Dynamic Binary Analysis and Instrumentation

or

Building Tools is Easy

Nicholas Nethercote

Trinity College

A dissertation submitted for the degree of

Doctor of Philosophy at the University of Cambridge

Statement of Originality

Most of the work described in this dissertation is my own, but some of the work described was done in collaboration with others.

Chapters 1, 6 and 7 are entirely my own work.

Chapter 2 describes a software system that I implemented large parts of, but which is partly the work of others. Without my contributions to this system, the work described in chapters 3, 4, 5 and 6 would not have been possible. The contributions of others to this system are described within the text in Sections 1.2.3, 2.1.3 and 2.4.1. Chapter 2 is partly based on a publication [82] that I co-authored with Julian Seward.

Chapters 3, 4 and 5 describe software tools that I designed and implemented. Chapter 3 contains one section (Section 3.4) that is derived from a publication [80] that I co-authored with Alan Mycroft. Chapter 4 is based on a publication [79] that I co-authored with Jeremy Fitzhardinge. Chapter 5 is based on a publication [81] that I co-authored with Alan Mycroft.

The section "Author Publications" below lists all these publications. They are also listed in the bibliography.

All the words, tables and figures within are my own.

No part of this dissertation has been submitted, or is being concurrently submitted, for a degree or diploma or any other qualification at this or any other university.

This dissertation does not exceed 60,000 words, including tables and footnotes, but excluding diagrams and the bibliography.

Nicholas Nethercote,
11th November 2004.

Author Publications

Parts of the research presented in this dissertation have been previously published or presented in the following papers.

Nicholas Nethercote and Alan Mycroft. The cache behaviour of large lazy functional programs on stock hardware. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance (MSP 2002)*, pages 44–55, Berlin, Germany, July 2002.

Nicholas Nethercote and Alan Mycroft. Redux: A dynamic dataflow tracer. In *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA, July 2003.

Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA, July 2003.

Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-checking entire programs without recompiling. In *Informal Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2004)*, Venice, Italy, January 2004.

Abstract

Dynamic binary analysis (DBA) tools such as profilers and checkers help programmers create better software. *Dynamic binary instrumentation* (DBI) frameworks make it easy to build new DBA tools. This dissertation advances the theory and practice of dynamic binary analysis and instrumentation, with an emphasis on the importance of the use and support of *metadata*.

The dissertation has three main parts.

The first part describes a DBI framework called Valgrind which provides novel features to support *heavyweight* DBA tools that maintain *rich metadata*, especially *location metadata*—the shadowing of every register and memory location with a *metavalue*. Location metadata is used in *shadow computation*, a kind of DBA where every normal operation is shadowed by an abstract operation.

The second part describes three powerful DBA tools. The first tool performs detailed cache profiling. The second tool does an old kind of dynamic analysis—bounds-checking—in a new way. The third tool produces *dynamic data flow graphs*, a novel visualisation that cuts to the *essence* of a program's execution. All three tools were built with Valgrind, and rely on Valgrind's support for heavyweight DBA and rich metadata, and the latter two perform shadow computation.

The third part describes a novel system of semi-formal descriptions of DBA tools. It gives many example descriptions, and also considers in detail exactly what dynamic analysis is.

The dissertation makes six main contributions.

First, the descriptions show that metadata is the key component of dynamic analysis; in particular, whereas static analysis predicts approximations of a program's future, dynamic analysis remembers approximations of a program's past, and these approximations are exactly what metadata is.

Second, the example tools show that rich metadata and shadow computation make for powerful and novel DBA tools that do more than the traditional tracing and profiling.

Third, Valgrind and the example tools show that a DBI framework can make it easy to build heavyweight DBA tools, by providing good support for rich metadata and shadow computation.

Fourth, the descriptions are a precise and concise way of characterising tools, provide a directed way of thinking about tools that can lead to better implementations, and indicate the theoretical upper limit of the power of DBA tools in general.

Fifth, the three example tools are interesting in their own right, and the latter two are novel.

Finally, the entire dissertation provides many details, and represents a great deal of condensed experience, about implementing DBI frameworks and DBA tools.

Acknowledgements

I must thank many people. Without their help and support, this work would not have been possible.

Julian Seward, for having the audacity and ability to create Valgrind; for letting me rip it to pieces; and for answering a million questions.

Alan Mycroft, for his experience; for stimulating discussions; for boundless optimism during the slow patches; and for telling me when to stop hacking.

Jeremy Fitzhardinge, Tom Hughes, Josef Weidendorfer, Dirk Müller, Robert Walsh, Paul Mackerras, Donna Robinson, and all the others who have contributed to Valgrind.

Jeremy Singer, Tom Stuart, and the other members of the Cambridge Programming Research Group, for their fine company.

Trinity College, Cambridge, for supporting me financially and providing a home and community.

And finally, my family and Christine Vogel, for their love and support, and doing their best to understand what it is that I study.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

This dissertation advances the theory and practice of dynamic binary analysis and instrumentation, with an emphasis on the importance of the use and support of metadata.

1.1 Background

Programming is difficult, especially in languages like C and C++ that are low-level and provide little protection against common programming errors. As both software and hardware systems grow increasingly complex, programmers need more help than ever. Tools that can be used to improve program quality, particularly correctness and speed, are therefore invaluable. Many such tools use some kind of *program analysis* to determine interesting information about programs.

This dissertation is largely about two things: *dynamic binary analysis* (DBA), a particular kind of program analysis; and *dynamic binary instrumentation* (DBI), a particular implementation technique for DBA. This section shows how these two things fit into the wide world of program analysis.

1.1.1 Static Analysis vs. Dynamic Analysis

Program analyses can be categorised into two groups according to when the analysis occurs.

1. *Static analysis* involves analysing a program's source code or machine code without running it. Many tools perform static analysis, in particular compilers; examples of static analyses used by compilers include analyses for correctness, such as type checking, and analyses for optimisation, which identify valid performance-improving transformations. Also, some stand-alone static analysis tools can identify bugs or help visualise code. Tools performing static analysis only need to read a program in order to analyse it.

2. *Dynamic analysis* involves analysing a *client program* as it executes. Many tools perform dynamic analysis, for example, profilers, checkers and execution visualisers. Tools performing

dynamic analysis must *instrument*[1] the client program with *analysis code*. The analysis code may be inserted entirely inline; it may also include external routines called from the inline analysis code. The analysis code runs as part of the program's normal execution, not disturbing the execution (other than probably slowing it down), but doing extra work "on the side", such as measuring performance, or identifying bugs.[2] The analysis code must maintain some kind of analysis state, which I call *metadata* (and individual pieces of metadata are *metavalues*). Metadata is absolutely crucial, and at the very heart of dynamic analysis, as this dissertation will show.

The two approaches are complementary. Static analysis can be sound, as it can consider all execution paths in a program, whereas dynamic analysis is unsound in general, as it only considers a single execution path [41]. However, dynamic analysis is typically more precise than static analysis because it works with real values "in the perfect light of run-time" [38]. For the same reason, dynamic analyses are often much simpler than static analyses.

This dissertation focuses on dynamic analysis, and does not consider static analysis any further.

1.1.2 Source Analysis vs. Binary Analysis

Program analyses can be categorised into another two groups, according to the type of code being analysed.

1. *Source analysis* involves analysing programs at the level of source code. Many tools perform source analysis; compilers are again a good example. This category includes analyses performed on program representations that are derived directly from source code, such as control-flow graphs. Source analyses are generally done in terms of programming language constructs, such as functions, statements, expressions, and variables.

2. *Binary analysis* involves analysing programs at the level of machine code, stored either as object code (pre-linking) or executable code (post-linking). This category includes analyses performed at the level of executable intermediate representations, such as byte-codes, which run on a virtual machine. Binary analyses are generally done in terms of machine entities, such as procedures, instructions, registers and memory locations.

As is the case with static analysis and dynamic analysis, the two approaches are complementary. Source analysis is platform- (architecture and operating system) independent, but language-specific; binary analysis is language-independent but platform-specific. Source code analysis has access to high-level information, which can make it more powerful; dually, binary analysis has access to low-level information (such as the results of register allocation) that is required for some tasks. One definite

[1]The verb "to instrument" is widely used to refer to the act of adding extra code to a program, and I will use it in this way. The code added during instrumentation is often called "instrumentation code"; however, this term is sometimes used to describe the code doing the instrumentation, or is used to refer only to specific kinds of added code that measure the client's performance in some way. I will use "analysis code" to refer to any code added for the purpose of doing dynamic analysis, as its meaning is unambiguous.

[2]My definition of "dynamic analysis" excludes tools that actively modify a client's semantics, e.g. by preventing certain actions from occurring. Section 2.7.1 discusses this distinction further.

	Static	Dynamic
Source	Static source analysis	Dynamic source analysis
Binary	Static binary analysis	Dynamic binary analysis

Table 1.1: Four kinds of program analysis

advantage of binary analysis is that the original source code is not needed, which can be particularly important for dealing with library code, for which the source code is often not available on systems.

This dissertation focuses on binary analysis of machine code, and does not consider source analysis or byte-code binary analysis any further.

1.1.3 Four Kinds of Program Analysis

The pair of two-way categorisations described in Sections 1.1.1 and 1.1.2 together divide program analysis into four categories: static source analysis, dynamic source analysis, static binary analysis, and dynamic binary analysis. Table 1.1 shows this categorisation.

The program analysis discussed in this dissertation is almost exclusively dynamic binary analysis (DBA), i.e. analysis of machine code that occurs at run-time (although Chapter 6 partly discusses dynamic analysis in general).

1.1.4 Static Binary Instrumentation vs. Dynamic Binary Instrumentation

Section 1.1.1 explained that dynamic analysis requires programs to be instrumented with analysis code. There are two main instrumentation techniques used for DBA, which are distinguished by when they occur.

1. *Static binary instrumentation* occurs before the program is run, in a phase that rewrites object code or executable code.

2. *Dynamic binary instrumentation* (DBI) occurs at run-time. The analysis code can be injected by a program grafted onto the client process, or by an external process. If the client uses dynamically-linked code the analysis code must be added after the dynamic linker has done its job.

(This dissertation only considers instrumentation techniques implementable purely in software; for example, it does not consider techniques that require custom hardware, or microcode manipulation.)

Dynamic binary instrumentation has two distinct advantages. First, it usually does not require the client program to be prepared in any way, which makes it very convenient for users. Second, it naturally covers all client code; instrumenting all code statically can be difficult if code and data are mixed or different modules are used, and is impossible if the client uses dynamically generated code. This ability to instrument all code is crucial for correct and complete handling of libraries. These advantages of DBI make it the best technique for many dynamic analysis tools. However, DBI has two main disadvantages. First, the cost of instrumentation is incurred at run-time. Second, it can be difficult to implement—rewriting executable code at run-time is not easy. Nonetheless, in recent years these problems have been largely overcome by the advent of several generic DBI frameworks, which are carefully optimised to minimise run-time overheads, and with which new DBA tools can be built with relative ease.

This dissertation focuses on dynamic binary instrumentation, and does not consider static binary instrumentation any further.

1.2 This Dissertation

This dissertation is largely about two things: dynamic binary analysis, the analysis of programs at run-time, at the level of machine code; and dynamic binary instrumentation, the technique of instrumenting a program with analysis code at run-time. This dissertation advances the theory and practice of dynamic binary analysis and instrumentation, with an emphasis on the importance of the use and support of metadata.

1.2.1 Dissertation Structure

This dissertation has three main parts.

Chapter 2 makes up the first main part, which describes a DBI framework called Valgrind which provides novel features to support *heavyweight* DBA tools that maintain *rich metadata*, especially *location metadata*—the shadowing of every register and memory location with a metavalue. Location metadata is used in *shadow computation*, a kind of DBA where every normal operation is shadowed by an abstract operation.

Chapters 3, 4 and 5 make up the second main part, which describes three powerful DBA tools. The first tool performs detailed cache profiling. The second tool does an old kind of dynamic analysis—bounds-checking—in a new way. The third tool produces *dynamic data flow graphs*, a novel visualisation that cuts to the *essence* of a program's execution. All three tools were built with Valgrind, and rely on Valgrind's support for heavyweight DBA and rich metadata, and the latter two perform shadow computation.

Chapter 6 makes up the third main part, which describes a novel system of semi-formal descriptions of DBA tools. It gives many example descriptions, and also considers in detail exactly what dynamic analysis is.

After the three main parts, Chapter 7 discusses future work and concludes, and Appendix A contains a glossary of important terms and acronyms used throughout the dissertation, some of which this dissertation introduces. Also, note that Section 1.1 gave only a cursory overview of DBI and DBA, and related work for each of the topics covered in this dissertation is presented in the relevant chapter, rather than in a single section.

1.2.2 Contributions

The dissertation makes six main contributions.

First, the descriptions show that metadata is the key component of dynamic analysis; in particular, whereas static analysis predicts approximations of a program's future, dynamic analysis remembers approximations of a program's past, and these approximations are exactly what metadata is. The importance and nature of metadata in dynamic analysis has not been previously recognised.

Second, the example tools show that rich metadata and shadow computation make for powerful and novel DBA tools that do more than the traditional tracing and profiling. Tools using such heavyweight techniques can perform much deeper analyses and tell us things about programs that more lightweight DBA tools cannot.

Third, Valgrind and the example tools show that a DBI framework can make it easy to build heavyweight DBA tools, by providing good support for rich metadata and shadow computation. This makes it much easier to experiment with heavyweight DBA tools, and create powerful new tools that perform novel analyses.

Fourth, the descriptions are a precise and concise way of characterising tools, provide a directed way of thinking about tools that can lead to better implementations, and indicate the theoretical upper limit of the power of DBA tools in general. This is the first time that such a range of dynamic binary analysis tools have been comprehensively characterised in a unified way.

Fifth, the three example tools are interesting in their own right, and the latter two are novel. This is significant because new tools implementing novel dynamic analyses are rare.

Finally, the entire dissertation provides many details, and represents a great deal of condensed experience, about implementing DBI frameworks and DBA tools. This kind of information is of great use to those implementing similar systems.

1.2.3 A Note About Implementations

Many computer science papers describing systems and tools have certain shortcomings. For example, there is often a gap between a proposed design and what has been implemented. This is unfortunate, because an implementation provides easily the best evidence that the design of a complex tool or system is valid. Also, some papers do not even make clear what has been proposed and what has been implemented. Finally, source code is rarely available for studying.

In contrast, the system and tools described in this dissertation have all been implemented. Experimental or prototype implementations are clearly distinguished from robust, well-tested implementations. The few cases where an implementation falls short of a design are clearly noted. All of the code is freely available under the GNU General Public License [45]. Valgrind and the mature tools are publicly available for download [102]. The experimental tools are available on request.

As is consistent with the Statement of Originality at the start of this dissertation, a significant proportion, but not all, of the implementation work described in Chapter 2 was done by me, and all of the implementation work described in Chapters 3–5 was done by me.

CHAPTER 2

A Framework for Building Tools

This chapter describes a dynamic binary instrumentation framework, named Valgrind, which provides unique support for heavyweight dynamic binary analysis.

2.1 Introduction

This chapter describes Valgrind, a DBI framework for the x86/Linux platform. The description emphasises Valgrind's support for heavyweight DBA.

2.1.1 Dynamic Binary Instrumentation Frameworks

Section 1.1.4 described the great advantages of DBI: programs need no preparation (e.g. recompilation or relinking) and all code is naturally covered. It also mentioned two disadvantages: DBI is difficult to implement, and the instrumentation overhead is incurred at run-time.

Generic DBI frameworks mitigate both problems. First, the basic task of adding analysis code is the same for all DBA tools. DBI frameworks mean that new DBA tools do not have to be built from scratch, and the difficult code can be concentrated in the framework, and reused by each new DBA tool. Second, several DBI framework implementations have shown that with the right implementation techniques, the run-time cost can be minimal.

2.1.2 Overview of Valgrind

Valgrind is a DBI framework for the x86/Linux platform. It is designed for building heavyweight DBA tools that are reasonably efficient. The term "heavyweight" here refers to tools that use analysis code that is *pervasive* (e.g. every instruction is instrumented) and *interconnected* (state is passed between pieces of analysis code, particularly at a low, local level), and for tools that track a lot of information, particularly *location metadata*—the shadowing of every register and memory value with a metavalue. This is in contrast to lightweight DBA which involves less complex analysis code, and less rich metadata.

DBA tools are created as plug-ins, written in C, to Valgrind's *core*. The basic view is this:

Valgrind core + tool plug-in = Valgrind tool.[1]

A tool's main job is to instrument code fragments that the core passes to it. Writing a new tool plug-in (and thus a new DBA tool) is relatively easy—certainly much easier than writing a new DBA tool from scratch. Valgrind's core does all the difficult work of executing the client, and also provides many services to tools, to make common tasks such as recording errors easier. Only one tool can be used at a time.

Valgrind was first described in [82]. Valgrind is a robust, mature system. It is used by thousands of programmers on a wide range of software types, including the developers of notable software projects such as OpenOffice, Mozilla, KDE, GNOME, MySQL, Perl, Samba, Unreal Tournament, and for software for NASA's Mars landers and rovers. I have received feedback from two different users who have successfully used Valgrind tools on projects containing 25 million lines of code. The Valgrind distribution contains the core, plus five tools: two memory checkers (one of which is Memcheck, described in Section 2.4.1), a cache profiler (Cachegrind, described in Chapter 3), a memory profiler, and a data race detector. The source code is available [102] under the GNU General Public License (GPL) [45].

2.1.3 Chapter Structure

This chapter is structured as follows. Section 2.2 shows an example of Valgrind's use. Section 2.3 describes how Valgrind's core works. Section 2.4 describes how tool plug-ins work and interact with the core, using Memcheck as an example, and with an emphasis on Valgrind's support for heavyweight DBA. Section 2.5 presents the code sizes for the core and various tool plug-ins. Section 2.6 discusses performance, Section 2.7 considers related work, and Section 2.8 concludes.

Note that Valgrind did not have the modular architecture described in Section 2.1.2 in its first incarnation. The original version was written by Julian Seward, and it was hard-wired to do only memory checking. Shortly after Valgrind was first released, I rewrote large pieces of it to create the modular core/plug-in architecture. I then wrote the tools described in Chapters 3–5, and have done a great deal of general work on it. Others, particularly Jeremy Fitzhardinge, have also made significant contributions to Valgrind's code base since then. Therefore, this chapter describes work that was done partly by myself, and partly by others.

