enter it into a hash-table recording the OOB object address (just like Ruwase-Lam). We use a separate OOB hash-table per pool, for reasons described below. The key difference is that, instead of returning the address of the newly created OOB object and recording that in the out-of-bounds pointer variable, we return an address from a part of the address space of the program reserved for the kernel (e.g., addresses greater than 0xbfffffff in standard Linux implemenations on 32-bit machines). Any access to this address by a user level programs will cause a hardware trap[1]. Within each pool, we maintain a second hash table, mapping the returned value and the OOB object. Note that we can reuse the high address space for different pools and so we have a gigabyte of address space (on 32 bit linux systems) for each pool for mapping the OOB objects.

A load/store using out of bounds values will immediately result in a hardware trap and we can safely abort the program. However all pointer arithmetic on such values needs to be done on the actual out of bounds value. So on every pointer arithmetic, we first check if the source pointer lies in the high gigabyte. If it is, we lookup the OOB hash map of the pool to get the corresponding OOB object. This OOB object contains the actual out of bounds value. We perform the pointer arithmetic on the actual out of bounds value. If the result after arithmetic goes back in to the bounds of the referent then we return that result. If the result after arithmetic is still out of bounds, we create a new OOB object and store the result in the new OOB. We then map this new OOB to an unused value in the high gigabyte, store the value along with the OOB object in the OOB hash map for the pool and return the value. Note that just like Ruwase-Lam, we need to change all pointer comparisons to take in to account the new out of bound values.

Step 2 in our approach is now modified as follows:
For every pointer arithmetic operation in the original program, `p = q + c`, we first check if q is a value in the high gigabyte. This is an inexpensive check and involves one comparison. There are two possibilities.

- Case 1: q is not in the high giga byte.
  Here we do the bounds check as before but with one key differnce. If the result p is out of bounds of the referent of q, then instead of flagging it as an error, we create a new OOB object to store the out of bounds value just like Ruwase-Lam extension. Now we map this OOB object to a value in the high address space and assign this high address space value to p.

[1]If no such reserved range is available, e.g. we are doing bounds-checking for kernel modules, then we will need to insert checks on individual loads and stores just like the Ruwase-Lam extension.

- Case 2: q is a value in the high address space.
  We do the following new check (from the discussion above): We first get the corresponding OOB object for that address using the hash map in the pool. We then retrieve the actual out of bounds value from the OOB object and do the arithmetic. If the result is within the bounds of the referent then we assign the result to p and proceed. If the result is still outside the bounds of the referent, then we create a new OOB object just like in Case 1.

3.4 Compatibility and Error Detection with External Libraries

Although Automatic Pool Allocation modifies function interfaces and function calls to add pool descriptors, both that transformation and our bounds checking algorithm can be implemented to work correctly and fully automatically with uninstrumented external code (e.g., external libraries), although some out-of-bound accesses may not be detected. First, to preserve compatibility, calls to external functions are left unmodified. Second, in any points-to graph node reachable from an external function (such nodes are marked as "incomplete" by omitting a **C** (Complete) flag), the `poolfree` for the corresponding pool must determine if it is passed a pointer not within its memory blocks (this is a fast search we call it `poolcheck` [5]), and simply pass the pointer through to `free`. Third, if an internal function may be called from external code, we must ensure that the external code calls the original function, not the pool-allocated version. This ensures backwards-compatibility but makes it possible to miss bounds errors in the corresponding function. In most cases, we can directly transform the program to pass in the original function and not the pool-allocated version (this change can be made at compile-time if it passes the function name but may have to be done at run-time if it passes the function pointer in a scalar variable). In the general case (which we have not encountered so far), the function pointer may be embedded inside another data structure. Even for most such functions, the compiler can automatically generate a "`varargs`" wrapper designed to distinguish transformed internal calls from external calls. When this is not possible, we must leave the callback function (and all internal calls to it), completely unmodified.

Except in call-back functions, bounds checks can still be performed within the available program for *all* heap-allocated objects (internal or external). Like JK, we intercept all direct calls to malloc and record the objects in a separate global splay tree. For pointer arithmetic on a pointer to an incomplete node, we check both the splay tree of the recorded pool for that node and the global splay tree. All heap objects must be in one of those trees, allowing us to detect bounds violations on all such objects.

Internal global and stack objects will be recorded in the splay tree for the pool and hence arithmetic on pointers to them can be checked. We cannot check any static or stack objects allocated in external code since we do not know the size of the objects. The JK and JKRL techniques have the same limitation.

3.5 Errors in Calling Standard Library Functions and System Calls

More powerful error checking is possible for uses of recognized standard library functions and system calls. Many bugs triggered inside such functions are due to incorrect usage of library interfaces and not bugs within the library itself. We can guard against these interface bugs by generating wrappers for each potentially unsafe library routine; the wrappers first check the necessary preconditions on buffers passed to the library call and then invoke the actual library call. For example, for a library call like `memcpy(void *s1, const void *s2, size_t n)`, we can generate a wrapper that checks (1) `n` > 0, (2) the object pointed to by `s2` has atleast `n` more bytes starting from `s2` and (2) the object pointed to by `s3` has atleast `n` more bytes starting from `s3`. These checks can be done using the same `getreferent` and `boundscheck` functions as before.

Note that the wrappers referred to here are not for compatibility between checked code and library code, and are only needed if extra bug detection is desired. We have written the wrappers for many of the standard C library functions because our compiler does not yet generate them automatically.

3.6 Optimizations

There are a number of ways to reduce the overheads of our run-time checks further. We briefly describe three optimizations that we have implemented. The first optimization below is specific to our approach because it requires a key property of pool allocation. The other two are orthogonal to the approach for finding referents and can also be used with the Jones-Kelly or Ruwase-Lam approaches.

First, we observe that a very large number of single-element objects (which may be scalars or single-element arrays) are entered into the splay trees in all three approaches. Since a pointer to any such object can be cast and then indexed as a pointer to an array (e.g., an array of bytes), references to all such objects (even scalars) must be checked for bounds violations. While many local scalars of integer or floating point type are promoted to registers, many other local and all global scalars may still stay memory-resident. Entering all such scalars into the search trees is extremely wasteful since few programs ever index into such scalars, legally or illegally. We propose a technique to avoid entering single-element objects into search trees while still detecting bounds violations for such objects.

To achieve this goal, two challenges must be solved: (1) to identify single-element object allocations, and (2) to detect bounds violations even if such objects are not in the splay trees. For the former, we observe that most pools even in C and C++ programs are *type-homogeneous* [13], i.e., all objects in the pool are of a single type or are arrays of that type. For non-type-homogeneous pools, the pool element type is simply a byte. Furthermore, all objects in such a pool are aligned on a boundary that is an exact multiple of the element size. The size of the element type is already recorded in each pool at pool creation time. This means that the run-time can detect allocations of scalars or single-element arrays: these are objects whose size is exactly the size of the pool element type. We simply do not enter such objects into the splay tree in the pool.

For the latter problem, the specific issue is that a referent look-up using a valid pool descriptor will not find the referent object in the splay tree. This can only happen for two reasons: (i) the object was a one-element object, or (ii) the object was an unchecked object or a non-existent object but the pointer being dereferenced was assigned the same pool

during pool allocation. The latter can happen, for example, with code of the form:

```
T* p = some_cond? malloc(..) : external_func(..);
```

Here, the pointer `p` is assigned a valid pool because of the possible malloc, but if it points to an object returned by the external function `external_func`, the referent lookup will not find a valid referent. The same situation arises if the pointer `p` were assigned an illegal value, e.g., from an uninitialized pointer or by casting an integer. To distinguish the first case from the second, we simply use the pool metadata to check if the object is part of the pool. This check, which we call a `poolcheck`, is a key runtime operation in our previous work on memory safety [5], and the pool run-time has been optimized to make it very efficient. Combining these techniques, we can successfully identify and omit single element arrays from the splay trees, and yet detect when they are indexed illegally.

The next two optimizations are far simpler and not specific to our approach. They both exploit the fact that it is very common for a loop nest or recursion to access very few arrays (often one or two) repeatedly. Since all accesses to the same array have the same referrent, we can exploit this locality by using a small lookup cache before each splay tree. We use a two-element cache to record the last two distinct referents accessed in each pool. When an access finds the referent in the cache, it reduces overhead because it avoids the cost of searching the splay tree to find the referrent (we found this to be more expensive even if the search succeeeds at the root), and also of rotating the root node when successive references to the same pool access distinct arrays. It increases the overhead on a cache miss, however, because all cache elements must be compared before searching the splay tree. We experimented with the cache size and found that a two-element cache provided a good balance between these tradeoffs, and improved performance very significantly over no cache or a one-element cache.