2.2 Using Valgrind

Valgrind tools are very easy to use. They are usually invoked from the command line. To run the program `uptime` under the Valgrind tool Memcheck (described in Section 2.4.1) which does memory

[1] I will use these three terms in this fashion, and "tool" as shorthand for "Valgrind tool", and "Valgrind" to refer generally to the entire framework. In practice, the terms "Valgrind" and "core" are used interchangeably, as are "tool plug-in" and "tool" as the distinction is often unimportant.

checking, one uses the following command:

```
valgrind --tool=memcheck uptime
```

The following start-up message is printed first.

```
==8040== Memcheck, a memory error detector for x86-linux.
==8040== Copyright (C) 2002-2004, and GNU GPL'd, by Julian Seward et al.
==8040== Using valgrind-2.1.2, a program supervision framework for x86-linux.
==8040== Copyright (C) 2000-2004, and GNU GPL'd, by Julian Seward et al.
==8040== For more details, rerun with: -v
==8040==
```

By default the output goes to standard error, although it can be redirected to a file, file descriptor, or socket with a command line option. The first two lines are tool-specific, the remainder are always printed. Each line is prefixed with `uptime`'s process ID, 8040 in this case.

The program then runs under Memcheck's control, typically about 25 times slower than normal. Other tools cause different slow-downs. All code is covered; dynamically linked libraries and the dynamic linker are run under Memcheck's control just the same as the main executable. Memcheck may issue messages about found errors as it proceeds. In this case, Memcheck issues no error messages, `uptime` prints its output to standard output, as normal:

```
 16:59:38  up 21 days,  3:49, 12 users,  load average: 0.00, 0.02, 0.18
```

and then terminates. Memcheck terminates with the following summary message.

```
==8040==
==8040== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 17 from 1)
==8040== malloc/free: in use at exit: 0 bytes in 0 blocks.
==8040== malloc/free: 43 allocs, 43 frees, 6345 bytes allocated.
==8040== For a detailed leak analysis,  rerun with: --leak-check=yes
==8040== For counts of detected errors, rerun with: -v
```

No error messages were issued, but the second line indicates that Memcheck detected one error seventeen times, but *suppressed* (ignored) it. This is an error that occurs in a system library that is out of the control of the typical user, and thus not of interest. Memcheck suppresses a number of such errors. Section 2.4.5 discusses suppressions in more detail.

All Valgrind tools work in the same basic way, although the information they emit varies. The information emitted can be used by the programmer to fix bugs, or improve performance, or for whatever other purpose the tool was designed for.

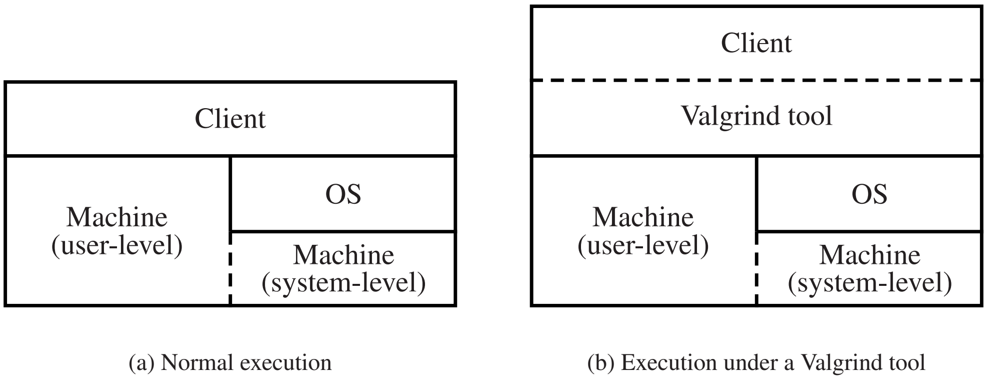


Figure 2.1: Conceptual view of program execution

2.3 How Valgrind Works: The Core

Valgrind's core provides the base execution mechanism for running and controlling client programs. This section describes all its main features except those involving tool plug-ins and instrumentation, which are covered in Section 2.4. Valgrind is a complex system; I have endeavoured to describe its various parts in an order that puts the more fundamental parts first, but without using too many forward references to later sections.

2.3.1 Overview

Valgrind uses *dynamic binary compilation and caching*. A tool grafts itself into the client process at start-up, and then (re)compiles the client's code, one basic block at a time, in a just-in-time, execution-driven fashion. The compilation process involves disassembling the machine code into an intermediate representation (IR) which is instrumented by the tool plug-in, and then converted back into x86 code. The result is called a *translation*[2], and is stored in a code cache to be rerun as necessary. The core spends most of its execution time making, finding, and running translations. None of the client's original code is run.

Because Valgrind is execution-driven, almost all code is handled naturally without difficulty; this includes normal executable code, dynamically linked libraries, and dynamically generated code. The only code not under a tool's control is system calls, but even they can be indirectly observed. The only code that can cause problems is self-modifying code (which is discussed in Section 2.3.13).

Dynamic compilation and caching can be viewed as an alternative to interpreted execution with a different time/space trade-off; by taking the extra space to store the compiled code, one avoids having to repeat operations such as instruction decoding. Also, by translating entire basic blocks, performance can be further improved with intra-basic-block optimisations.

Figure 2.1(a) gives a conceptual view of normal program execution, from the point of view of the client. The client can directly access the user-level parts of the machine (e.g. general-purpose registers), but can only access the system-level parts of the machine through the operating system

[2]A slightly misleading name, as the resulting code is x86 code, the same as the original code.

(OS), using system calls. Figure 2.1(b) shows how this changes when a program is run under the control of a Valgrind tool. The client and tool are part of the same process, but the tool mediates everything the client does, giving it complete control over the client.

There are many complications that arise from effectively squeezing two programs—the client and the tool—into a single process. Many resources have to be shared, such as registers and memory. Also, Valgrind must be careful not to relinquish its control over the client in the presence of things like system calls, signals, and threads. The following sections describe the basic operations and these complications in detail.

2.3.2 Definition of a Basic Block

Because Valgrind is execution-driven, the client's code is compiled on demand, one basic block at a time, just before the basic block is executed. The meaning of "basic block" here is a straight-line sequence of x86 code, whose head is jumped to, and which ends in a control flow transfer such as a jump, call, or return. I will refer to particular basic blocks as BB_1, BB_2, etc., and the translations of these basic blocks as $t(BB_1)$, $t(BB_2)$, etc.

Note that this definition is different to the usual meaning of the term; in particular, a jump can land in the middle of a basic block. If control transfers to the middle of a basic block that has been previously translated, the second half of the basic block will be retranslated. A similar thing happens if control transfers first to the middle of a basic block, and then later to the start. In practice, measurements show that typically only about 2% of code is re-translated because of this.

2.3.3 Resource Conflicts

Because the tool becomes part of the client's process, resource conflicts are a big issue. All the resources that a client has to itself when executing normally must be shared with the tool. Valgrind handles each of the various resource conflicts in one of the following ways.

- *Partitioning* (or *space-multiplexing*) involves breaking up the resource-space into separate parts. Section 2.3.4 shows how this works for address space. Valgrind's core intercepts and manages all system calls that involve memory mapping—`brk()`, `mmap()`, `munmap()`, `mprotect()`, `shmat()`, and `shmdt()`—to enforce the partitioning. It also executes all (translated) client code within a segment (using the x86 segment registers) to prevent the client wildly writing to the tool's address space. (It is assumed Valgrind and the tool do not do wild writes into the client's address space.) The tool also uses a separate stack from the client.

  File descriptors are another partitioned resource; the core reserves a small number at the top of the allowed range for itself and the tool plug-in, and prevents the client from using them.

- *Time-multiplexing* involves sharing a resource by letting the client use it some of the time, and the tool use it some of the time. The prime example is registers. First, the *client state* is the client's values for the general-purpose registers (including the stack and frame pointers), the condition code register `%eflags`, the floating-point registers, and the MMX and SSE registers.

Second, the tool may track shadow values (also called metavalues) for each register, and it may also need spare registers to hold temporary values in analysis code. Finally, Valgrind's core needs some registers for its own basic operation.

All these values obviously cannot fit in the register file at once, so sometimes they must be spilled to memory. In fact, between basic blocks, the core keeps all client state and any shadow state in a block of memory called the `baseBlock`. All state gets loaded into the real machine registers as needed, after the beginning of the translated basic block, and if updated, gets written back to the `baseBlock` before the end of the basic block.[3] This constant shifting of client state from memory to registers and back is a considerable overhead, but is necessary to support shadow registers, which are very important for heavyweight DBA.

- *Virtualisation* is another option, whereby a resource is completely emulated in software. One example is the local descriptor table that is used in conjunction with the x86 segment registers (see Section 2.3.8 for details).

- *Sharing* can be used for resources that both the client and the tool can use without clashing. A lot of process state is handled this way: the process ID and related IDs (e.g. user and group IDs), current working directory, file creation mode mask, etc.

More is said about avoiding resource conflicts in the following sections.

2.3.4 Starting Up

The following ingredients are used at start-up:

- Valgrind's loader (a statically-linked ELF executable, called `valgrind`);

- Valgrind's core (a dynamically-linked ELF executable, called `stage2`);

- the tool plug-in (a shared object);

- the client program (an ELF executable, or a script).

The first step is to get the last three parts loaded into a single process, sharing the same address space; the loader is not present in the final layout.

The loader is loaded and executed normally by the operating system. The loader's single task is to load `stage2` at a high address and execute it. Then `stage2` does a preliminary parse of the command line options to see which tool has been chosen with the `--tool` option. It finds the shared object for the chosen tool and uses `dlopen()` to load the tool plug-in and any dynamic libraries the tool plug-in uses. Then `stage2` loads the client executable (overwriting the no-longer-needed loader in the process). If the client program is a script, the script's interpreter is also loaded. Both the loader and `stage2` judiciously use empty memory mappings along the way to ensure the different pieces end up in the desired locations in the address space. The typical resulting memory layout is as follows.

[3]This means Valgrind tools cannot simulate precise exceptions. In practice, this is rarely a problem.

Step	Done by	Transformation
Disassembly	Core	x86 → UCode
Optimisation	Core	→ UCode
Instrumentation	Tool	→ Instrumented UCode
Register allocation	Core	→ Instrumented, register-allocated UCode
Code generation	Core	→ Instrumented x86

Table 2.1: Five steps to translate a basic block

- `0xc0000000`–`0xffffffff`. The top 1GB is reserved for the kernel on typical x86/Linux systems.

- `0xb0000000`–`0xbfffffff`. The next 256MB are used for `stage2` and the tool plug-in, any libraries (shared objects) and mappings used by them, and their stack. (They do not need a heap, as all dynamic allocations are done using mappings.)

- `0x00000000`–`0xafffffff`. The lower part of the bottom 2.75GB is used for the client itself. The remainder is used for shadow memory required by the tool; the amount needed, if any, depends on the tool. (Shadow memory is discussed further in Section 2.4.5.)

The chosen boundaries can be adjusted to accommodate systems with less typical memory layouts. Small (1MB) *red-zones*—areas that should not be touched—separate each section.

Once everything is in place, "normal" execution begins. The core (i.e. `stage2`) processes any command line arguments intended for it. Tool-specific arguments are possible; the core passes any arguments it does not recognise to the tool plug-in. The core initialises itself, and tells the tool to perform any initialisation it needs. Once all this is complete, the Valgrind tool is in complete control, and everything is in place to begin translating and executing the client from its first instruction.

(In earlier versions of Valgrind, the core used the `LD_PRELOAD` environment variable to graft itself and the tool plug-in to the client process, and then "hijacked" execution. This approach was simpler but had four disadvantages. First, it did not work with statically linked executables. Second, the core did not gain control over the executable until some of the dynamic linker's code had run. Third, the client and Valgrind tool were not well separated in memory, so an erroneous client could wildly overwrite the tool's code or data. Fourth, the core and tool plug-ins could not use any libraries also used by the client, including `glibc`, and so had to use a private implementation of common library functions.)

2.3.5 Making Translations

The compilation of a basic block BB results in the translation $t(BB)$, and involves the following steps, which are summarised in Table 2.1.

1. *Disassembly.* Valgrind represents code with a RISC-like two-address intermediate representation called *UCode* which uses virtual registers. The hand-written disassembler converts each

```
pushl %eax              0: GETL     %EAX, t0
                        1: GETL     %ESP, t2
                        2: SUBL     $0x4, t2
                        3: PUTL     t2, %ESP
                        4: STL      t0, (t2)
                        5: INCEIPo  $1
andl %eax,%ebx          6: GETL     %EBX, t4
                        7: GETL     %EAX, t6
                        8: ANDL     t6, t4  (-wOSZACP)
                        9: PUTL     t4, %EBX
                       10: INCEIPo  $2
addl $0x3, %ecx        11: GETL     %ECX, t10
                       12: ADDL     $0x3, t10  (-wOSZACP)
                       13: PUTL     t10, %ECX
                       14: INCEIPo  $3
jmp-8 0x8048307        15: JMPo     $0x8048307
```

Figure 2.2: Disassembly: x86 code → UCode

x86 instruction independently into one or more UCode instructions; this is straightforward but tedious due to the complexity of the x86 instruction set.

The UCode for each x86 instruction fully updates the affected client state in memory: client state is pulled from the `baseBlock` into virtual registers, operated on, and then pushed back. Most UCode instructions only operate on literal constants and virtual registers. Figure 2.2 gives an example, using AT&T assembler syntax (where the destination operand is shown second). The client state registers held in memory are called `%EAX`, `%EBX`, etc. Virtual registers are named `t0`, `t2`, etc. `GET` and `PUT` move client state values from the `baseBlock` to the virtual registers and back. `ADD`, `SUB` and `AND` work with virtual registers. Each UCode instruction that affects the `%eflags` register is marked as such; the `-wOSZACP` suffix indicates that the `AND` and `ADD` instructions update all six flags. `ST` does a store. `INCEIP` instructions mark where the UCode for each x86 instruction ends; the argument gives the length (in bytes) of the original x86 instruction. The `L` suffix indicates the arguments are word-sized (4 bytes); the `o` suffix indicates the argument size is irrelevant.

UCode's RISC-ness has two big advantages. First, it is much simpler than x86, so there are far fewer cases for tools to handle when adding instrumentation; also, its load/store nature makes it easy for tools to identify memory accesses. Second, it breaks up many complex x86 instructions, exposing implicit intermediate values. For example, the x86 instruction

```
addl 16(%eax,%ebx,4), %ecx
```

performs a complex address computation, creating an implicit intermediate value ($16 + \texttt{\%eax} + \texttt{\%ebx} \times 4$). Many tools need to know about addresses, so making this value explicit makes things easier for these tools. In Figure 2.2 the `push` is broken up, exposing the intermediate value of `%esp` that is used for the store.

```
                                      <prologue>
 0: GETL     %EAX, %eax               movl    0x0(%ebp), %eax
 1: GETL     %ESP, %ebx               movl    0x10(%ebp), %ebx
 2: SUBL     $0x4, %ebx               subl    $0x4, %ebx
 3: PUTL     %ebx, %ESP               movl    %ebx, 0x10(%ebp)
 4: STL      %eax, (%ebx)             movl    %eax, (%ebx)
 5: INCEIPo  $1                       movl    $0x8048300, 0x24(%ebp)
 6: GETL     %EBX, %ecx               movl    0xC(%ebp), %ecx
 7: ANDL     %eax, %ecx               andl    %eax, %ecx
 8: PUTL     %ecx, %EBX               movl    %ecx, 0xC(%ebp)
 9: INCEIPo  $2                       movb    $0x2, 0x24(%ebp)
10: GETL     %ECX, %edx               movl    0x4(%ebp), %edx
11: ADDL     $0x3, %edx (-wOSZACP)    addl    $0x3, %edx
12: PUTL     %edx, %ECX               movl    %edx, 0x4(%ebp)
13: INCEIPo  $3                       movb    $0x5, 0x24(%ebp)
14: JMPo     $0x8048307               pushfl ; popl 32(%ebp)
                                      movl    $0x8048307, %eax
                                      movl    %eax, 0x24(%ebp)
                                      call    VG_(patch_me)
```

Figure 2.3: Code generation: Register-allocated UCode → x86

Also note that the UCode instructions representing arithmetic instructions in the x86 code (instructions #8 and #12) have flag annotations, whereas those introduced by the translation (instruction #2) do not. This is because the subtraction implicitly performed by the `push` does not affect the flags. Finally, although UCode is RISC-like, it does have some x86-specific features.

2. *Optimisation.* Any redundancies introduced by the simplistic disassembler are then removed. In particular, many `GET` and `PUT` instructions are redundant, as are many flag annotations.

   For the example in Figure 2.2, this phase deletes the `GET` at instruction #7 (which is not necessary because of instruction #0), and renames `t6` as `t0` in instruction #8. Also, the flags annotation on instruction #8 can be removed as its effects are clobbered by instruction #12.

3. *Instrumentation.* The tool adds its desired analysis code. It can make as many passes over the code as it likes, including optimisation passes. For clarity, the example adds no analysis code. Instrumentation will be considered in detail in Section 2.4.

4. *Register allocation.* Each virtual register is assigned to one of the six real, freely usable general-purpose registers: `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`. `%ebp` is reserved to point always to the `baseBlock`, and `%esp` is reserved to point to Valgrind's stack. Spill code is generated as needed, along with any temporary register swaps required to account for the fact that `%esi` and `%edi` cannot be used for 1-byte operations. The linear-scan register allocator [110] does a fairly good job; importantly it passes over the basic block only twice, minimising compilation times. The left side of Figure 2.3 shows the results of allocation.

   There is one complication caused by re-allocating registers. A number of x86 instructions use a fixed register(s). For example, some variations of the `mul` instruction multiply `%eax` by another register, and put the 64-bit result in `%edx:%eax`. These are handled with small assembly code helper routines which swap the operands into the fixed registers, perform the operation, then swap the results out as necessary.

5. *Code generation.* Each translation starts with a short prologue which decrements a global counter; every few thousand translations this reaches zero and Valgrind's core checks for certain unusual events like signals (see Section 2.3.6).

   In the body of the translation, each UCode instruction is converted independently into a small number of x86 instructions. Some call assembly code routines. The right side of Figure 2.3 continues the example. The use of `%ebp` to point to the `baseBlock` is obvious from the `GET` and `PUT` instructions.

   Often the instruction generated for a UCode instruction is the same as the one that came in from the original code, but with different registers. For example, an `add` becomes an `ADD` in UCode (plus some other instructions such as `GET`, `PUT`), and then an `add` again in the generated code.

   `INCEIP` updates the client's program counter, `%EIP`; if the update only modifies the least-significant byte of the program counter, a `movb` is used instead of a full-word write to update it, because it is a shorter instruction (arithmetic instructions are not used for this so that the condition codes are not affected). The generated code for the `JMP` stores the condition codes (written by the `ADD` at UCode instruction #11) in the `baseBlock`, and updates `%EIP`. The condition codes are stored with the `pushfl; popl` sequence because x86 does not have a `movfl` instruction for copying the flags register. The translation ends by copying the new `%EIP` into `%eax` and calling `VG_(patch_me)`[4], Valgrind's basic block chainer, which is discussed in Section 2.3.6.