The third optimization attempts to achieve the same effect via a compile-time optimization, viz., loop-invariant code motion (LICM) of the referent lookup. (We find that the two-element cache is important even with this optimization because LICM sometimes fails, e.g., with recursion, or if the loop nest is spread across multiple functions, or the referent lookup does not dominate all loop exits. Implementing this optimization is easy because the referent lookup is a pure function: the same pointer argument always returns the same referent object (or none). Therefore, the lookup is loop-invariant if and only if the pointer is loop-invariant.

4. COMPILER IMPLEMENTATION

We have implemented our approach using the LLVM compiler infrastructure [12]. LLVM already includes the implementation of Automatic Pool Allocation, using a context-sensitive pointer analysis called Data Structure Analysis (DSA). We implemented the compiler instrumentation as an additional pass after pool allocation. We also run a standard set of scalar optimizations needed to clean up the output of pool allocation [13]. Because DSA and pool allocation are interprocedural passes, this entire sequence of passes is run at link-time so that they can be applied to as complete a program as possible, excluding libraries available only in binary form. Doing cross-module transformations at link-time is standard in commercial compilers today because it preserves the benefits of separate compilation.

Our implementation includes three optimizations described earlier: leaving out single-element objects from the splay tree in each pool, the two-element cache to reduce searches of the splay tree, and moving loop-invariant referent lookups out of loops. In previous work, we have also implemented an aggressive interprocedural static array bounds checking algorithm, which can optionally be used to eliminate a subset of run-time checks [6].

We compile each application source file to the LLVM compiler IR with standard intra-module optimizations, link the LLVM IR files into a single LLVM module, perform our analyses and insert run-time checks, then translate LLVM back to ANSI C and compile the resulting code using GCC 3.4.4 at -O3 level of optimization. The final code is linked with any external (pre-compiled) libraries.

In terms of compilation time, DSA and Automatic Pool Allocation are both very fast, requiring less than 3 seconds combined for programs up to 130K lines of code that we have tested. This time is in fact a small fraction of the time taken by gcc or g++ at -O3 for the same programs) [13]. The additional compiler techniques for bounds checking described and implemented in this work add negligible additional compile time.

5. EXPERIMENTS

We present an experimental evaluation of our bounds checking technique, with the following goals:

- To measure the net overhead incurred by our approach.
- To isolate the effect of using multiple distinct splay trees and the associated optimizations, which is our key technical improvement over the Ruwase-Lam (and so Jones-Kelley) approaches.
- To evaluate the effectiveness of our approach in detecting known vulnerabilities. For this purpose, we use Zitser's suite of programs modeling vulnerabilities found in real-world software [23].

It is also interesting to confirm the backwards-compatibility of our approach. In our experience so far, we have required *no* changes to any of the programs we have evaluated, i.e., our compiler works on these programs "out-of-the-box." This is similar to Jones-Kelly and Ruwase-Lam but significantly better than other previous techniques that use metadata on pointers, applied to the same programs, discussed in Section 5.3 below.

5.1 Overheads

We have evaluated the run-time overheads of our approach using the Olden [3] suite of benchmarks, and the unix daemons, ghttpd, bsd-fingerd, and wu-ftpd-2.6.2. We use the Olden benchmarks because they are pointer-intensive programs that have been used in a few previous studies of memory error detection tools [20, 15, 21]. We compare our overheads with these and other reported overheads in Section 5.3. The benchmarks and their characteristics are listed in Table 2. The programs are compiled via LLVM and GCC, as described in the previous section. For the benchmarks we used a large input size to obtain reliable measurements. For the daemon programs we ran the server and the client on the same machine to avoid network overhead and measured the response times for client requests.

The "LLVM (base)" column in the table represents execution time when the program is compiled to the LLVM IR with all standard LLVM optimizations (including the standard optimizations used to clean up after pool allocation, but not pool allocation itself), translated back to C code, and the resultant code is compiled directly with GCC -03. The "PA" column shows the time when we run the above passes as well as the pool allocator but do not insert any run-time checks. Notice that in a few cases, pool allocation speeds up the program slightly but doesn't significantly degrade the performance in any of these cases. We use the LLVM(base) column as the baseline for our experiments in calculating the net overhead of our bounds checking approach because we believe that gives the most meaningful comparisons to previous techniques. Since Automatic Pool Allocation can be used as a separate optimization, the PA column could be used as a baseline instead of LLVM(base), but the two are close enough for the benchmarks in the table that we do not expect this choice to affect our conclusions.