Translations are stored in the *translation table*, a fixed-size, linear-probe hash table. The translation table is large (300,000 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 much simpler to implement and it still does a fairly good job. This is better than the simplistic strategy of clearing the entire translation table used by many DBI frameworks. Translations are also evicted when code in shared objects is unloaded (by `munmap()`).

2.3.6 Executing Translations

Once a translation is made it can be executed. Basic blocks are translated and executed one-by-one, but what happens between them? Control flows from one translation to the next in one of three ways, from fastest to slowest: via *chaining*, the *dispatcher*, or the *scheduler*.

When one basic block, BB_1, directly jumps to another, BB_2, their translations can be chained. Translation $t(BB_1)$ is made to end with a call to `VG_(patch_me)`, a hand-written assembly code

[4] `VG_()` is a C macro; it expands `VG_(patch_me)` into `vgPlain_patch_me`. This macro is used for all names that Valgrind exports, to minimise pollution of the global symbol namespace.

routine. If $t(BB_2)$ has been created, when `VG_(patch_me)` is called it will replace the call to itself with a direct jump to $t(BB_2)$. Otherwise `VG_(patch_me)` invokes the dispatcher to create $t(BB_2)$. The patching will then succeed the next time $t(BB_1)$ is executed.

When chaining is not appropriate, at the translation's end control falls back to the dispatcher, a hand-crafted assembly code loop. At this point all client registers are in the `baseBlock`. The only live registers are `%eax`, which holds the client's program counter, and `%ebp`, which is used only for unusual events, explained shortly, whereby control must fall back into the scheduler. The dispatcher looks for the appropriate translation not in the full translation table, but in a small direct-mapped table. This table acts as a cache for recently-used translations, and it has a hit rate of around 98%. If that look-up succeeds, the translation is executed immediately (using a `call` instruction, so control will eventually return to the dispatcher). This fast case takes only fourteen x86 instructions.

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 table is updated to contain the translation for the basic 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 must periodically check whether a thread switch is due (see Section 2.3.9) or whether there are any outstanding signals to be handled (see Section 2.3.11). To support this, all translations begin with a prologue which checks a counter and causes control to fall out to the scheduler every few thousand translation executions. Control also returns to the scheduler (by the translation setting `%ebp` appropriately before returning control to the dispatcher) when system calls (see Section 2.3.10) and client requests (see Section 2.3.12) occur.

2.3.7 Floating Point, MMX and SSE Instructions

Everything said so far only applies to general-purpose registers and instructions. Floating point (FP) and single-instruction multiple-data (SIMD) MMX/SSE/SSE2 instructions and registers are handled much more crudely.

FP instructions are classified into one of three categories: those which update the FPU state but do not read or write memory, those which also read memory, and those which also write memory. Each one is represented with one of the `FPU`, `FPU_R`, or `FPU_W` UCode instructions. The raw bytes of the original instruction are held within the UCode instruction. The translation for an FP instruction copies the client's complete FPU state from the `baseBlock` into the real CPU, executes the original FP instruction (known from the raw bytes), then copies the updated FPU state back. Effort is made to avoid redundant FPU state copying. FPU instructions that read or write memory or otherwise refer to the integer registers have their addressing modes adjusted to match the real integer registers assigned by register allocation, but are otherwise executed unmodified on the real CPU. A similar approach is used for handling the SIMD instructions and registers. Valgrind does not support AMD's 3dNow! SIMD extensions for x86 [2] at all, as they are not widely used.

This approach keeps the handling of FP and SIMD instructions simple. However, it prevents tools from instrumenting these instructions in very meaningful ways. Although the client state swapping approach for FP instructions is essentially the same as for general-purpose instructions, each FP instruction is encoded in a single UCode instruction that hides the original operand and the FP register values. Tools can only see FP and SIMD load/store addresses and data widths. Correspondingly, Valgrind currently provides no built-in support for tools to shadow the FP and SIMD registers. So far, these restrictions have not been a problem, as all the tools built have not needed to track FP and SIMD state in any detail. In the long-term, some tools may require more on this front.

2.3.8 Segment Registers

One further difficulty comes from the x86 segment registers, which are used by a small number of programs. (See the Intel x86 manual [57] for details of how segment registers are used.) Tools are not interested in "far pointers" (consisting of a 16-bit segment selector and a 32-bit offset); they only want to see flat 32-bit virtual addresses. Therefore, Valgrind's core intercepts all calls to the `modify_ldt()` system call and virtualises the program's local descriptor table (LDT) which contains the segment selectors. Any x86 instructions that use a segment-override prefix have the `USESEG` UCode instruction in their translation, which performs a look-up of the virtual LDT and returns the flat 32-bit address. This whole approach is distasteful, but hard to avoid.

2.3.9 Pthreads

How should threads be handled in this framework? Instrumented code could be run in separate kernel threads, one per child thread in the client program. This sounds simple, but it would be complex and slow. The core's internal data structures would need to be suitably threaded and locked. This might be viable. However, tool plug-ins would also have to lock their own data structures. For many tools, this would mean locking shadow memory (see Section 2.4.5) for every load/store done by the client, which would be hopelessly slow. The reason is that originally-atomic loads and stores can become non-atomic in instrumented code when shadow memory is in use; each load or store translates into a the original load/store plus a load/store of shadow memory. It is unclear how to guarantee, efficiently, that when multiple threads access the same memory location, updates to shadow memory would complete in the same order as the original updates.

To sidestep these problems, Valgrind only supports the POSIX pthreads model, and provides its own binary-compatible replacement for the standard `libpthread` pthread library. This, combined with the core's scheduler, provides a user-space threads package. All application threads are run on a single kernel thread (modulo some complications; see Sections 2.3.10 and 2.3.11) and all thread switching and scheduling is entirely under the core's control. The standard abstractions are supported—mutexes, condition variables, etc.

This scheme works well enough to run almost all threaded programs. It also makes important thread-related events, such as thread creation and termination, and mutex locking/unlocking, visible to the core, and hence indirectly to tools.[5]

Unfortunately, the reimplementation of `libpthread` greatly complicates Valgrind's core. In par-

[5] These are critical for the data race detection tool, Helgrind.

ticular, the standard `libpthread` interacts closely with `glibc` (sometimes through undocumented interfaces) and renders Valgrind's version quite susceptible to breakage when the C library changes. A possible compromise would be to use the system `libpthread`, but have the core still schedule the resulting threads itself; if thread switches only occur between basic blocks, there is no problem with shadow memory accesses. This might be feasible because the core can easily intercept the `clone()` system call with which new kernel threads are started. However, it is unclear whether this scheme will work, and whether it will simplify matters.

2.3.10 System Calls

System calls must be performed normally, untranslated, as Valgrind cannot trace into the kernel. For system calls to work correctly, the tool must make it look like the client is running normally, but not lose control of program execution. When a system call happens, control falls back into the scheduler, which takes the following steps.

1. Save the tool's stack pointer;

2. copy the client state into the real registers, except the program counter;

3. do the system call;

4. copy the client state back out to memory, except the program counter;

5. restore the tool's stack pointer.

Note that by copying the client's stack pointer, the system call is run on the client's stack, as it should be (`%esp` normally points to the tool's stack).

System calls involving partitioned resources such as memory (e.g. `mmap()`, `mprotect()`, `brk()`) and file descriptors (e.g. `open()`, `close()`) are wrapped and checked to ensure they do not cause conflicts with the core and the tool plug-in.

System calls involving processes are also noteworthy. The system call `fork()` creates a child process that is also running under the tool's control, as one would expect, but the two processes are distinct and cannot communicate, unless the tool does something very clever, e.g. with a pipe or file. Note that any output files produced by a tool should include the process ID in the filename to avoid file-writing clashes between forked parents and children, although this will only work if the file is not already open when the fork takes place. Any `vfork()` calls are replaced by `fork()`, as `vfork()`'s semantics are basically impossible to enforce with Valgrind; this is not a problem in practice. The `--trace-children` option (off by default) dictates whether processes started by `execve()` run under Valgrind's control.

Blocking system calls are a further complication. Because the core schedules application threads on a single kernel thread, if an application thread blocks, it is difficult to continue executing the remaining application threads in the meantime, and then continue the blocked thread once the system call unblocks. Valgrind's core originally did this all itself, but it was difficult to get right, and a constant source of subtle bugs. The improved approach involves the use of extra kernel threads.

Every application thread is shadowed with a *proxy kernel thread*. When an application thread goes to execute a blocking system call, the core makes the proxy thread execute it instead, and then reverts control to the application thread once the system call completes. That way, if the system call blocks, the kernel handles it and the remaining application threads can continue normally. Basically, the core makes the kernel do the difficult actions for it.

2.3.11 Signals

Unix signal handling presents special problems 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 should be used to deliver the signal. This cannot be allowed to happen, since the client's original handler code would be executed untranslated. Even worse, if the handler does not return but instead does a `longjmp`, the tool would permanently lose control. Therefore, the core intercepts the `sigaction()` and `sigprocmask()` system calls, which are used to register signal handlers. The core notes the address of the signal handler specified, and instead asks the kernel to deliver that signal to the core's handler(s).

The core catches all synchronous signals, such as `SIGSEGV`. When one arrives the core immediately redirects it specifically at the application thread which raised the signal. This guarantees that the signal handler for that signal (if there is one) is invoked before the thread makes any further progress, as required by POSIX. If the client catches the signal, signal delivery frames are built on the client's stack, and the (translated) handler code is run; if a signal frame is observed to return, the core removes the frame from the client's stack and resumes executing the client wherever it was before the frame was pushed. If the client does not catch the signal, the core will abort with an informative message indicating where in the client the signal was raised.

Things are more complicated for asynchronous signals. Valgrind's core blocks all asynchronous signals. Instead, proxy kernel threads (described in Section 2.3.10) sit around waiting for asynchronous signals. When an asynchronous signal arrives, a suitable proxy kernel thread (as chosen by the kernel) will catch the signal, and move into a state that the scheduler can recognise. Every few thousand basic blocks, the scheduler checks if any proxy kernel threads have caught an asynchronous signal. If so, it passes on the signal to the relevant application thread in the same manner as for synchronous signals, and the proxy kernel thread reverts back to its waiting-for-signals state. If any proxy kernel threads are blocked in system calls, the system call will be interrupted in an appropriate manner. Again, Valgrind's core lets the kernel do the difficult parts for it, such as deciding which thread should receive any signal, accounting for signal masks, and interrupting blocked system calls.

2.3.12 Client Requests

Valgrind's core has a trapdoor mechanism that allows a client program to pass messages and queries, called *client requests*, to the core or a tool plug-in. This is done by inserting into the client a short sequence of instructions that are effectively a no-op (six highly improbable, value-preserving rotations of register `%eax`). When the core spots this sequence of instructions during disassembly, the resulting translation causes control to drop through the scheduler into the core's code for handling client requests. Arguments can be passed to client requests, and they can return a value to the client. Each client request has a code that identifies whether it should be delivered to the core or a particular tool. This allows client requests for different tools to be embedded in a single program; tool-specific client requests that are not for the tool in use are ignored. Sections 2.3.13 and 2.4.5 give examples of the use of client requests.

Client requests can be embedded in any program written in any language in which assembly code can be embedded. A macro makes this easy for C and C++ programs; the client needs to include a header file and be recompiled with the client requests inserted, but it does not need to be linked with any extra libraries. And because the magic sequence is a no-op, a client program can be run normally without any change to its behaviour, except perhaps a marginal slow-down.

2.3.13 Self-modifying Code

As mentioned in Section 2.1.2, self-modifying code is the only kind of code Valgrind (and dynamic binary compilation and caching in general) does not handle.

On some platforms, handling self-modifying code is easy because an explicit "flush" instruction must be used when code is modified, but the x86 does not have this feature. One could write-protect pages containing code that has been compiled, and then flush them if that page is written [69]; however, this approach is not practical if a program puts code on the stack (as GCC does for C code that uses nested functions, for example), and can be slow if code and data segments appear on the same page (as the x86/Linux ELF executable format allows).

Since relatively few programs use self-modifying code, Valgrind only provides indirect support for handling it, via the `VALGRIND_DISCARD_TRANSLATIONS` client request, which tells it to discard any translations of x86 code in a certain address range. This operation is not very fast, and so should not be performed very often. But, with minor source-code modifications (usually not many), it does allow Valgrind tools to work with programs that use self-modifying code.

2.3.14 Memory Management

Valgrind tools must manage their own memory rather than using the system `malloc()` to avoid all re-entrancy issues and also to make sure all memory allocated by the tool ends up in its part of the address space. The core provides an allocator which uses `mmap()` to allocate superblocks, and then hands out smaller blocks from each superblock on request.

2.3.15 Ensuring Correctness

The correctness of the core and tool plug-ins is paramount. The code is littered with assertions, each piece of UCode is sanity-checked several times during the translation process, and the core periodically sanity-checks various critical data structures. These measures have found many bugs during Valgrind's development, and contributed immensely to its robustness.

2.3.16 Termination

Eventually the client program calls the `exit()` system call, indicating that it wishes to quit. The core stops translating, performs its final actions, tells the tool to perform any necessary finalisation, and calls `exit()` itself, passing to the kernel the exit code that the client gave it.

2.3.17 Self-hosting

Valgrind tools cannot run themselves. This is because the memory layout used (described in Section 2.3.4) is fairly rigid, and relies on the core and tool being loaded at a high address, and the client program at a low address. This could be worked around with some effort, but it has not been done. This shortcoming is unfortunate, as it would be great to use Memcheck and other tools on the core and the tool plug-ins themselves.

2.4 How Valgrind Works: Tool Plug-ins

Valgrind tools are implemented as plug-ins to Valgrind's core. This section describes how tool plug-ins work and interact with the core, using Memcheck as an example.

2.4.1 An Example Tool: Memcheck

Memcheck is a Purify-style [52] memory checker designed primarily for checking C and C++ programs. The original version was written by Julian Seward; a number of other people, including myself, have made changes to it since then. It is a good example tool as it is the most complicated Valgrind tool written and it utilises almost all of Valgrind's tool-related features. However, this section is not intended to be a comprehensive description of Memcheck itself.

Memcheck tracks three kinds of metadata.

1. Each memory byte is shadowed by a single addressability (A) bit, which indicates whether the program can safely access that byte. A bits are updated by all operations that allocate or deallocate memory. They are used to detect, at a per-byte granularity, if any memory accesses touch memory they should not.

2. Each register byte and memory byte is shadowed by eight validity (V) bits, which indicate whether each bit has been initialised, according to an approximation of machine semantics. The V bits are used to detect if any of the following are not fully initialised: conditional test inputs; memory blocks read by system calls; or addresses used in memory accesses. The V bits are not checked simply when a value is read, because partially defined words are often copied around without any problem, due to the common practice, used by both programmers and compilers, of padding structures to ensure fields are word-aligned.

   This per-bit validity checking is expensive in space and time, but it can detect the use of single uninitialised bits, and does not report spurious errors on bit-field operations. Because of the expense, the Valgrind distribution includes another memory checker called Addrcheck. It is identical to Memcheck except that it does not track V bits. As a result, it runs substantially faster, but identifies a smaller class of errors.

3. For each heap block allocated, Memcheck records its address and which allocation function (`malloc()/calloc()/realloc()`, `new`, `new[]`) it was allocated with. When a heap block is freed, this information is used to detect if the pointer passed to the deallocation function actually points to a heap block, and also if the correct deallocation function (`free()`, `delete`, `delete[]`, respectively) is used. This metadata is also used when the program terminates, to detect memory leaks; any unfreed heap blocks that have no pointers pointing to them have leaked. (The A bits are reused here, to determine which parts of memory should be scanned for pointers).

As well as this metadata, Memcheck uses two further techniques to find more errors. First, it replaces the client's implementation of `malloc()` and friends with Valgrind's, which allows it to postpone the freeing of blocks; this delays the recycling of heap memory, which improves the chances of catching any accesses to freed heap blocks. This also allows each heap block to be padded with *red-zones*—unused areas at their edges—which are marked as inaccessible and can help with detection of block overruns and underruns. Second, it replaces functions like `memcpy()`, `strcpy()` and `strcat()` with its own versions that detect if the blocks passed overlap; such overlapping violates the ISO C standard and can cause subtle bugs. It does not do this check for `memmove()` for which the blocks may overlap.

Memcheck's requirements place great demands on a DBI framework. Section 2.4.5 explains how Valgrind satisfies those demands.

2.4.2 Execution Spaces

To understand tool plug-ins, one must understand the three *spaces* in which a client program's code executes, and the different levels of control that a tool plug-in has over these spaces.

1. *User-space* covers all code that is translated. The tool sees all such code and can instrument it any way it likes, providing it with (more or less) total control. This includes all the main program code, and almost all of the C library (including the dynamic linker) and other libraries. This is the vast majority of code.

2. *Core-space* covers the small proportion of the program's execution that takes place entirely within Valgrind's core, replacing original code in the client. It includes parts of signal handling, pthread and scheduling operations. It does not include code that the core and tool plug-ins are performing for themselves, however.

   A tool plug-in never sees the code for these operations, and cannot instrument them. However, a tool plug-in can register callbacks in order to be notified when certain interesting events happen in core-space, such as when memory is allocated, a signal is delivered, a thread switch occurs, a pthread mutex is locked, etc.

3. *Kernel-space* covers execution in the operating system kernel. System call internals cannot be directly observed by Valgrind tools, but the core has built-in knowledge about what each system

call does with its arguments, and provides callbacks for events like memory allocation by system calls. Tool plug-ins can also wrap system calls if they want to treat them more specifically. All other kernel activity (e.g. process scheduling) is opaque to tool plug-ins and irrelevant to their execution.

2.4.3 Tool Structure

At a minimum, a tool must define four functions, which the core calls at the appropriate times.

- `void SK_(pre_clo_init)(void)`
  `void SK_(post_clo_init)(void)`[6]

  These two functions are invoked before and after command line processing occurs. Both are needed; `SK_(pre_clo_init)()` so a tool plug-in can declare that it wants to process command line options, and `SK_(post_clo_init)()` so it can do any initialisation that relies on the command line options given. These functions let the tool plug-in do internal initialisation. The tool plug-in also uses them to tell the core certain things. This includes basic things like the tool's name, copyright notice and version number; which *core services* the tool plug-in wants to use, such as error recording or shadow registers; which callbacks should be called when interesting events occur; and any assembly code routines and C functions that the added analysis code will call.

- `UCodeBlock* SK_(instrument)(UCodeBlock* cb, Addr orig_addr)`

  This function is called every time a basic block is translated. It is passed a basic block of UCode, and the address of the x86 basic block that it represents. It must return a basic block of UCode. The function can modify the basic block arbitrarily, but usually the code is only augmented with analysis code.