The "BoundsCheck" column shows the execution times with bounds checking. Here, we have turned on the three optimizations that we have discussed in Section 3.6: caching on top of the the splay tree, loop invariant code motion, and not storing single-element objects in the splay tree. The "Slowdown" ratio shows the net overhead of our approach relative to the base LLVM. In almost half of the benchmarks, we found that overheads are within 3%. Only two programs (em3d, health) have overheads greater than 25%.

In order to isolate the benefits of smaller splay data structures, we conducted another experiment. The pool allocator pass provides an option to force it to merge all the pools in the program in to one single global pool. This pool uses the same memory allocation algorithm as before but puts all tracked objects into a single splay tree. This allowed us to isolate the effect of using multiple splay trees instead of the single splay tree used by **JK** and **JKRL**. Note that we cannot use optimization 1 (leaving singleton objects out of the splay tree) because after merging pools, type information for the pool is lost and we cannot identify singleton object allocations. The other two optimizations – caching splay tree results and LICM for referent lookups – are used, which is again appropriate because they can also be used with the previous approaches. Columns "PA with one pool" and "PA with one pool + bounds checking" show the execution times of this single-global-pool program without and with our run-time checks, and the last column shows the ratio of these. The benchmark `health` used up all system memory and started thrashing. The main reason is because we could not eliminate singleton objects from the splay tree, making the single global splay tree much larger than the combined splay trees in the original code. Comparing the last column with the column labelled "Our Slowdown Ratio" shows that in atleast three cases (health, mst, perimeter) the overheads when using multiple search data structures is dramatically better (more than 100%) than using a single datastructure for the entire heap. The benefits are also significant in tsp and bisort. The remaining programs show little difference in overheads for the two cases.

5.2 Effectiveness in detecting known attacks

We used Zitser's suite of programs modeling real-world vulnerabilities [23] to evaluate the effectiveness of our approach in detecting buffer overrun violations in real software. The suite consists of 14 model programs, each program containing a real world vulnerability reported in bugtraq. 7 of these vulnerabilties were in `sendmail`, 3 were in `wu-ftpd`, and 4 were in `bind`. This suite has been used previously to compare dynamic buffer overflow detection approaches [22].

The results of our experiments are reported in Figure 5.2. We are able to detect all the vulnerabilities in all three programs out of the box. In each case, the illegal memory reference was detected and the program was halted with a run-time error. The four bugs in bind are not triggered in the main program but in a library routine (e.g. due to passing a negative argument to memcpy). These bugs are automatically detected by our approach using the wrappers described earlier because they are due to incorrect usage of the library functions (and not bugs within the library).

5.3 Performance comparison with previous approaches

Finally, we briefly compare the overheads observed in our work with those reported by other work, to the extent possible. We can make direct comparisons in cases where there are published results for Olden suite of benchmarks. When such numbers are not available, only a rough comparison is possible, and then only in cases where the differences are obviously large. Note also that some previous techniques including [20, 16] detect a wider range of bugs than we do in the current work. Where possible, we try to compare the overheads they incur due to bounds checking alone.

The two previous approaches with no compatibility problems, JK and JKRL, have both reported far higher overheads than ours, as noted in the Introduction. Jones and Kelly say that in practice, most programs showed overheads of 5x-6x. Ruwase and Lam report slowdowns up to a factor of 11x–12x if enforcing bounds for all objects, and up to a factor of 1.6x–2x for several significant programs even if only enforcing bounds for strings. Their overheads are even higher than those of Jones and Kelly because of the additional cost of checking all loads and stores and also of checking for OOB objects that may have to be deallocated as they go out of bounds. While two of our optimizations (the two-element cache and LICM for loop-invariant referent lookups) might reduce these reported overheads, it seems unlikely that they would come close to our reported overheads. Our overheads are dramatically lower than these previous techniques because of a combination of using multiple splay trees (whose benefit was shown earlier), not requiring checks on loads and stores, and the additional optimizations.

Xu. et al [20] have proposed to use metadata for pointer variables that is held in a separate data structure that mirrors the program data in terms of connectivity. They use the metadata to identify both spatial errors (array bounds, uninitialized pointers) and temporal errors (dangling pointer errors). Their average overheads for Olden benchmarks for just the spatial errors are 1.63 while ours are far less at 1.12. Moreover, their approach incurs some difficulties with backwards compatibility, as described in Section 6.