  Analysis code can be expressed in three ways. First, inline as normal UCode. Second, as calls to assembly code routines defined by the tool plug-in, using UCode's `CALLM` instruction. Third, as calls to C functions defined by the tool plug-in, using UCode's `CCALL`[7] instruction. Tools using `CCALL` need not worry about preserving the necessary registers and flags across the call, or the details of argument and return value passing; this is all handled by the core. This helps efficiency, too, because the core's implementation of C calls is carefully optimised—the code generator preserves only live caller-save registers across calls, and it allows called functions to utilise gcc's `regparms` attribute so that their arguments are passed in registers instead of on the stack. If one has luck with liveness, and arguments are already in the right registers, a C call requires just a single `call` instruction.

  Because UCode is instrumented one basic block at a time, basic block-level instrumentation is easy. Instrumentation at the level of x86 instructions is also possible, thanks to the `INCEIP` instruction which groups together all UCode instructions from a single x86 instruction. However,

[6] `SK_()` is a macro like `VG_()` that adds a prefix to the name.

[7] The inconsistency between the names `CALLM` and `CCALL` is due to historical reasons.

function-level instrumentation is surprisingly difficult. How does one know if a basic block is the first in a function? On x86, there is no tell-tale instruction sequence at a function's start, and one cannot rely on spotting a `call` instruction, because functions in dynamically linked shared objects are called using a `jmp`, as are tail-called functions. Some compilers can produce "debugging grade" code which includes hooks that tell tools when functions are entered and exited. This would make things much simpler, but GCC does not provide this facility. Instead, one can use symbol information. If a basic block shares an address with a function, it must be that function's first basic block. This does not work for programs and libraries that have had their symbols stripped, but there is no obviously better approach. Also, obtaining the function arguments is not straightforward, as the function's calling convention must be known. A more reliable but more invasive alternative is *function replacement*, which is described in Section 2.4.5.

- `void SK_(fini)(Int exitcode)`[8]

  This function lets the tool plug-in do any final processing, such as printing the final results, writing a log file, etc. It is passed the exit code returned by the client.

Memcheck's versions of the four main functions behave in the following way.

- Memcheck's initialisation functions are straightforward. They process any Memcheck-specific command line arguments and do appropriate initialisation.

- Memcheck's instrumentation function performs two passes over the code. The first pass adds the necessary analysis code. Much of the added analysis code involves the *shadow computation* of V bits, which is described more fully in Section 2.4.4. A and/or V bit tests are also added before UCode instructions that require them; if these tests fail they call a function which issues the error message, and then control returns and execution continues. It is important that the checks take place before the operation, so that any error messages are issued before a bad memory access that might crash the client.

  The second pass optimises the added analysis code, by performing constant-propagation and constant-folding of operations on V bits, to remove redundant analysis code.

- Memcheck's finalisation function finishes by printing some basic memory statistics such as the number of bytes allocated and freed, summarising any found errors, and running its leak checker if the `--leak-check=yes` option was specified.

In addition to these four functions, tools must define extra functions if they use certain core services, such as error recording, to help the core with tool-specific tasks.

2.4.4 Shadow Computation

An important class of DBA tools perform *shadow computation*. This is a particularly heavyweight form of DBA, and the term implies that a tool does two things.

[8]The type `Int` is just a synonym for `int` used inside Valgrind.

1. Every value in registers and memory is shadowed with a shadow value, or metavalue.

2. Every value-writing operation is shadowed with a shadow operation that computes and writes the corresponding metavalues. The shadow inputs are the metavalues of the inputs to the original operation. Instructions that do not write values, such as jumps, do not require instrumentation (unless the tool shadows the program counter).

Memcheck's handling of V bits is an example. There are three ways metavalues are propagated in shadow computation.

1. *Copying metavalues.* Value-copying operations must be shadowed with metavalue-copying operations.

   For example, Memcheck instruments UCode instructions that copy values (e.g. `GET`, `PUT`, `LOAD`, `STORE`, `MOV`) so that the corresponding shadow values are copied as well.

2. *New dynamic metavalues.* Value-producing operations—arithmetic operations, logic operations, and address computations—must be shadowed with a shadow operation that computes and writes the appropriate metavalue. The inputs for the shadow operation are the metavalues of the inputs to the original operation. Values produced by system calls—both the return value, and any values written to memory by the system call—must also have their metavalues set appropriately.

   For example, Memcheck's shadow operations combine the V bits of operation inputs in appropriate ways to produce the V bits for the produced values. For some operations, the shadow operation is perfectly accurate. For example, the shadow operation for a shift or rotate is exactly the same shift or rotate—the V bits should follow the motion of the "real" bits. For others where doing a perfect job would be slow, such as addition, Memcheck uses a reasonable approximation; this rarely causes problems in practice. Finally, all values written by system calls have their V bits set as initialised.

3. *New static metavalues.* Instructions using constants effectively introduce new values. Tools using shadow computation need a function to produce a metavalue from a given constant value.

   For example, for an N-bit value Memcheck produces a metavalue with the low N bits set as initialised, and the rest set as uninitialised.

The granularity of the shadowing can vary. Memcheck shadows values at byte-granularity—i.e. each real byte has a metavalue—but some tools use word-granularity.

Shadow computation has similarities with abstract interpretation [35], where the shadow value is the abstract counterpart to the concrete value, except done at run-time. One interesting instance of shadow computation is the *concrete semantics*, where the shadow operations are the same as the real operations: metavalues become the same as values, and one ends up shadowing the real execution with itself! The significance of this is that the program's normal data can be considered as just another kind of metadata, albeit the one actually computed by the program and hardware by default.

It is worth noting that although Memcheck's handling of V bits is an example of shadow computation, its handling of A bits is not. This is because A bits are a property of memory locations, rather than a property of the values within them, and a memory location's A bits do not change as the value within the memory location does. Chapter 6 discusses this matter in more detail.

2.4.5 Crucial Features

Memcheck's analysis, particularly its V bit tracking, is extremely heavyweight. First, the analysis code is pervasive: every instruction that involves values must be instrumented with analysis code that does shadow operations on the shadow V bits, and every load and store must be instrumented with analysis code that checks the relevant A bits; the amount of analysis code added to the translations is actually greater than the amount of original code. Second, the analysis code is interconnected: the analysis code for each instruction is typically two or three instructions, which are connected via intermediate values. Third, it tracks location metadata: a metavalue (the V bits) for every value in the program. The following list describes various crucial instrumentation features that Valgrind provides that make such heavyweight DBA possible.

- *State multiplexing.* UCode's use of virtual registers is a huge help. Memcheck does not have to handle the difficulties of of multiplexing two sets of state (client state and analysis state) onto the one set of machine registers. In particular, Memcheck can manipulate shadow registers and the intermediate V bit computation values without being constrained by the original code—Memcheck does not have to worry about about finding or spilling registers for intermediate analysis values. It also means Memcheck does not need to worry about the analysis code accidentally disturbing the client state; Valgrind's handling of condition codes is a particular help here.

- *Code interleaving.* Both client code and analysis code are expressed in UCode, and the core handles the interleaving of the two code streams.

- *Flexible instrumentation.* Memcheck uses all three forms of analysis code. Analysis code is performed entirely inline where possible; this is vital for efficiency of V bit computations. For more complicated operations, Memcheck calls various assembly code routines and C functions; UCode's `CALLM` and `CCALL` instructions make these calls easy and efficient.

- *UCode's RISC-ness.* Memcheck instruments almost every UCode instruction differently, and there are a substantial number of them. However, this number is far fewer than the number of x86 instructions, which makes things a lot easier. Also, the breaking up of complex instructions (e.g. `push`, which decrements `%esp` and then does a store) into multiple explicit parts exposes implicit intermediate values that Memcheck needs to see, such as addresses. Finally, the load/store nature of UCode makes memory accesses very easy to identify.

- *Shadow register support.* Valgrind provides explicit support for shadow registers for tools that need them. Shadow registers are treated just like normal client registers: they are stored in the `baseBlock`; they can be pulled into virtual registers and operated on with all UCode instructions; and they are preserved/restored when thread switches occur and signal stack frames are pushed/popped.

- *Shadow memory support.* Valgrind provides explicit support for shadow memory, by allocating the necessary address space when the client is loaded (as explained in Section 2.3.4). Tools need to specify a ratio that indicates how much shadow memory they need for each byte of client memory. In Memcheck's case, the ratio is 9:8—every byte of memory is shadowed with one A bit and eight V bits.

  Memcheck instruments each load/store with a corresponding shadow memory load/store, which is done with a call to a C function. Shadow memory is stored in a table, in chunks. Each chunk holds the shadow values for a 64KB region of memory. Chunks are created lazily; if a load/store touches a 64KB region that has not been touched before, Memcheck allocates and initialises the chunk and inserts it in the table. The C function then gets/sets the shadow value.

  Equally important is the fact that Valgrind schedules multi-threaded programs itself; without this feature shadow memory accesses would not be atomic (see Section 2.3.9), and thus not reliable.

In short, all this support is really support for shadow computation. Without these features, Memcheck could probably do all the necessary work itself, but it would make it much larger, more complex and fragile. And the same is true for many heavyweight DBA tools, such as those discussed in the following chapters.

The following list gives important features that Valgrind provides, not directly related to instrumentation, that also make Memcheck's life easy.

- *Memory event callbacks.* Memcheck needs to know about every memory event, particularly allocations and deallocations. This covers all areas of memory: static memory, the heap, the stack, and other mapped segments. The events include stack growth and shrinkage, and the system calls `mmap()`, `brk()`, `mprotect()`, `mremap()`, `munmap()`, `shmat()`, `shmdt()`. It also needs to know about memory reads and writes done within kernel-space so it can handle system calls appropriately. For example, A and V bits are checked before all system calls that read memory (e.g. `write()`), and V bits are updated after all those that write memory (e.g. `read()`).

  The core provides callbacks for each of these events, even though tool plug-ins could, with some effort, detect many for themselves (e.g. stack allocations/deallocations occur in user-space and could be detected at instrumentation-time by identifying stack pointer manipulations in UCode). The core provides these because many tools need to know about memory operations. It also helps efficiency; in particular, stack allocation/deallocation callbacks must be well optimised because `%esp` is changed *very* frequently. Common cases (when `%esp` is changed by 4, 8, 12, 16 or 32 bytes) are optimised with unrolled loops.

- *Function replacement.* Valgrind provides support for tools to override library functions with their own versions, which run in user-space. The replacement functions are put into a shared object (not mentioned in Section 2.3.4) which is linked into the client's memory section, overriding the default versions. The replacements must replicate the replaced function's actions, as well as doing any extra work on behalf of the tool.

Memcheck replaces the standard C and C++ memory management functions (`malloc()` and friends) with its own versions. These replacements run in user-space, but use a client request to transfer control to core-space, so that the core's memory management routines (described in Section 2.3.14) can be used. There are three reasons to use replacements, as Section 2.4.1 explained. First, it provides the necessary hook for the heap memory event callbacks (and so all tools that need to know about heap allocations must replace these functions; the core provides support for this). Second, the heap blocks can be flanked with red-zones to improve the detection of block overruns and underruns. Third, Memcheck's versions of the deallocation functions postpone heap block deallocation for a certain time to delay potential heap block recycling, which improves the likelihood of use-after-free errors being detected.

Memcheck also replaces some C string functions: `strlen()`, `strcpy()`, `memcpy()`, etc. This is because their x86 `glibc` implementations use highly optimised assembly code which is not handled well by Memcheck—they rely on a certain behaviour of the x86 carry bit, which Memcheck's shadow addition operation does not faithfully replicate—causing Memcheck to issue many spurious error messages. Simpler versions do not cause these problems. Also, these replacement versions can check that the source and destination memory blocks passed to these functions do not overlap; if an error is found, a client request is used to transfer control to core-space, so an error message can be issued.

- *Error recording and suppression.* Valgrind provides built-in support for recording errors that a tool detects. A tool can record an error by passing the necessary information about the detected error to the core, which issues an error message containing the given information, plus a detailed stack trace. The core reads any client symbol tables and debug information present in order to make the stack traces as informative as possible; when debug information is present, the exact line of source code that caused the error is pin-pointed. If the tool detects a duplicate error—where the error type and code location is the same as a previously detected error—the core will ignore it, and will not issue any error message. This is important so that the user is not flooded with duplicate error messages. Finally, the core supports error *suppressions*—a user can write suppressions in a file, instructing the core to ignore particular errors detected by the tool. This is extremely important for ignoring errors detected within libraries that are beyond the control of the user.

  A tool plug-in using the core's error reporting features must provide definitions of several callbacks for comparing and printing the tool-specific parts of errors, and reading the tool-specific parts of suppressions from file. This is some work, but much easier than doing error handling from scratch. A checking tool is only as good as its error messages, and Memcheck relies heavily on the core's error handling support.

- *Client Requests.* Tools can define their own client requests. If a request is not recognised by the

Component	C	asm
Core	64,795	1,076
Memcheck	7,737	65
Addrcheck	1,345	0
Cachegrind	1,683	0
Annelid	3,727	0
Redux	5,503	0
Nulgrind	30	0

Table 2.2: Lines of code in core and tool plug-ins

core, it can be passed to the tool which can interpret it as it likes. This opens up the possibility of easily combining static and dynamic analysis—a compiler could systematically embed client requests into a compiled program in order to pass statically-obtained information to a DBA tool. When the program is run normally, the client requests would have no effect, but when the program is run under the DBA tool's control, the tool would have access to more information than it otherwise would. This could make the DBA more accurate or efficient. However, none of the tools described in this dissertation do this in any way, so I will not consider it further.

Memcheck provides client requests that let a program declare an area of memory as readable, writable, or not accessible. This is occasionally useful when a program does something unusual that violates Memcheck's assumptions. For example, OpenSSL deliberately reads uninitialised memory in some circumstances as an (additional) source of randomness. Also, Memcheck can miss some errors in programs with custom memory management that it would otherwise detect in programs that only use `malloc()`, `new`, etc. So it provides a number of client requests that can be embedded in the custom allocators, which tell Memcheck when a block has been allocated or freed; they can even be used with pool-based allocators that can free a number of blocks at once.

2.5 Size of Core and Tool Plug-ins

Having considered which of Valgrind's features made Memcheck easy to write, it is worth quantifying in some way the amount of effort required to write a tool plug-in. Table 2.2 shows the code size of the core and various tool plug-ins. Column 1 gives the component name, columns 2 and 3 give the number of lines of C and assembly code (including comments and blank lines, and all the code is quite well commented) respectively. 10,723 lines of the core's C code are in the C++ name demangler, which was copied with little change from the GNU binutils. Memcheck is clearly the most complicated tool; comparing it with Addrcheck, one sees how much complexity is added by Memcheck's V bit tracking (the only feature it has that Addrcheck does not). Nulgrind is the "null" tool that adds no analysis code. Cachegrind, Annelid and Redux are described in Chapters 3, 4 and 5 respectively.

The core is clearly much bigger than any of the tool plug-ins. So how much easier is it to build a tool with Valgrind, rather than from scratch? As mentioned in Section 2.1.3, originally Valgrind was not a generic tool-building framework, but had Memcheck's functionality hard-wired into it. The last non-generic version (1.0.4), which can be considered a reasonable benchmark for the size of a well-implemented stand-alone DBA tool, had 42,955 lines of C code and 1192 lines of assembly code. (At the time of writing, the size of the core plus Memcheck is 73,673 lines of C and assembly code; this difference comes from the infrastructure added for genericity, plus two years' worth of additional maturity and features.) Memcheck's current code size as a plug-in is thus 5.7 times smaller than the earlier stand-alone version. Even though lines of code is not a particularly good measure of coding effort, the benefit of using Valgrind is clear, especially since DBI frameworks are hard to implement well, and so Valgrind's core contains a lot of difficult code. For the simpler tools, the ratio between plug-in size and core size is even higher, so the benefit of using Valgrind rather than implementing them as stand-alone tools is even greater.

2.6 Performance

This section discusses the performance of various Valgrind tools. All experiments were performed on an 1400 MHz AMD Athlon with 1GB of RAM, running Red Hat Linux 9, kernel version 2.4.20. Valgrind 2.1.2 was used for all tests, except for the tests run under Annelid, which used Valgrind 2.0.0. The test programs are a subset of the SPEC CPU2000 suite [106]. All were tested with the "test" (smallest) inputs. The time measured was the "real" time, as reported by `/usr/bin/time`. Each program was run once normally, and once under each of the Valgrind tools; this is not a very rigorous approach but that does not matter, as the figures here are only intended to give a broad idea of performance.

Table 2.3 shows the time performance of five tools. Column 1 gives the benchmark name, column 2 gives its normal running time in seconds, and columns 3–7 give the slow-down factor for each tool relative to column 2 (smaller is better). The first nine programs are integer programs, the remaining four are floating point programs.

Table 2.4 shows the post-instrumentation code expansion for the five tools. Column 1 gives the benchmark name, column 2 gives the original x86 code size (excluding data) in kilobytes, and columns 3–7 give the code expansion factor for each tool relative to column 2 (smaller is better).

In addition to the space used by instrumented code, the core uses some extra memory, and each tool also introduces its own space overhead; for example, Memcheck uses an extra 9 bits per byte of addressable memory. Note that the slow-down and code expansion factors for each tool do not correlate, because analysis code speed varies greatly. In particular, Cachegrind's analysis code includes many calls to C functions that update the simulated cache state, so its code expansion factor is relatively low, but its slow-down factor is high.

The time and space figures are quite high. This is because Valgrind is designed for building heavyweight DBA tools. The slow-down figure for Nulgrind is a red herring; it would not be difficult to improve it significantly, just by making the emitted code more similar to the original code. But that is not the interesting case—no-one uses Nulgrind, other than the developers for testing purposes. For lightweight DBA tools, one of the frameworks discussed in the next section may be more appropriate. But for heavyweight DBA tools, the time spent in analysis code time far outweighs the time spent in client code, and slow-down factors such as 20, 30 or 50 times are basically unavoidable, and Valgrind

Program	Time (s)	Nulgrind	Memcheck	Addrcheck	Cachegrind	Annelid
bzip2	10.8	2.5	13.8	10.2	40.4	34.0
crafty	3.5	7.9	45.3	27.4	93.3	71.6
gap	1.0	5.6	26.5	19.2	46.0	39.5
gcc	1.5	9.2	35.5	23.7	67.7	50.1
gzip	1.8	4.7	22.7	17.7	64.0	45.8
mcf	0.4	2.6	14.0	7.1	20.5	20.4
parser	3.6	4.2	18.4	13.5	44.9	35.0
twolf	0.2	6.1	30.1	20.5	54.9	46.8
vortex	6.4	8.5	47.9	36.5	85.2	90.3
ammp	19.1	2.2	24.7	23.3	50.0	29.3
art	28.6	5.5	13.0	10.9	21.1	15.0
equake	2.1	5.8	31.1	28.8	54.8	38.1
mesa	2.3	5.6	43.1	35.9	87.4	59.9
median		5.6	26.5	20.5	54.8	38.1
geom. mean		4.9	25.7	19.0	51.1	39.2

Table 2.3: Slow-down factor of five tools

tools actually perform quite well. Without features such as support for pervasive, interconnected analysis code and fast C calls, these tools would probably run much more slowly if they ran at all; for example, if Memcheck could only use C calls, and no inline analysis code, it would run much more slowly than Cachegrind. Besides, judging from extensive user feedback, performance is less important than robustness—most users are not too concerned if a tool runs, say, 30 times slower rather than 20 times slower, but they do notice if the tool falls over on their program. (The performance figures for Cachegrind and Annelid are discussed further in Sections 3.3.5 and 4.4.10.)

2.7 Related Work

This section starts with a brief overview of things that are similar to, but not the same as, DBA and DBI. It then describes a number of DBI frameworks in detail.

2.7.1 Not Quite Dynamic Binary Analysis

Section 1.1.1 briefly covered the differences between static analysis vs. dynamic analysis, and source analysis vs. binary analysis. Putting them together gives four categories of program analysis. *Static source analysis* is a huge field, and accounts for the majority of static analysis.[9] Entire books have been written about it (e.g. [83]), and almost all compilers use some kind of static analysis. *Static binary analysis* is much less common. A good example is [10]. Probably the main use is for decompilation (e.g. [27, 76]). *Dynamic source analysis* can be used for dynamic checking such as bounds-checking (e.g. [60]). The analysis code can be inserted statically by a compiler or a pre-processor, or dynamically by a just-in-time (JIT) compiler or interpreter. Dynamic binary analysis is, of course, the fourth category. This section does not consider related work for DBA at all, because the related work

[9] So much so that "static analysis" usually means "static source analysis".

Program	Size (KB)	Nulgrind	Memcheck	Addrcheck	Cachegrind	Annelid
bzip2	58	5.3	12.3	6.8	9.2	11.3
crafty	176	4.7	11.3	6.1	8.4	11.1
gap	161	5.8	13.1	7.5	9.7	13.1
gcc	553	6.3	13.7	8.0	10.2	13.4
gzip	53	5.5	12.7	7.1	9.4	12.2
mcf	54	5.6	13.0	7.2	9.6	12.2
parser	119	6.1	13.8	7.8	10.2	13.5
twolf	139	5.3	12.3	7.0	9.3	11.7
vortex	252	6.0	13.5	8.2	10.1	15.0
ammp	92	4.9	11.9	7.1	9.5	10.7
art	46	5.5	12.8	7.1	9.5	12.2
equake	68	5.1	12.3	6.9	9.2	11.3
mesa	93	4.9	11.5	6.8	9.0	10.5
median		5.5	12.7	7.1	9.5	12.2
geom. mean		5.4	12.6	7.2	9.5	12.1

Table 2.4: Code expansion factor of five tools

for DBA profilers, checkers and visualisers is given in Sections 3.5, 4.6 and 5.7 respectively.

Section 1.1.1 defined dynamic analysis as not affecting the semantics of the client program. Analysis code added for this purpose is sometimes described as *passive* or *observing* or *monitoring*, as opposed to *active* or *manipulating* or *controlling* code which does affect the semantics.[10] Passive analysis code may well only augment the original code, whereas active analysis code requires the original code to be modified in some way. Active analysis code is used by tools such as sandboxes [112] which prevent a program from doing things like making certain system calls, or allocating too much memory. Some of the DBI frameworks mentioned in Section 2.7.3 (including Valgrind) support active analysis code.

Aspect-oriented programming (AOP) [61] is a programming approach that modularises "cross-cutting concerns", or *aspects*, that are spread across many parts of a program, e.g. error checking and handling, synchronisation, or monitoring and logging. At an implementation level, aspect-oriented programming involves the insertion of code fragments at certain points in a program; this is sometimes called *code weaving*. These added fragments can perform certain kinds of dynamic analysis. DBI works similarly in that it adds extra code to a program, and DBI can be used to do certain kinds of code weaving. The main difference between AOP and DBI is that AOP code weaving is usually done by a compiler at the level of source code, and the instrumentation is typically less pervasive (e.g. analysis code is added to the entry and exit of functions, rather than to every instruction).

2.7.2 Not Quite Dynamic Binary Instrumentation

Static binary instrumentation (mentioned in Section 1.1.4) is an alternative technique for building DBA tools whereby binaries are instrumented before execution. The classic, most widely-used static

[10] I will use the terms "passive" and "active".

binary instrumentation tool was ATOM [105], which could insert calls to arbitrary C functions before and after functions, basic blocks, and individual instructions. It worked on Alphas, and thus is unfortunately now defunct. Etch [95] (also now defunct) and EEL [65] are similar tools, but for the Win32/x86 and SPARC/Solaris platforms respectively. Dixie [44] (also now defunct) was a static binary instrumentation framework that supported multiple architectures by translating binaries to a machine-independent IR and running the instrumented IR on a virtual machine. Static binary instrumentation has been largely overtaken by DBI in recent years due to the reasons given in Section 1.1.4.

Emulators such as Embra [119] provide a detailed emulation of a particular architecture. The line between such emulators and DBI frameworks such as Valgrind is not clear-cut[11]; both kinds of systems can be used for building DBA tools. However, emulators typically model low-level machine details such as memory management unit (MMU) address translations, privileged instructions, and even pipelines, and the DBA tools built with them usually work at this level; they are generally designed to analyse hardware. By comparison, DBI frameworks usually do user-level DBA; they are generally designed to analyse software. The two kinds of DBA sometimes overlap, particularly in the area of cache simulation, but I will not consider emulators further.

DBA tools are sometimes implemented not purely in software. Custom hardware probes and microcode modification are two hardware-dependent techniques that have been used to implement DBA tools; Section 4 of [111] gives examples of several trace-driven memory simulators using these techniques. Such techniques are rarely used these days, as they are not suitable for use with recent machines.

All the discussion so far has been based around sequential programming. There are various tools and tool-building frameworks designed for analysing distributed parallel programs. An example framework is OCM [118], which implements the OMIS [67] tool/monitor interface, and has been used to implement tools such as profilers, interactive debuggers, load balancers, and visualisation tools. OMIS has impressive goals of interoperability and portability, but it is designed for monitoring large parallel systems, and it is based around higher-level message passing, and so is not directly comparable to sequential DBI frameworks.

Dynamic binary translation[12] tools are designed to run programs compiled for one platform (the *guest*) on another platform (the *host*).[13] This can be very difficult, although the difficulty depends very much on the choice of the guest and host architectures. Example systems include `bintrans` [91], which translates x86 to PowerPC and PowerPC to Alpha (all under Linux); QEMU [15] which translates four guest architectures (x86, ARM, PowerPC, SPARC) to six host architectures (x86, PowerPC, Alpha, SPARC32, ARM, S390) under various operating systems; and Transmeta's Crusoe chips [64] which have a permanently running software layer that translates x86 code to the VLIW (very long instruction word) architecture of the underlying processor. Some of the DBI frameworks mentioned in Section 2.7.3 perform dynamic binary translation as well as instrumenting code.

Dynamic binary optimisation tools are designed to speed up programs. They work in almost

[11] Nor is the terminology; the terms "simulation" and "emulation" are used in several different, overlapping ways.

[12] This term is sometimes used to refer to dynamic binary compilation and caching in general, but it is more commonly used in the sense I am using here.

[13] Some people confusingly use the terms "target" and "host", or even "source" and "target" instead.

exactly the same way as DBI frameworks, except they only add lightweight profiling analysis code to determine hot traces. These systems can actually speed up programs, despite the compilation overhead, by focusing on optimisation opportunities that only manifest themselves at run-time. The gains come from optimisations such as: inlining small functions; "straightening out" branches; cache locality improvements; and eliminating redundant loads. Dynamo [9] was the first such system; it was implemented for the PA-RISC/HPUX platform, and achieved speed-ups of 5–20% in some cases; it also reverted to native execution if it judged itself to be performing badly. Dynamo was reimplemented for x86/Win32 [17], and Mojo [24] was a similar system also for x86/Win32; both do not perform as well as Dynamo, and usually slow programs down slightly. Some of the DBI frameworks mentioned in Section 2.7.3 perform dynamic binary optimisation as well as instrumenting code.

2.7.3 Dynamic Binary Instrumentation Frameworks

This section describes eleven DBI frameworks, including Valgrind, in no particular order. Valgrind is described in the same way as the other ten to ease direct comparisons. The details were gleaned from a mixture of papers, websites, source code, manuals, and personal communication with the authors. I tried to contact all framework authors so that they could check draft versions of the descriptions for inaccuracies; those descriptions that have been so checked are marked with a † symbol. Both checked and unchecked descriptions may contain incorrect information, but the checked ones should be much more reliable.

Table 2.5 gives a high-level comparison of the eleven frameworks. Column 1 gives the name of each framework. Column 2 gives the year each framework was first used or released, if known, or the year of the earliest publication about the framework. Column 3 indicates the host platform(s) that each framework runs on (not the guest platform(s), which in some cases can be different). Columns 4 and 5 give characteristics of the execution and instrumentation mechanisms. Column 6 indicates roughly how robust each framework is. Column 7 indicates the reported slow-down factor for the framework when it runs programs without adding any analysis code; this number can be misleading as it gives no idea how efficient added analysis code can be, but no better comparison is possible. It gives at most a rough idea of speed, but at least shows which frameworks cause drastic slow-downs. Column 8 indicates the framework's availability. The meanings of the used abbreviations are given in the table's legend; note that the '*' annotation applies only to the closest item, rather than every item in the column.

In what follows, each framework is described in the following way. The introduction gives a basic description of the framework, including its main aim. It also mentions if the framework does dynamic binary optimisation, dynamic binary translation, whether it supports active analysis code, and when the framework first appeared or was first described in a publication. The following entries are then given.

Platform. Describes the host platform(s) the framework runs on. Also, if it supports dynamic binary translation, the guest platform(s) of the binaries it can execute.

Execution. Gives basics of the execution mechanism, describing whether it uses dynamic binary compilation and caching, mixed compiled/interpreted execution, or if the original code is run and

Framework	Date	Host	Exec	Instr	Robust	Slower	Avail
Shade†	1993	SPARC/Solaris	C,T	c,b	≥med	3–6×	B
DynamoRIO†	2002	x86/{Win32,Linux}	C,O	c,i,m	high*	1–2×	B
DynInst†	1996	Many*	I*	c,f,m*	high	1×*	S*
Pin†	2003	IA64/Linux, others*	C	c,i*,f,m	high	1.4–2.4×	S*,B
DIOTA†	2002	x86/Linux	C,O*	c*,f	high	1.2–23×*	S
Walkabout†	2002	SPARC/Solaris*	M,O*,T	c,f	≥med	0.7–175×	S
Aprobe	?	x86/{Win32,Unix*}	I?*	c,f	?	?*	C
Vulcan†	1999	x86/Win32	C*,O?*	c,i*	high	?	no*
Strata	2001	Several*	C	?*	≥med	1–3×	S?*
DELI	2002	Various?*	C,O,T*	i,m	high	?	no?*
Valgrind†	2002	x86/Linux	C	c,i,f,m*	high	2–10×	S

Execution:
- C Dynamic binary compilation and caching
- I Normal execution with inline trampolines
- M Mixed compiled/interpreted execution
- O Dynamic binary optimisation
- T Dynamic binary translation

Instrumentation:
- c C function/external routine calls possible
- b Has a built-in instrumentation mode
- i Inline analysis code possible
- f Function entry/exit can be instrumented
- m Active analysis code supported

Robustness:
- ≥med can run at least SPEC benchmarks
- high can run large applications

Availability:
- S Source code available free-of-charge
- B Binary code available free-of-charge
- C Commercial

Annotations for all columns:
- † information checked by author(s)
- * see the text below for the full story
- ? unknown or uncertain

Table 2.5: Eleven dynamic binary instrumentation frameworks

modified in-place with jumps to out-of-line analysis code. It also describes the IR if the framework uses one. Except where mentioned, all frameworks only run user-level code.

Instrumentation. Describes what kinds of instrumentation the framework supports, including support for inline instrumentation, calls to C functions/external routines, instrumentation of function entry/exit, and whether active analysis code is possible.

Robustness. Indicates whether the framework can run large programs. This is a very rough measure of robustness; it is meant to indicate which frameworks have been developed past the point of proof-of-concept into practical, mature tools. This is important, as DBI is difficult to do well, and there are lots of details to get right (as Section 2.3 demonstrated).

Performance. Gives the reported slow-down factor for the "no instrumentation" case, and any other relevant performance figures, such as code-expansion factors, or slow-down when instrumentation is included. Performance methodologies (e.g. number of runs, what exactly was measured, what means are used) are not described, because many of the papers were vague about this; SPEC benchmarks were used for most of the measurements. This entry should be taken only as a basic indication of performance, as performance will of course vary depending on the client program, host hardware configuration, and many other factors.

Tools Built. Lists any tools built with the framework.

Availability. Describes whether the framework can be obtained, and if so, where from and under what licence.

Evaluation. Presents particular pros and cons of the framework, and circumstances under which it might be appropriately used. These entries are as much opinion as fact, and reflect my perceptions of the frameworks, many of which I have only read about. It also mostly ignores the question of which platforms the tool supports. Nonetheless, it provides a useful summary for each tool.

Shade† Shade [31, 30] was an early and seminal DBI framework designed for building simple tracing and profiling tools. It pioneered many now-standard features: it was the first framework to use dynamic binary compilation and caching, the first to give control over analysis code, and the first to efficiently interleave analysis code with client code. It also supports dynamic binary translation. It was first publicly released in 1991. The two papers cited are remarkable for their clarity and level of detail.

Platform. Shade can run SPARC.V8/Solaris and SPARC.V9/Solaris and MIPS/UMIPS-V binaries on SPARC.V8/Solaris. There was also a version of Shade that could run x86/Solaris binaries on SPARC/Solaris, although it was missing some x86 features such as segmentation and 80-bit floating point numbers.

Execution. Shade uses dynamic binary compilation and caching. Like Valgrind, Shade stores the client state in memory and pulls it into the machine registers to operate on when necessary.

Instrumentation. Shade can insert calls to C functions before and after any instruction. It also has built-in support for simple analysis code that collects various pieces of client state, such as register contents or instruction opcodes, into a trace buffer which can be processed at regular intervals. Client instructions can be modified in simple ways, e.g. by changing their opcodes.

Robustness. Shade can run large applications, including other dynamic translation tools, and itself.

Performance. Without any instrumentation, Shade runs SPARC.V8 programs on SPARC.V8 2.8–6.1 times slower than native, with a code expansion factor of 4.1–4.7; these figures are quite similar to Nulgrind (the "null" Valgrind tool) which makes sense since Shade and Valgrind both store client state in memory by default.

Tools Built. Shade has been used to build many profiling/tracing tools, such as instruction counters, cache simulators, branch analysers, etc.

Availability. Shade is available [107] free-of-charge with a non-transferable binary-only licence.

Evaluation. Shade introduced many important DBI (and dynamic binary translation) features and has been highly influential. However, it has now been eclipsed—the lightweight DBA it supports can be done with better performance and greater flexibility by frameworks such as DynamoRIO.

DynamoRIO† DynamoRIO [19], derived from Dynamo, is a dynamic binary optimisation and instrumentation framework ("RIO" is short for "run-time introspection and optimisation"). It also supports active analysis code. It was first released in 2001, but instrumentation support was not added until 2002.

Platform. x86/Win32 and x86/Linux.

Execution. DynamoRIO uses dynamic binary compilation and caching. It works almost directly on the original x86 code—only control-flow instructions are modified; if no instrumentation is added and no modifications made, the produced code is almost identical to the original code (except for the hot traces that get optimised).

Instrumentation. DynamoRIO provides a detailed and well-documented API (application programming interface) for modifying code and adding analysis code. The representation of the code has an adaptive level of detail to minimise decoding/encoding times. Analysis code can be added inline, and support for calling C functions is also provided. There is also support for spilling registers to thread-local storage and for saving/restoring the `%eflags` condition code register, which makes it easier to avoid conflicts between client code and analysis code; however, unlike Valgrind, the code using the API must decide itself when these operations are necessary. The nature of the API ties it tightly to the x86 architecture.

Robustness. The Win32 implementation is highly robust and can run programs such as Microsoft Office and Mozilla; the Linux implementation is apparently less robust, but can run at least the SPEC CPU2000 benchmarks.

Performance. DynamoRIO has excellent performance; without any analysis code added or special optimisations performed, the typical slow-down factor is around 1.0–1.5. With some minor peephole optimisations it runs some programs faster than normal.

Tools Built. DynamoRIO has been used to build various dynamic optimisers, and also lightweight DBA tools; one example is a tool that protects against some security attacks, by checking that all jumps in a program look safe [62].

Availability. DynamoRIO is available [18] free-of-charge under a non-transferable, binary-only licence.

Evaluation. DynamoRIO's speed and exact representation of the original client code make it ideal for building lightweight DBA tools, tools using active analysis code such as sandboxes, and for experimenting with dynamic binary optimisation. To me, it looks to be the best of the frameworks for doing lightweight DBI. Like several of the tools mentioned in this section, although it is quite comparable to Valgrind, it is not suitable for more heavyweight analyses, as it lacks the necessary support for things like shadow registers.

DynInst† DynInst [20] is a DBI framework with one significant difference—it uses a separate "mutator" process which can insert and remove analysis code snippets into an already running program. It is aimed particularly at profiling long-running scientific programs, for which the ability to insert and remove analysis code at run-time is very useful, but can also be used for general DBA. DynInst was first released in 1996, although the instrumentation method dates from 1994.

Platform. DynInst is available on many major platforms: x86/Linux, x86/Win32, IA64/Linux, SPARC/Solaris, PowerPC/AIX, Alpha/Tru64 and MIPS/IRIX. This support of many platforms is due to a large coding effort, rather than any particular features it has.

Execution. The client program executes much as normal; the mutator instruments the client by replacing each chosen instruction in-place with a jump to an analysis code snippet (which contains a copy of the replaced instruction). If the chosen instruction is smaller than the replacement jump, nearby instructions are relocated to make space if possible, otherwise the entire function is rewritten in memory. External routines to be called by the snippets can be put in a dynamically linked library which the mutator can graft onto the client.

Instrumentation. The analysis code snippets are specified in an architecture-independent way as an abstract syntax tree built up from variables, expressions, if-then-else statements, etc. The API can be used manually, although it is cumbersome to do so; it was intended that analysis code specifications be machine-generated. No inline instrumentation is possible, as code snippets are inserted using jumps [54]. Function entry/exit points can be instrumented. Functions can be replaced with other functions.