CCured [15] divides the pointers of the program into safe, seq pointers (for arrays) and wild (potentially unsafe) pointers at compile-time. At run-time CCured checks that seq pointers never go out of bounds and wild pointers do not clobber the memory of other objects. While CCured checking for WILD pointers is more extensive than ours, in the case of Olden benchmarks, they did not encounter any wild

Benchmark	LOC	Base LLVM	PA	BoundsCheck	**Our slowdown ratio**	PA with one pool	PA with one pool + boundschecks	**One-pool ratio**
bh	2053	9.146	9.156	9.138	**1.00**	9.175	10.062	**1.10**
bisort	707	12.982	12.454	12.443	**0.96**	12.425	14.172	**1.14**
em3d	557	6.753	6.785	11.388	**1.69**	6.803	11.419	**1.68**
health	725	14.305	13.822	19.902	**1.39**	13.618	-	**-**
mst	617	12.952	12.017	15.137	**1.17**	12.203	28.925	**2.37**
perimeter	395	2.963	2.601	2.587	**0.87**	2.547	6.306	**2.48**
power	763	2.943	2.920	2.928	**0.99**	2.925	2.931	**1.00**
treeadd	385	17.704	17.729	17.310	**0.98**	17.706	21.063	**1.19**
tsp	561	7.086	6.989	7.219	**1.02**	6.978	8.897	**1.27**
AVG					**1.12**			
Applications								
fingerd	336	2.379	2.384	2.475	**1.04**	2.510	2.607	**1.04**
ghttpd	837	11.405	9.423	9.466	**0.83**	11.737	12.182	**1.03**
ftpd	23033	1.551	1.539	1.542	**0.99**	1.551	1.546	**1.00**

Figure 2: Benchmarks and Run-time Overheads. The One-Pool Ratio compared with Our Slowdown Ratio isolates the benefit of partitioning the splay-tree.

Program	No. of vulnerabilities	No. of vulnerabilties detected	No. of vulnerabilties detected with std. lib. check
sendmail	7	7	7
bind	4	0	4
wu-ftpd	3	3	3

Figure 3: Effectiveness of our approach in detecting known attacks/vulnerabilities

pointers [15]. It is important to note, however, that CCured uses garbage collection for dynamic memory management and the overhead due to garbage collection is unknown. The reported average overheads for Olden are 1.28, which is only slightly higher than our observed overheads. However, they needed to change 1287 lines of code in total to achieve these results while our technique works out of the box.

Yong et al [21] describe a technique to identify many illegal write references and free operations via pointers, by identifying a set of pointers that might be unsafe using a pointer-analysis and tagging the memory corresponding to the objects those pointers may point to. They use a shadow memory with 1 tag bit per byte of memory, setting this tag bit on allocations and clearing them on deallocations. They check these tag bits on every write or free of a potentially unsafe pointer, allowing them to detect a number of potential security attacks and some errors such as accesses to a freed memory that has not been reallocated. They report an average overhead of 1.37x for the Olden benchmarks (the fraction of overhead due to array references is unknown). Unlike our work and the previous papers described above, they do not perform any checks on read operations and read operations are far more frequent than writes.

6. RELATED WORK

We focus our comparisons on techniques for run-time bounds checking, and any optimizations directly related to those techniques. We do not discuss existing compile-time techniques for bounds checking here (including our own), because these techniques are complementary and can be used to eliminate some run-time checks in any of the approaches discussed here.

There are a number of debugging tools like `purify` and `valgrind` that use binary instrumentation to detect a wide range of memory referencing errors. Using binary instrumentation allows these tools to add arbitrary metadata to pointers without the compatibility problems of other approaches. These tools, however, incur very high run-time overheads, e.g., often greater than a factor of 10x for `purify` and `valgrind`. Also, in case of purify it does not catch some pointer arithmetic violations if the arithmetic arithmetic yields a pointer to a valid region [10].

A number of other approaches target debugging but work at the source level. These include Loginov's work on runtime type checking [14], Kendall's bcc [11], Steffens' rtcc [18]. All of these approaches focus on debugging and usually performance is not a serious consideration. For instance, the reported overheads for Loginov's work are up to 900%.

Some tools including SafeC [1] and Cyclone [9] use an augmented pointer representation that includes the object base and size of the legal target object for every pointer value. Such "fat pointers" require significant changes to programs to allow the use of external libraries, typically introducing wrappers around library calls to convert pointer representations. Furthermore, writing such wrappers may be impractical for indirect function calls, and for functions that access global variables or other pointers in memory. Unlike the remaining techniques, below, however, fat pointers have the major advantage that there is no cost to find the metadata for each pointer value.