Robustness. DynInst is highly robust, and can run programs such as Microsoft Office and MySQL.

Performance. No performance figures are given in [20] but the slow-down is presumably nil or almost nil without instrumentation, and then increases as more instrumentation is added. The trampolining technique means that calling analysis code is not very efficient. On the other hand, analysis code can be removed if it has served its purpose, which can speed up some tools.

Tools Built. DynInst has been used to build many profiling tools, coverage tools, debugging tools (by providing efficient breakpoints), "computational steering" tools, etc. It forms the core of the Paradyn parallel profiling tool [75], and DPCL [37] is a layer on top of it with some features to support large parallel programs.

Availability. DynInst's source code is available [40] under a non-transferable licence, free-of-charge for research purposes, and for a fee for commercial use.

Evaluation. DynInst's main strength is that it can add/remove instrumentation to/from an already-running program, which is very useful in certain circumstances, such as when profiling long-running scientific applications. DynInst is also very applicable when very little analysis code needs to be added (due to the overhead of jumps to analysis code, if more analysis code is necessary, DynamoRIO or Valgrind might be more appropriate). DynInst is also noteworthy because it works on by far the most platforms of any of the mentioned frameworks.

Pin† Pin [32] is a DBI framework designed to provide similar instrumentation functionality to ATOM. It also supports active analysis code. The cited reference is a tutorial; it and the user manual give some information about Pin but not a huge amount, so some of what follows is incomplete. The Pin website dates from 2003.

Platform. Pin has been released for IA64/Linux; x86, x86-64 and ARM versions (all for Linux) are in development.

Execution. Pin uses dynamic binary compilation and caching. It uses the Ispike post-link optimiser [68] to improve performance.

Instrumentation. Pin can add calls to analysis routines (written in C or assembly code) before and after any instruction, and on the taken edge of branches. Simple analysis routines (i.e. containing no control flow) are inlined by Pin. Function entry/exit points can also be instrumented. Active analysis code can be used; Pin tools can modify register and memory values at run-time, modify instructions, and replace procedure calls. Symbol information can be accessed easily. An API is provided that allows architecture-independent tools to be written; the API is similar to Valgrind's interface for instrumenting UCode. Virtual registers can be used to pass information between analysis routines.

Robustness. Pin is used to analyse large multi-threaded and multi-process commercial applications.

Performance. According to the author, Pin's slow-down factor with no instrumentation for IA64 is 1.4–2.4, and for x86 it is around 2.0.

Tools Built. The Pin distribution includes several basic tracing and profiling tools: a cache simulator, a branch prediction simulator, a load latency profiler, an edge counter, a stride profiler, etc. The cited tutorial mentions a stack access checker.

Availability. Pin is available [33] free-of-charge. The instrumentation engine is available only as a binary, but the instrumentation libraries and example tools are available under a BSD-style licence.

Evaluation. Pin is a nice framework for doing ATOM-style instrumentation, although it might not be suitable for very heavyweight DBA tools such as Memcheck.

DIOTA† DIOTA [70, 69] is another DBI framework, with some support for dynamic binary optimisation. The earlier cited paper is from 2002.

Platform. x86/Linux.

Execution. DIOTA uses dynamic binary compilation and caching. It supports self-modifying code by using write-protection of pages that have been translated.

Instrumentation. DIOTA provides basic instrumentation facilities. There are built-in tracing modes for calling a routine on each memory access, and at the end of each basic block. It can also intercept function calls when symbol table information is present. It also has an optimisation mode that tries to speed up the client program. Users can write "backends" which specify a combination of the built-in instrumentation modes to use. Inline instrumentation does not seem possible.

Robustness. DIOTA is robust, and can run web browsers such as Mozilla and Konqueror.

Performance. DIOTA's slow-down factor when no instrumentation is present is 1.2–5 [70], and 2.4–22.6 when the self-modifying code support is used [69].

Tools Built. DIOTA has been used to build a range of tools, including a precise memory leak detector [71], basic data and code tracers, a simple memory sanity-checker, a record/replay module for pthread operations, a data race detector and a coverage tool.

Availability. DIOTA's source code is available [96] free-of-charge under the GNU GPL.

Evaluation. DIOTA runs on x86/Linux, like Valgrind, and the performance and robustness of the two frameworks are similar. However, DIOTA's instrumentation support is much more limited, so Valgrind seems to be better for building DBA tools.

Walkabout† Walkabout [29] is a framework for experimenting with dynamic binary translation, designed from the ground up to be highly retargetable and machine-independent. The cited paper is from 2002.

Platform. Walkabout runs x86/Linux and SPARC.V8/Solaris binaries on SPARC.V8/Solaris, SPARC.V9/Solaris and x86/Linux, although the PathFinder component only runs SPARC.V8/Solaris code on SPARC.V9/Solaris.

Execution. Walkabout has two main parts. The first part is an instrumented interpreter generator. It takes three files as input: a machine syntax description, a machine semantics description, and a file describing the analysis code in a format called INSTR. It generates an interpreter (in C or Java) for SPARC.V8 or x86. The second part is called the PathFinder; it uses dynamic compilation and caching and dynamic binary optimisation techniques to speed up the generated interpreters.

Instrumentation. The analysis code is specified in INSTR as C/C++ code, and can be attached to individual instructions; the specification is architecture-specific, so tools would have to be re-written for each architecture. Function entry/exit instrumentation can be done by instrumenting the appropriate instructions, which is much easier on SPARC than on x86. It seems most suited for simple DBA tools such as basic block counters.

Robustness. Walkabout is at least robust enough to run SPEC CPU95 and CPU2000 benchmarks. The project has finished and no more work is planned.

Performance. The generated instrumented interpreters can be run as-is, although their performance is poor, as would be expected for interpreters—interpreting SPARC.V8 code on SPARC.V8 results in a slow-down factor of 139–261. However, the PathFinder (which runs on SPARC.V9) can be used in conjunction with an interpreter to perform dynamic binary compilation and optimisation of hot traces. PathFinder's performance varies wildly; the slow-down factor with it was 0.6–175.

Tools Built. Walkabout has been used to build basic proof-of-concept tracing tools that do basic block tracing, memory access tracing, instruction tracing, etc.

Availability. Walkabout's source code is available [28] free-of-charge under a BSD-style licence.

Evaluation. Walkabout incorporates some very interesting ideas, however, its greatly varying performance and limited instrumentation support mean that other frameworks are more suitable for building DBA tools.

Aprobe OCSystems' Aprobe [84] is a commercial DBI framework. The cited white paper, from OCSystems' website, provides little detail about how Aprobe works. Therefore the following description is largely guesswork.

Platform. Aprobe is available for x86/Win32, and also for Solaris, AIX and Linux (also presumably for x86).

Execution. There are few details available about how Aprobe works, although it sounds like it works similarly to DynInst, using some kind of in-place instrumentation.

Instrumentation. Lightweight instrumentation "probes" can be written in a C-like language. Probes can be attached to function entry/exit (and can gather function arguments) and specified lines of source code. Each probe must be pre-compiled to work with a particular program binary; when Aprobe loads the client the probes are inserted. It seems instrumentation cannot be inserted inline. Aprobe apparently provides ring buffers for collecting trace information.

Robustness. Being a commercial framework one would hope its robustness is high although there is no evidence provided for this.

Performance. No data is available for performance. Presumably if no probes are added there is little or no slow-down.

Tools Built. Aprobe comes with various pre-defined probes, for basic profiling and tracing purposes, such as time profiling, heap profiling and test coverage.

Availability. Aprobe is sold under a commercial licence.

Evaluation. Aprobe seems to be most similar to DynInst, however there seem to be no compelling reasons to use Aprobe in favour of DynInst. In particular, the need to recompile probes for each program binary removes one of the key advantages of DBI.

Vulcan† Vulcan [104] is a framework that supports both static and dynamic binary instrumentation, and is designed to handle heterogeneous programs. It may also support dynamic binary optimisation. The Vulcan project is run by Microsoft. The cited paper is from 2001, but it superseded a technical report from 1999.

Platform. Vulcan runs on x86/Win32. It handles all supported Win32 binaries, namely x86, x86-64, IA64 and MSIL (Microsoft Intermediate Language).

Execution. For static instrumentation, it seems that static binary rewriting is used, and the program is then executed normally. For dynamic instrumentation, it seems Vulcan replaces code in-place where possible, or jumps to out-of-line code if the replacement code is larger than the original code.

Instrumentation. An architecture-independent intermediate code representation is used. Instrumentation takes place at this level, although the original code can be referenced for architecture-specific instrumentation. Analysis code can be generated from basic building blocks such as memory addresses, branches and register contents; calls to C functions are also possible. Inline analysis code is possible. For static instrumentation the binary file is rewritten. For dynamic instrumentation the analysis code is injected into the running process; the injection is done by another process, which can even be running on a remote machine.

Robustness. Vulcan is robust, and can run large programs such as Microsoft Office applications. It is also being used to build post-link optimisers.

Performance. The cited paper gives no performance figures.

Tools Built. Vulcan has been used to build a variety of tools, including basic profiling tools, binary matching and differencing tools, code coverage tools, and fault injection tools. It has also been used to implement some dynamic optimisations, although the details of these are not clear. It has apparently been used internally at Microsoft to improve the performance of various Microsoft products.

Availability. Vulcan is not publicly available, but the website [74] of the developing research group makes it clear that Vulcan is under active development.

Evaluation. Vulcan's outstanding features are its support for both static and dynamic binary instrumentation, and its support for heterogeneous programs. However, because it is not publicly available it cannot be used by anyone outside Microsoft, which is a shame as it sounds quite powerful.

Strata Strata [101, 100, 99] is a DBI framework that is designed to work on multiple platforms (but it does not perform dynamic binary translation). The earliest cited paper is from 2001.

Platform. The architecture-dependent and architecture-independent parts are carefully separated; Strata has been ported to SPARC/Solaris, MIPS/IRIX and x86 (presumably Linux).

Execution. Strata uses standard dynamic binary compilation and caching. Like DynamoRIO, Strata modifies the client's code as little as possible before instrumentation.

Instrumentation. The cited papers contain little detail about how instrumentation is added; it seems that inline code is not possible, and that tools would have to be re-written for each platform, although this is not clear. The paper [99] states "We are currently developing a flexible interface and specification language that will allow a user to define their own architectural simulators without having any knowledge of how... Strata works."

Robustness. Strata can run at least the SPEC CPU2000 benchmarks.

Performance. Performance is good; uninstrumented code runs 1.0–1.8 times slower than normal on SPARC and x86, and 1.1–3.0 times slower on MIPS.

Tools Built. Strata has been used to build several DBA tools, including a system call monitor, a malicious code injection detector (which prevents the execution of code from the stack or heap), and a cache simulator.

Availability. The paper [99] states "The source code is available for Strata" however I was not able to find it on the web, and my efforts to contact the authors were unsuccessful.

Evaluation. Strata seems quite similar to DynamoRIO, but works on more platforms. If it were publicly available, this property and its supposed open-source licence could make it a DynamoRIO-beater.

DELI DELI [38], also derived from Dynamo, is intended to provide an infrastructure for building tools that do dynamic code transformation. It provides compilation, caching and optimisation services for dynamic code, and also a hardware abstraction layer. As a result, it can be used both for DBI and also as a component in dynamic binary translators, although it does not have the binary translation capabilities built-in. It also supports active analysis code. The cited paper is from 2002.

Platform. The cited paper does not give much information about the implementation. A related presentation [39] mentions that it was being ported to x86, ARM, ST200 and SuperH architectures, for the PocketPC and Linux operating systems.

Execution. DELI uses dynamic binary compilation and caching, augmented with the hardware abstraction layer. It can run above or below the operating system. It also features a "transparent" mode in which it can run a normal program under its control, like Dynamo and Valgrind do.

Instrumentation. DELI uses an intermediate representation that is basically at the assembly code-level, plus it has virtual registers. It provides an API for tools to modify the code and add analysis code, so inline code is possible. There does not seem to be built-in support for calling C functions from instrumented code.

Robustness. The cited paper states that DELI is robust enough to run WinCE.

Performance. The cited paper has little in the way of performance figures.

Tools Built. The DELI paper suggests using DELI in a number of ways: for dynamically patching code to remove instructions that a processor might not support; for decompressing or decrypting code on the fly; for sandboxing; and for binary translation. None of these seem to have been implemented, though, except for a binary translator for running Hitachi SH3 code on an Lx/ST210 VLIW embedded processor, which allowed mixed emulated/native execution and seemed to work well.

Availability. DELI does not seem to be publicly available. It is unclear whether DELI is still being worked on actively.

Evaluation. DELI seems to provide good infrastructure for building emulators. For building lightweight DBA tools, it seems roughly comparable to DynamoRIO, although there is less information available and it does not seem to be publicly available.

Valgrind† Valgrind is a DBI framework that is particularly suited for building heavyweight DBA tools. It also supports active analysis code. Valgrind was first released in 2002.

Platform. x86/Linux.

Execution. Valgrind uses dynamic binary compilation and caching.

Instrumentation. Original x86 code is converted to a RISC-like IR called UCode. Instrumentation occurs at the UCode-level. Added analysis code can be inline, specified as UCode; calls to external assembly code routines and C functions are also possible. Valgrind can also intercept entry to chosen functions. Symbol information can be accessed easily. UCode's form makes it easy to write pervasive and interconnected analysis code; Valgrind also provides support for location metadata, the shadowing of every register and memory value with a metavalue. Active analysis code is possible, although the use of UCode means that a tool does not have total control over the end code; thus Valgrind could be used for sandboxing, but would not suitable for low-level peephole optimisations.

Robustness. Extremely high; Valgrind can run programs with millions of lines of code.

Performance. Without instrumentation, its slow-down factor is about 2–10, typically 5, and the code expansion is about 4.5–6.0. This slow-down is somewhat misleading, however, as Valgrind provides the infrastructure necessary to support heavyweight DBA tools in which the analysis code time greatly outweighs the client code's time.

Tools Built. Valgrind has been used to build several heavyweight DBA tools, including three memory checkers, a cache profiler, a dynamic data flow tracer, a heap profiler and a data race detector. Several of these tools, particularly Memcheck, one of the memory checkers, are in wide use in the open source world, with thousands of users.

Valgrind was modified and used for the RISE project [13] which protects against binary code injection attacks by using randomised instruction set emulation, an instruction set obfuscation technique. Timothy Harris used Valgrind for a prototype implementation of a "virtualized debugger" [50] designed for debugging threaded and distributed programs that are difficult to debug using traditional techniques. A modified version of Valgrind was used to create Kvasir, a front-end for the Daikon invariant detection system [42]; Stephen McCamant said [73]: "Leveraging Valgrind's infrastructure has been amazingly helpful in getting this project together: after one undergraduate-semester's worth of effort, we already have a tool that works better than our old, source-based instrumenter does after years of effort." Christopher January created a Valgrind tool called Logrind [59] that captures complete program traces in a relational database, which can be queried for interesting events. He also augmented Valgrind so the database can be used for debugging within GDB; since the whole program trace is in the database, this allows backward debugging.

Availability. Valgrind's source is available [102] free-of-charge under the GNU GPL.

Evaluation. Valgrind's main strength is its suitability for building heavyweight DBA tools. Valgrind can be used for building lightweight DBA tools, although their performance will not be as good as if they were built with a faster DBI framework such as DynamoRIO. Valgrind is also very robust.

2.8 Conclusion

This chapter has described Valgrind, a DBI framework for the x86/Linux platform. It began with an example of Valgrind's use. It then described in great detail how Valgrind's core works, including many gory details. It then described how tool plug-ins work and interact with the core, using Memcheck as an example. It finally considered code sizes, performance, and related work.

Valgrind demonstrates how a DBI framework can support heavyweight DBA. Although various DBI frameworks share some features with Valgrind, it is Valgrind's combination of features—support for pervasive, interconnected instrumentation, and support for location metadata—that is unique. This makes Valgrind suitable for building heavyweight DBA tools, such as the three described in the following chapters, that are of a substantially different nature to those built with other DBI frameworks.

CHAPTER 3

A Profiling Tool

This chapter describes a useful profiling tool, built with Valgrind, which uses moderately heavy-weight dynamic binary analysis.

3.1 Introduction

This chapter describes Cachegrind, a cache profiling tool, which was built with Valgrind.

3.1.1 Profiling Tools

The first major group of DBA tools are profilers. Profilers collect information about a program that a programmer (or another program) can use to rewrite the program so that it runs faster, and/or uses less memory. Countless numbers of profilers have been written. Many attribute time to sections of code, typically procedures. Others give other statistics about how the hardware was used, such as cache miss ratios, or the number of times each line is executed.

Profilers are becoming more important as computers become more complex. A modern CPU, such as a Pentium 4, features deeply pipelined, out-of-order, superscalar execution, with multiple functional units; a core that breaks x86 instructions into lower-level, RISC-like micro-ops; a complex memory hierarchy with multiple caches plus other components such as translation look-aside buffers; and sophisticated branch prediction [58]. Programmers have little chance of understanding how their programs interact with such complex hardware. Tools are needed to help them with this. Tools dealing with the memory hierarchy and cache utilisation are particularly important, since the memory system is often a critical performance bottleneck.

Architecture	AMD K7, model 4
Clock speed	1400 MHz
I1 cache	64KB, 64B lines, 2-way
D1 cache	64KB, 64B lines, 2-way, write-allocate, write-back, 2 64-bit ports, LRU replacement
L2 unified cache	256KB, 64B lines, 8-way, on-die, exclusive (contains only victim blocks)
System bus	Pair of unidirectional 13-bit address and control channels; bidirectional, 64-bit, 200 MHz data bus
Write buffer	4-entry, 64-byte
D1 replace time	12 cycles
L2 replace time	206 cycles

Table 3.1: Model 4 Athlon characteristics

```
// fast.c
int main(void)
{
   int h, i, j, a[1024][1024];

   for (h = 0; h < 10; h++)
      for (i = 0; i < 1024; i++)
         for (j = 0; j < 1024; j++)
            a[i][j] = 0;      // !!

   return 0;
}
```

```
// slow.c
int main(void)
{
   int h, i, j, a[1024][1024];

   for (h = 0; h < 10; h++)
      for (i = 0; i < 1024; i++)
         for (j = 0; j < 1024; j++)
            a[j][i] = 0;      // !!

   return 0;
}
```

Figure 3.1: Fast and slow array traversal: `fast.c` and `slow.c`

3.1.2 Cache Effects

Cache misses are expensive. A main memory access on a modern personal computer can take hundreds of CPU cycles. For example, Table 3.1 gives the characteristics of an AMD model 4 Athlon that was new in 2001. The information in the first part of the table was gathered from AMD documentation [3] and the results of the `cpuid` instruction. The cache replace times in the second part of the table were found using Calibrator v0.9e [72], a micro-benchmark which performs multiple dependent array accesses with various stride lengths to estimate worst-case D1 and L2 cache latencies.

The high cost of cache misses means that the cache utilisation of a program can have a huge effect on its overall speed. For example, consider the two simple programs in Figure 3.1, identical except for the line assigning zero to the array `a[][]`. Both programs were timed on the mentioned AMD Athlon (running Red Hat Linux 9.0, kernel version 2.4.20).

The first program does a row-major traversal, and executes in 0.19s ("user time", as measured by `/usr/bin/time`). The second program does a column-major traversal, and executes in 2.38s, i.e. 12.5 times slower. In both programs, the 4MB array is too big to fit into the cache. In the first version, the traversal is done along cache lines, so only every 16th array accesses causes a (D1 and

L2) cache miss. In the second, the traversal is done across cache lines, so every array access causes a cache miss. A 16-fold increase in L2 cache misses causes a 12.5-fold increase in execution time.

3.1.3 Cache Profiling

Cache utilisation can clearly have a huge effect on program speed. While techniques for optimising the cache utilisation of array-based programs are mature and well-known, the situation is murkier for general purpose programs. Even though some work has been done on predicting cache contents (e.g. [43]), reasoning about the cache utilisation of programs doing anything more than simple array access is very difficult. Therefore, good profiling tools are crucial for getting good cache performance.

The performance monitoring counters of modern machines provide one way of measuring cache utilisation. However, they only give global statistics, such as the number of cache hits and misses, which are not much help to programmers trying to improve the cache utilisation of their programs. What is needed are tools that tie the cache utilisation statistics to particular parts of a program, so the programmer has some hope of knowing what changes will improve things.

DBI frameworks have the features necessary to build good profiling tools. In fact, profilers generally do not stretch a framework much, as they involve pervasive but simple analysis code, and fairly simple metadata—typically certain pieces of code are instrumented with calls to functions that update some profiling state, such as counts. Any DBI framework should be able to support the creation of profilers with ease. Cachegrind is a profiling DBA tool built with Valgrind.

3.1.4 Overview of Cachegrind

Cachegrind is a robust tool that performs a trace-driven simulation of a machine's cache as a program executes. The simulation is of a machine with a split L1 (I1 and D1) cache and a unified L2 cache. This configuration is typical in current x86-based machines.

Cachegrind tracks cache statistics (I1, D1 and L2 hits and misses) for every individual line of source code executed by the program. At program termination, it prints a summary of global statistics, and dumps the line-by-line information to a file. This information can then be used by an accompanying script to annotate the original source code with per-line cache statistics.

Cachegrind gains the benefits of DBI shared by all Valgrind tools: it is extremely easy to run, requiring no recompilation; it naturally covers all code executed, including libraries; and it works with programs written in any language. It is fairly fast for a tool of its kind. It is also quite flexible, being able to simulate different cache configurations by varying the size, line size and associativity of each cache; the configuration can be selected easily with a command line option. The cache simulation is not perfect, due to various reasons described later, but is good enough to provide useful results.

Cachegrind is extremely robust, and has been used by many programmers to profile large programs. It is part of the Valgrind distribution.

3.1.5 Chapter Structure

This chapter is structured as follows. Section 3.2 shows an example of Cachegrind's use. Section 3.3 explains how Cachegrind works. Section 3.4 shows how Cachegrind can be useful in practice, by showing how it was used to speed up Haskell programs compiled with the Glasgow Haskell Compiler. Section 3.5 discusses related work, and Section 3.6 concludes.

3.2 Using Cachegrind

Cachegrind is invoked from the command line like any other Valgrind tool. To profile a program `foo` with Cachegrind, one would use the following command:

```
valgrind --tool=cachegrind foo
```

The program then runs under Cachegrind's control. Cachegrind instruments every instruction (even instructions that do not perform data accesses, because the instruction cache is also simulated) so that when the instruction executes, the simulated caches are updated, and the hit and miss counts are also updated.

When the program terminates, Cachegrind emits two kinds of information. First, it prints some global statistics about cache utilisation. Second, it dumps the per-line hit and miss counts to a file.

Figure 3.2 shows the global summary statistics for the two programs from Figure 3.1 when run under Cachegrind. The information includes the total numbers of cache accesses and misses for instruction fetches (I1 and L2), cache accesses and misses for data fetches (D1 and L2), and L2 cache accesses and misses. It also shows the relevant miss rates. The results for the I1 cache for the two programs are, not surprisingly, identical; however, as expected, `slow.c` has sixteen times as many data cache misses (in both the D1 and L2 caches).

Figure 3.3 shows the result of annotating the same two programs with the per-instruction counts. Note that more than one instruction may map to each line of source code, in which case the counts for all those instructions are combined to give the line's counts. Each line's hit and miss counts are shown. The source lines are shown truncated here; in practice they are not. The `Ir`, `I1mr`, `I2mr` columns show the number of cache accesses, L1 misses and L2 misses for instruction reads. The `Dr`, `D1mr` and `D2mr` columns show the same counts for data reads, and the `Dw`, `D1mw`, `D2mw` columns show the same counts for data writes. The annotation script's output also shows the hit/miss counts for each function in the program, but that information is omitted here.

From the annotated source, a programmer can easily determine that the array assignment is causing a large number of cache misses, and rewrite the program to get better cache utilisation. As a secondary benefit, since one instruction cache read is performed per instruction executed, one can find out how many instructions are executed per line (the `Ir` count), which can be useful for instruction-count profiling and test coverage.

Output for `fast.c`:

```
I   refs:      94,566,217
I1  misses:           565
L2i misses:           560
I1  miss rate:        0.0%
L2i miss rate:        0.0%

D   refs:      52,518,512  (42,009,281 rd + 10,509,231 wr)
D1  misses:       656,324  (       801 rd +    655,523 wr)
L2d misses:       656,274  (       752 rd +    655,522 wr)
D1  miss rate:        1.2% (       0.0%   +        6.2%  )
L2d miss rate:        1.2% (       0.0%   +        6.2%  )

L2 refs:          656,889  (     1,366 rd +    655,523 wr)
L2 misses:        656,834  (     1,312 rd +    655,522 wr)
L2 miss rate:         0.4% (       0.0%   +        6.2%  )
```

Output for `slow.c`:

```
I   refs:      94,566,217
I1  misses:           565
L2i misses:           560
I1  miss rate:        0.0%
L2i miss rate:        0.0%

D   refs:      52,518,512  (42,009,281 rd + 10,509,231 wr)
D1  misses:    10,486,724  (       801 rd + 10,485,923 wr)
L2d misses:    10,486,674  (       752 rd + 10,485,922 wr)
D1  miss rate:       19.9% (       0.0%   +       99.7%  )
L2d miss rate:       19.9% (       0.0%   +       99.7%  )

L2 refs:       10,487,289  (     1,366 rd + 10,485,923 wr)
L2 misses:     10,487,234  (     1,312 rd + 10,485,922 wr)
L2 miss rate:         7.1% (       0.0%   +       99.7%  )
```

Figure 3.2: Global cache statistics for `fast.c` and `slow.c`

3.3 How Cachegrind Works

This section describes the details of how Cachegrind works. It begins with the main data structures, then covers instrumentation and related details, post-mortem code annotation, performance, Valgrind features that help Cachegrind, and finishes with Cachegrind's shortcomings.

3.3.1 Metadata

Cachegrind's metadata is stored in three main data structures, as the following paragraphs explain.

Global Cache State The first data structure is the *global cache state*, a global structure representing the state of the three (I1, D1, L2) simulated caches. It is updated on every instruction execution. It does not track each cache's contents since that is irrelevant for determining hits and misses; it only tracks

Annotated `fast.c`:

```
         Ir I1mr I2mr          Dr D1mr D2mr          Dw    D1mw    D2mw
          .    .    .           .    .    .           .       .       .  int main(void)
          6    1    1           0    0    0           1       0       0  {
          .    .    .           .    .    .           .       .       .     int h, i, j, a[1...
          .    .    .           .    .    .           .       .       .
         54    0    0          21    0    0           1       0       0     for (h = 0; h < ...
     51,240    0    0      20,490    0    0          10       0       0        for (i = 0; i...
 52,469,760    0    0  20,981,760    0    0      10,240       0       0           for (j = 0...
 41,943,040    0    0  20,971,520    0    0  10,485,760 655,360 655,360              a[i][j]...
          .    .    .           .    .    .           .       .       .
          1    0    0           0    0    0           0       0       0     return 0;
          2    0    0           2    0    0           0       0       0  }
```

Annotated `slow.c`:

```
         Ir I1mr I2mr          Dr D1mr D2mr          Dw        D1mw        D2mw
          .    .    .           .    .    .           .           .           .  int main(void)
          6    1    1           0    0    0           1           0           0  {
          .    .    .           .    .    .           .           .           .     int h, i, ...
          .    .    .           .    .    .           .           .           .
         54    0    0          21    0    0           1           0           0     for (h = 0...
     51,240    0    0      20,490    0    0          10           0           0        for (i ...
 52,469,760    0    0  20,981,760    0    0      10,240           0           0           for ...
 41,943,040    0    0  20,971,520    0    0  10,485,760  10,485,760  10,485,760              a...
          .    .    .           .    .    .           .           .           .
          1    0    0           0    0    0           0           0           0     return 0;
          2    0    0           2    0    0           0           0           0  }
```

Figure 3.3: Line-by-line cache statistics for `fast.c` and `slow.c`

which memory blocks are in each cache. It is updated with calls to the cache simulation functions, one per instruction executed. The functions are passed the necessary information about access addresses and sizes.

The simulation has the following particular characteristics.

1. *Write-allocate.* When a write miss occurs, the block written to is brought into the D1 cache. Most modern caches have this property.

2. *Modify instructions treated like reads.* Instructions that modify a memory location (e.g. `inc` and `dec`) are counted as doing just a read, i.e. a single data reference. This may seem strange, but since the write can never cause a miss (the read guarantees the block is in the cache) the write is not very interesting. Thus Cachegrind measures not the number of times the data cache is accessed, but the number of times a data cache miss could occur. This behaviour is the same as that used by the AMD Athlon hardware counters. It also has the benefit of simplifying the implementation—instructions that modify a memory word are treated like instructions that read memory.

3. *Bit-selection hash function.* The line(s) in the cache to which a memory block maps is chosen by the bits $[M, (M + N - 1)]$ of the address, where:

   (a) line size = 2^M bytes;

   (b) (cache size / line size) = 2^N bytes.

4. *Inclusive L2 cache.* The L2 cache replicates all the entries of the L1 cache. This is standard on Pentium chips, but AMD Athlons use an exclusive L2 cache that only holds blocks evicted from L1.

5. *LRU replacement.* The replacement algorithm is LRU (least recently used). When choosing which block to evict from a set, it chooses the least recently used block.

6. *Straddling accesses are merged.* References that straddle two cache lines are treated as follows:

   (a) if both blocks hit, it is counted as one hit;

   (b) if one block hits, and one misses, it is counted as one miss (and zero hits);

   (c) if both blocks miss, it is counted as one miss (not two).

The parameters of the simulated caches—cache size, associativity and line size—are determined in one of two ways. The default way is with the `cpuid` instruction when Cachegrind starts. For old x86 machines that do not have the `cpuid` instruction, or have the early incarnation that does not give any cache information, Cachegrind falls back to using a default configuration. Alternatively, the user can manually specify the parameters of all three caches from the command line.

If one wants to simulate a cache with properties different to those provided, it is easy to write an alternative cache simulator, as the interface to the analysis code is very simple.

Cost Centre Table The second data structure is the *cost centre table*, which contains the per-line *cost centres* (CCs). Every source line that gets instrumented and executed is allocated its own cost centre in the table, which records the number of cache accesses and misses caused by the instructions derived from that line.

The type of a cost centre is shown in Figure 3.4. `ULong` is an unsigned 64-bit integer; 64-bits are needed because the numbers involved can easily grow too big for 32 bits. `UChar` is an `unsigned char`, and `Addr` is an integral type the same size as a pointer (a 32-bit `unsigned int` for x86). In the `CC` structure, `a`, `m1` and `m2` represent the number of accesses, L1 misses and L2 misses respectively. The `lineCC` struct contains three `CC` elements, one each for instruction cache reads, data cache reads, and data cache writes. The `next` field is necessary because the cost centre table is a separately-chained hash table. The `line` field holds the source line number for this cost centre. A line number alone does not uniquely identify a source code line; a filename is also necessary. In fact, the table has three levels; the cost centres are grouped first by filename, then by function name, and only finally by line number. This is necessary to minimise the size of the output file dumped at termination; otherwise every cost centre would have to be annotated with its filename and function name. All instructions that do not have debug information share a special cost centre which has "???" as the filename and function name, and 0 as the line number.

```
typedef struct {
   ULong a;          // number of accesses
   ULong m1;         // number of L1 misses
   ULong m2;         // number of L2 misses
} CC;

typedef struct _lineCC lineCC;
struct _lineCC {
   Int     line;     // source code line number
   CC      Ir;       // CC for I-cache reads
   CC      Dr;       // CC for D-cache reads
   CC      Dw;       // CC for D-cache writes
   lineCC* next;     // next lineCC node in hash table
};
```

Figure 3.4: Cost centre types

```
typedef struct _instr_info instr_info;
struct _instr_info {
   Addr    instr_addr;      // instruction address
   UChar   instr_size;      // instruction's length in bytes
   UChar   data_size;       // size of data accessed by instruction
   lineCC* parent;          // lineCC for this instruction
};
```

Figure 3.5: Instr-info nodes

Happily, no look-ups need to be performed in the cost centre table at run-time because they can be performed at instrumentation-time, and their results effectively cached in the instr-info table, as the next paragraph explains.

Instr-info Table The third data structure is the *instr-info table*. It is used to cache unchanging information about each x86 instruction at instrumentation-time, which reduces the size of the added analysis code and improves its speed, by reducing the number of arguments that must be passed to the cache simulation functions.

Each x86 instruction instrumented is given a node in the table; each node has the type `instr_info` shown in Figure 3.5. The `instr_addr` field records the instruction's address, `instr_size` records the instruction's length in bytes, `data_size` records the size of the data accessed (0 if the instruction does not access memory), and `parent` points to the cost centre for the line of source code from which this instruction was derived, which is the cost centre that will be updated each time the instruction is executed.

As for the higher-level structure of the instr-info table, it is arranged so that the `instr_info` nodes for each basic block are stored together. This is necessary for the easy flushing of nodes when code is unloaded, as Section 3.3.3 explains.

The instr-info table is not necessary; its presence is merely an optimisation. The elements within an `instr_info` node could be passed to the cache simulation functions individually, but by bundling together those known at instrumentation time, fewer arguments need to be passed to the cache simulation functions, which reduces the size of the analysis code, and makes it faster. Also, each instruction has a permanent pointer to the appropriate line cost centre in the cost centre table, which means cost centres never need to be looked up at run-time, saving more time.

3.3.2 Instrumentation

Having explained the three main data structures, one can now consider instrumentation works. The first step is a quick pass over the basic block, which counts how many x86 instructions it represents. With this number known, an array of `instr_info` nodes can be allocated and inserted (as yet uninitialised) into the instr-info table.

Then the main instrumentation pass occurs. Cachegrind categorises each x86 instruction into one of the following five categories.

1. *None.* Instructions that do not access memory, e.g. `movl %eax, %ebx`.
2. *Read.* Instructions that read a memory location, e.g. `movl (%eax), %ebx`.
3. *Write.* Instructions that write a memory location, e.g. `movl %eax, (%ebx)`.
4. *Modify.* Instructions that read and write (modify) a memory location, e.g. `incl (%ecx)`.
5. *Read-write.* Instructions that read one memory location, and write another, e.g. `pushl (%edx)` (which reads `(%edx)`, writes `-4(%esp)`), or `movsw` (which reads `(%esi)`, writes `(%edi)`).

Because each x86 instruction is represented by multiple UCode instructions, Cachegrind does its categorisation by looking for UCode `LOAD` and `STORE` instructions within the `INCEIP` instructions that mark the boundary of each x86 instruction's UCode.

As well as categorising the instruction, Cachegrind also looks up its debug information: its line number, function name and filename. With this information, it finds the appropriate `lineCC` node in the cost centre table (creating the node if necessary), as described in Section 3.3.1. Recall that a special "???" cost centre is used if the debug information is missing. Cachegrind then initialises the appropriate `instr_info` node in the array that was allocated for the basic block (where the nth `instr_info` node in the array represents the nth x86 instruction in the basic block), as Section 3.3.1 described.

Once the `lineCC` and `instr_info` nodes are dealt with, the x86 instruction can be instrumented. Each x86 instruction is instrumented with a C call to one of the four cache simulation functions; which one depends on its categorisation. There are only four simulation functions for the five categories because the same function is shared by *Read* and *Modify* instructions (as Section 3.3.1 explained). The arguments passed to the cache simulation function also depend on the categorisation: *None* instructions need only a pointer to the `instr_info` node; *Read*, *Write* and *Modify* instructions need an

```
0x3A965CEC:  addl $0x8, %ecx

   0: GETL     %ECX, t0
   1: ADDL     $0x8, t0
   2: PUTL     t0, %ECX
*  3: MOVL     $0xB01E873C, t12        # address of 1st instr_info node
*  4: CCALLo   0xB1019901(t12)         # 'None' cache sim function
   5: INCEIPo  $3

0x3A965CEF:  movl (%ecx),%eax

*  7: MOVL     t0, t14                 # remember data address
   8: LDL      (t0), t4
   9: PUTL     t4, %EAX
* 10: MOVL     $0xB01E8748, t16        # address of 2nd instr_info node
* 11: CCALLo   0xB101992B(t16, t14)    # 'Read' cache sim function
  12: INCEIPo  $2
```

Figure 3.6: Example basic block, instrumented

`instr_info` pointer plus the contents of the data address register; *Read-write* instructions need an `instr_info` pointer plus the contents of both data address registers.

Figure 3.6 shows the analysis code added to part of a basic block. The original x86 instructions are shown. The analysis code added by Cachegrind is marked with '`*`'. Cachegrind puts the first x86 instruction in the *None* category, and just passes the address of the instruction's `instr_info` (`0xB01E873C`) to the C function (the C function is at address `0xB1019901`). The `MOV` is necessary because `CCALL` instructions cannot take literal operands. As mentioned in Section 2.4.3, the code generated for the `CCALL` preserves and restores any caller-save registers (including `%eflags`) as needed; the argument is passed in a register.