To reduce the compatibility problems caused by fat pointers, several recent systems store pointer metadata separately from the pointer variables themselves, at the cost of significantly greater overhead for finding the metadata associated with each pointer. This approach was used by Patil and Fisher [16], CCured [15], and Xu et al. [20]. Separating the metadata eliminates the potential for program failures mentioned above, and reduces the need for wrappers on library calls. This technique does not require wrappers for pointers passed *to* library functions or pointer values explicitly

returned by such functions. Wrappers are still needed for checked pointers that may be modified indirectly as a side-effect of a library call, because the metadata before the call would be invalid if the call overwrites the pointer. Such wrappers are likely to be needed less often but, if needed, may be impractical to write for the same reasons as with fat pointers, described above. The work of Xu et al. is also more restrictive than ours because they restrict pointer casts between structures of incompatible types. Finally, and most important from a practical viewpoint, all these techniques have significantly higher overhead than ours, as discussed in more detail in Section 5.3.

As noted in the Introduction, the compatibility problems of both fat pointers and pointers with separately stored metadata occur because the metadata is associated with the pointer itself, and not the object that is the target of a pointer. The work of Jones and Kelly [10] and Ruwase and Lam [17] associate metadata with objects instead of pointers, which greatly reduces the compatibility problem. However, the overheads of these two approaches are quite high. As the comparison in Section 5.3 shows, our approach is able to reduce these overheads greatly, sufficient (we believe) for the technique to be used in production code.

7. SUMMARY AND FUTURE WORK

We have described a collection of techniques that dramatically reduce the overhead of an attractive, fully automatic approach for run-time bounds checking of arrays and strings in C and C++ programs. Our techniques are essentially based on a fine-grain partitioning of memory. They bring the average overhead of run-time checks down to only 12% for a set of benchmarks we have evaluated. Thus, we believe we have achieved the twin goals that have not been simultaneously achieved so far: overhead low enough for production use, and fully automatic checking, i.e., not requiring manual effort to circumvent compatibility problems or to assist the compiler's checking techniques.

We have two goals for the future. First, we aim to evaluate our overheads for a wider range of real-world application programs in the future. Second, we aim to integrate our array bounds checks into the SAFECode system [5, 4], which detects pointer cast errors and dangling pointer errors but not all array bounds errors.

8. REFERENCES

[1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 1994.

[2] R. Bodik, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2000.

[3] M. C. Carlisle. *Olden: parallelizing programs with dynamic data structures on distributed-memory machines*. PhD thesis, 1996.

[4] D. Dhurjati and V. Adve. Detecting all dangling pointer uses in production servers. In *International Conference on Dependable Systems and Networks (DSN)*, June 2006.

[5] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2006.

[6] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without garbage collection for embedded applications. *ACM Transactions on Embedded Computing Systems*, Feb. 2005.

[7] N. Dor, M. Rodeh, and M. Sagiv. Cssv: Towards a realistic tool for statically detecting all buffer overflows in c. In *SIGPLAN Conference on Programming Language Design and Implementation*, Sandiego, June 2003.

[8] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM conference on Computer and communications security*, New York, NY, USA, 2003.

[9] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in cyclone. In *Proc. of the 4th international symposium on Memory management (ISMM)*, 2004.

[10] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[11] S. C. Kendall. Bcc: Runtime checking for c programs. In *In Proceedings of the USENIX*, 1983.

[12] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*, San Jose, Mar 2004.

[13] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Chicago, IL, Jun 2005.

[14] A. Loginov, S. H. Yong, S. Horwitz, and T. Reps. Debugging via run-time type checking. *Lecture Notes in Computer Science*, 2001.

[15] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[16] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. *Software–Practice and Experience*, 27(1):87–110, 1997.

[17] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *In Proceedings of the Network and Distributed System Security (NDSS) Symposium*, pages 159–169, Feb. 2004.

[18] J. L. Steffen. Adding run-time checking to the portable c compiler. In *Software: Practice and Experience*, 1992.

[19] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. *SIGSOFT Softw. Eng. Notes*, 28(5):327–336, 2003.

[20] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of c programs. In *Proc. 12th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 117–126, 2004.

[21] S. H. Yong and S. Horwitz. Protecting c programs from attacks via invalid pointer dereferences. In *Foundations of Software Engineering*, 2003.

[22] M. Zhivich, T. Leek, and R. Lippmann. Dynamic buffer overflow detection. In *BUGS : Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[23] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT symposium on Foundations of software engineering*, 2004.