The second x86 instruction is a *Read* instruction and so the data address used in the `LOAD` (held in `t0` and copied into `t14`) needs to be passed as well. Note that UCode instruction #7 is not necessary here—virtual register `t0` could be passed directly to the C function, and `t14` never used—but copying it like this guards against the possibility of `t0` being updated between the `LOAD` and the `CCALL`. Currently no such updates ever happen, but it is best to be cautious in the face of possible future changes in Valgrind which would break such an assumption and affect Cachegrind's correctness; such changes might be very difficult to spot. The run-time cost of this extra copy is very small; it is outweighed by the other analysis code such as the simulation.

At run-time, each cache simulation function uses the passed information to update the global cache state as described in Section 3.3.1, and also the hit/miss counts for the instruction's `lineCC` cost centre.

3.3.3 Code Unloading

There is one further complication. As Section 2.3.5 mentioned, when code is unloaded translations are flushed from the translation table. Valgrind provides a hook whereby tools can be told when this happens, so that they can flush any state they are maintaining about the unloaded code.

For each basic block that gets unloaded, Cachegrind removes from the instr-info table all the `instr_info` nodes for all the instructions in that basic block, and frees the memory. This is why the instr-info table is structured so that all the nodes for a basic block are together, as Section 3.3.1 explained. Then, if new code is loaded into the memory region in which the old code was unloaded from, it will be freshly instrumented when necessary, and new `instr_info` nodes will be created.

The cost centre table does not need to be touched at all when code is unloaded, as its information is purely in terms of source code locations, and the memory locations in which instructions happen to be loaded are irrelevant.

3.3.4 Output and Source Annotation

When the client program terminates, Cachegrind dumps all the collected cost centres to a file called `cachegrind.out.`*`pid`*; the *`pid`* (process ID) suffix is used so that when Cachegrind traces into child processes there is one output file per process (as Section 2.3.10 explained). Cachegrind groups the cost centres by file and function, which is easy because the cost centre table is indexed in that way. The grouping makes the output files smaller, and makes the annotation easy. The global statistics (e.g. total cache accesses, total L1 misses) are calculated when traversing the cost centre table rather than during execution, to save time—the cache simulation functions are called so often that even one or two extra adds per call can cause a noticeable slow-down.

The file format is designed to minimise file sizes, while still being simple. To give an idea of typical sizes, on the machine described in Section 2.6 the file size for the utility "`ls`" is 9KB, and for the web browser Konqueror is 660KB. The format is also quite generic, and independent of Cachegrind. The script can be reused for annotating source code with any sets of counts; see Section 5.5.2 for an example of such reuse.

The very start of an output file is shown in Figure 3.7. The `desc:` lines are just comments about the data in the file; for Cachegrind they describe the cache configuration. The mandatory `cmd:` line gives the command line with which the program was invoked. The mandatory `events:` line names the counts within each cost centre. The `fl=` lines indicate the filename; its scope lasts until the next `fl=` line. The `fn=` lines are similar, but for function names. The remaining lines indicate the line number (in the first column) and the number of counts for each of the events given in the `events:` line. The line numbers are not necessarily in order.

So, for example, the first line of numeric data in Figure 3.7 states that line 486—which is in the function `CFF_Driver_Init()`—of the file `cffobjs.c` caused two `Ir` events (instruction reads), one `I1mr` event (I1 miss caused by an instruction read), one `I2mr` event (L2 miss caused by an instruction read), and one `Dw` event (data write).

Annotation is done by a Perl script. The script is fairly straightforward: it reads all the cost centres

```
desc: I1 cache:         65536 B, 64 B, 2-way associative
desc: D1 cache:         65536 B, 64 B, 2-way associative
desc: L2 cache:         262144 B, 64 B, 8-way associative
cmd: konqueror
events: Ir I1mr I2mr Dr D1mr D2mr Dw D1mw D2mw
fl=/usr/groups/cprg/share/freetype-2.1.2/src/cff/cffobjs.c
fn=CFF_Driver_Init
486 2 1 1 0 0 0 1 0 0
489 1 0 0 0 0 0 0 0 0
490 2 0 0 2 0 0 0 0 0
fn=CFF_Face_Init
259 6 0 0 0 0 0 6 0 0
260 6 0 0 0 0 0 6 0 0
335 8 0 0 4 0 0 4 0 0
263 42 0 0 12 0 0 24 0 0
265 18 0 0 0 0 0 0 0 0
```

Figure 3.7: Cachegrind output file example

from the file, and then runs through all the chosen source files, printing them out with per-line counts attached. The user can choose to annotate the source with a subset of the counts (e.g. only instruction cache statistics). The event names from the `events:` line are used as column headings for the count columns in the annotated code. The script minimises the size of the annotation output by excluding un-annotated parts of source files and automatically minimising column widths. It also gives function-by-function statistics. The source files can be specified manually, or found automatically by the script. The script issues a warning if the source files have changed more recently than the profiling data file. The script is quite fast and annotation times are typically very short. Figure 3.3 showed examples of source annotations.

3.3.5 Performance

Performance figures for Cachegrind on the SPEC CPU2000 suite were given in Figure 2.3. The slow-downs were in the range 20.5–93.3, with a median of 54.8. The slow-down is quite large, but not so large as to be unusable, especially since Cachegrind is not likely to be used terribly frequently with a program. Also, it compares quite well with the many cache simulators mentioned in [111], especially since Cachegrind models the I1, D1 and L2 caches. The code expansion factor (see Table 2.4) was 8.4–10.2, with a median of 9.5.

Quite a lot of effort went into making Cachegrind as fast as possible. To recap, the following features are particularly important.

- Unchanging information about each instruction is stored in its `instr_info` node, minimising argument passing, which reduces code size and improves speed.

  Also, the inclusion of cost centre addresses in the `instr_info` nodes at instrumentation-time is extremely important, because it means that no cost centre look-ups need to be performed at run-time.

- Only one C function call is made per instruction simulated (as opposed to having separate calls for the instruction cache and data cache accesses, for example). Just the cost of the C functions (e.g. preserving and restoring caller-save registers, plus the change in control-flow) accounts for a large chunk of execution time.

  In an attempt to reduce the C call overhead, I experimented with a trace buffer. The analysis code for each instruction placed the relevant pieces of information (those things normally passed as arguments to the C functions) in the buffer, and at the end of each basic block, all the entries for the basic block were batch-processed. This reduced the number of C calls made, but actually hurt performance. One reason was that the batch processor had to do a case selection on the instruction type (e.g. *None*, *Read*) for each instruction in order to decide how to update the simulated cache, and the jumps required resulted in a lot of mispredicted branches. By comparison, when each instruction has its own C call, the branch prediction is very straightforward as the same code always calls the same C functions. Another reason was that more memory traffic was caused by the constant reading and writing of the trace buffer, whereas the C function arguments are passed in registers.

- Summary counts are calculated at the end, rather than during execution.

- The output files can contain huge amounts of information; the file format was carefully chosen to minimise file sizes. This reduces I/O times for both Cachegrind and the annotation script.

Despite these optimisations, there is no escaping the fact that one C function call is made per x86 instruction, so a significant slow-down is unavoidable.

3.3.6 Useful Features

Cachegrind was very easy to implement as a Valgrind tool. As Figure 2.2 showed, Cachegrind is implemented in 1,683 lines of C code, including blank lines and comments. The annotation script is 891 lines of Perl, including blank lines and comments.

The following features of Valgrind helped simplify the implementation.

- It is quite easy to determine which instructions access memory from the UCode stream due to its load/store nature; much easier than by looking at the original x86 instructions.

- It is very easy to call C functions from code, thanks to the `CCALL` UCode instruction. The implementation of `CCALL` is very efficient too, minimising the registers that must be saved, and passing arguments in registers where possible. And thanks to the use of virtual registers in UCode, Cachegrind does not have to worry about putting the arguments into the right registers, since Valgrind looks after those details in its register re-allocation and code generation phases.

- Valgrind's automatic reading of debug information made mapping of instructions to lines of source code extremely easy.

- Valgrind's hook for telling tools when code is unloaded is crucial, otherwise the caching of information about each instruction in the instr-info table would not be possible.

If Cachegrind were a stand-alone tool, it would undoubtedly require at least ten times as much code, if not more, so building it with Valgrind definitely saved a great deal of effort. However, Cachegrind is only a moderately heavyweight DBA tool. The analysis code it uses is pervasive (every instruction is instrumented) but not interconnected (each piece of analysis code is independent). It maintains a lot of state, but no location metadata. Therefore using Valgrind to build it was very helpful, but not crucial; it could have been implemented in some other DBI frameworks with a similar amount of effort. Nonetheless, it does show that Valgrind is suitable for building profiling DBA tools.

3.3.7 Simulation Shortcomings

Cachegrind is certainly not perfect. First, the trace of addresses passed to the simulation is imperfect, due to the following reasons.

- Because Valgrind tools cannot instrument kernel code, code executed within system calls is not considered, nor is the thread scheduling and signal handling performed by the kernel.
- Because some client code is replaced with code in Valgrind's core which is not instrumented (see the discussion of *Core-space* in Section 2.4.2), some client code is not considered.
- Other processes running concurrently are not considered. When considering the program by itself (the more likely option), this is a good thing; when considering the machine in its entirety, this is bad.
- Valgrind's conversion from x86 to UCode loses a small amount of the original information. For example, the common idiom:

  ```
     call x
  x: pop %eax
  ```

  puts the program counter in `%eax`. Valgrind's core spots this idiom and translates it to UCode as if the original instruction was:

  ```
  movl &x, %eax
  ```

  where `&x` represents the literal address of the second instruction. Valgrind can do this because it knows at instrumentation-time what the address of that instruction is. The resulting UCode is more efficient, albeit slightly misleading for Cachegrind.

  The result is that this sequence of two instructions that does a store and a load is treated by Cachegrind as if it is one instruction that does not access memory at all.

Second, those addresses passed to the simulation are not truly correct, for the following reasons.

- Simulation is done entirely with virtual addresses; there is no consideration of the virtual-to-physical address mappings. Thus, it is not a true representation of what is happening in the cache.

- Cache misses not visible at the machine code level are not modelled, for example, translation look-aside buffer (TLB) misses, and cache misses from speculative execution that was annulled.

- Valgrind's thread scheduling is different to that of the standard threads library, which can change the execution order and thus the results.

- The memory layout of the client will be different under Cachegrind, which can also affect the results. For example, when simulating caches that are not fully associative, the conflict misses that occur will be different due to the different memory layout.

Third, the simulation itself has the following inaccuracies.

- Some approximations are made in the cache simulation. In particular, FPU instructions with data sizes of 28 and 108 bytes (e.g. `fsave`) are treated as though they only access 16 bytes. These instructions are rare enough that this should have little effect.

- The simulated cache does not necessarily match the real machine's behaviour. For example, real caches may not use LRU replacement; indeed, it is very difficult to find out what replacement algorithms are used in machines such as modern Pentiums. Also, various real machines have certain quirks not modelled by the simulation. For example, upon an instruction cache miss the model 4 Athlon loads the missed block and also prefetches the following block.

- Prefetching instructions are ignored. (Prefetches are not represented at all in UCode, and the code generated by Valgrind does not preserve them.)

This list is depressingly long. However, Cachegrind's inaccuracy is not a big problem, so long as one does not expect Cachegrind to give a picture-perfect simulation of what is happening in the real cache. Instead, one should view Cachegrind as a way of gaining insight into the locality of a program, and how it would behave on some kind of generic machine. Section 3.4 shows that this is enough for Cachegrind to be useful in practice.

3.3.8 Usability Shortcomings

The nature of the information Cachegrind produces is not ideal. With any profiling tool, one wants a minimal "conceptual distance" between the information produced, and the action required to act on that information. For example, with traditional time-based profilers that say how much time each function takes, this conceptual distance is quite small—if a function `f()` accounts for 80% of run-time, one knows immediately that speeding it up will have a large effect. By contrast, Cachegrind's conceptual distance is larger, for two reasons.

1. The number of cache misses occurring in a section of code does not necessarily relate obviously to the execution time of that section. Ideally, Cachegrind would specify how many cycles of machine time each instruction accounts for. However, given the complexity of modern machines, this is not feasible. One could use crude cost models to approximate cycle counts, but they are so likely to be misleading that this would be more of a hindrance than a help. (However, Section 3.4.3 provides a rare counter-example.)

2. Even if you know a line of source code is causing a lot of cache misses, and you are confident the misses are slowing down the program a lot, it is not always clear what can be done to improve this. It can be strongly indicative that, for example, a data structure could be redesigned. But the results rarely provide a "smoking gun", and some non-trivial insight about how the program interacts with the caches is required to act upon them.

There is no silver bullet; optimising a program's cache utilisation is hard, and often requires trial and error. What Cachegrind does is turn an extremely difficult problem into a moderately difficult one.

3.4 In Practice

This section shows how Cachegrind is useful. It describes an example of Cachegrind being used "in anger", to analyse the cache utilisation of Haskell programs compiled with the Glasgow Haskell Compiler (GHC). The information found with Cachegrind was used to guide the insertion of prefetching instructions into GHC code and GHC's run-time system, which sped up a set of benchmark programs by up to 22%. This section is a much-abbreviated version of [80].

3.4.1 Language and Implementation

Haskell [88] is a polymorphically typed, lazy, purely functional programming language widely used in the research community. The Glasgow Haskell Compiler (GHC) [47] is a highly optimising "industrial-strength" compiler for Haskell. The compiler itself is written in Haskell; its run-time system is written in C. Although its distribution contains an interpreter, GHC is a true compiler designed primarily for creating stand-alone programs. The optimisations it performs include full laziness, deforestation, let-floating, beta reduction, lambda lifting and strictness optimisations; it also supports unboxed values [90]. It is widely considered to be the fastest implementation of a lazy functional language [51]. Because it is highly optimising, it is not a soft target. This is important, since optimising non-optimised systems is always less of a challenge than optimising optimised systems.

GHC can compile via C, or use its x86 native code generator; the distinction is unimportant here, as programs compiled by the two routes have extremely similar cache utilisations. All measured programs were compiled with a development version of GHC (derived from v5.02.2), via C using GCC 3.0.4, using GHC's `-O` optimisation flag.

Program	Description	lines
`anna`	Frontier-based strictness analyser	5740
`cacheprof`	x86 assembly code annotator	1489
`compress`	LZW text compression	403
`compress2`	Text compression	147
`fulsom`	Solid modeller	857
`gamteb`	Monte Carlo photon transport	510
`hidden`	PostScript polygon renderer	362
`hpg`	Random Haskell program generator	761
`infer`	Hindley-Milner type inference	561
`parser`	Partial Haskell parser	932
`rsa`	RSA file encryptor	48
`symalg`	Symbolic algebra program	831
`ghc`	GHC, no optimisation	78950
`ghc -O`	GHC, with `-O` optimisation	78950

Table 3.2: Haskell program descriptions

3.4.2 Benchmark Suite

Twelve of the benchmark programs tested come from the "real" part of the `nofib` suite [85] of Haskell programs. These programs were all written to perform an actual task; most are a reasonable size, and none have trivial input or behaviour. The other program tested was GHC itself, compiling a large module with and without optimisation. The benchmark programs are described in Table 3.2, and their sizes in lines of code (minus blank lines and comments) are given. Inputs were chosen so each program ran for about 2–3 seconds, except for the `ghc` benchmarks, which were substantially longer. This was long enough to realistically stress the cache, but short enough that the programs ran for reasonable times when run under Cachegrind.

3.4.3 Motivating Measurements

Before using Cachegrind, I made a series of measurements on the `nofib` programs, to determine whether cache stalls were accounting for a significant part of their execution times. The machine used for the experiments was the AMD Athlon described in Figure 3.1, running Red Hat Linux 7.1, kernel version 2.4.7. The measurements were taken with Rabbit [53], which provides control over the Athlon's hardware performance counters [3].

The conclusion was that L2 cache data stalls accounted for 1–60% of execution times. This was determined using a simple but surprisingly accurate execution cost model, which took into account only the four largest components of execution time: instruction execution, stalls due to L1 and L2 cache data misses, and stalls due to branch mispredictions. Using the cost model, the proportion of execution time taken up by L2 miss and branch misprediction stalls—the stalls that were found to be significant for Glasgow Haskell programs—could be estimated confidently, just from the counts provided by the Athlon's hardware counters.

In short, L2 cache misses were a big problem. The obvious next question was "how can things be

Program	Instr %	Ref %	D1 miss %	D2 miss %
anna	99.4	71.6	92.7	80.2
cacheprof	99.5	82.6	92.5	69.5
compress	99.4	84.0	97.0	64.5
compress2	99.4	83.3	94.7	94.1
fulsom	99.5	78.3	89.8	72.8
gamteb	99.2	81.8	91.5	71.0
hidden	99.2	74.5	94.6	54.3
hpg	97.4	80.8	98.1	38.2
infer	99.0	76.3	96.0	90.6
parser	99.3	81.1	89.2	81.5
rsa	99.4	91.4	97.3	83.1
symalg	99.2	96.8	96.8	45.7
ghc	99.4	78.6	87.3	88.0
ghc -O	100.0	78.3	86.6	81.2

Table 3.3: Ratios between hardware counter and software simulation event counts

improved?" This is where Cachegrind was helpful.

3.4.4 Quantifying Cachegrind's Accuracy

As discussed in Section 3.3.7, Cachegrind's simulation is not perfect. Table 3.3 compares its global statistics directly with the results from the hardware counters. Column 1 gives the program name. Column 2 gives the ratio of instructions counted by Cachegrind to retired instructions counted by Rabbit (event `0xc0`). This is the best comparison of the two techniques, as they are measuring exactly the same event. As expected, Cachegrind gave marginally lower counts than Rabbit, because unlike Rabbit it does not measure other processes and the kernel. Despite this, Cachegrind counted more than 99% of the events counted by Rabbit for all programs except `hpg`, for which it counted 97.4%. Column 3 contains the memory reference ratios, where Rabbit measures the number of data cache accesses (event `0x40`). Cachegrind falls further short here, by 3–28%. As mentioned in Section 3.3.7, this is because some cache accesses are occurring that are not visible at the program level, such as those from TLB misses. Columns 4 and 5 give the D1 cache miss and L2 cache data miss ratios, where Rabbit is measuring the number of data cache refills from L2 and data cache refills from system (Athlon events `0x42`, `0x43`). Cachegrind underestimates these misses by 3–62%.

Cachegrind's results gives a general picture of where cache misses occur for machines with this kind of cache configuration, rather than matching exactly every miss that occurs for the Athlon. Although this picture is not perfect, it gives a good indication of where cache misses are occurring in these programs.

3.4.5 Use of Cachegrind

All benchmark programs were run under Cachegrind, and each line of code in the compiled programs and the GHC run-time system was annotated—at the level of assembly code for the compiled GHC

Program	GC	Prog	Both	Theory	Ratio
anna	3%	0%	4%	5%	0.8
cacheprof	3%	1%	5%	9%	0.6
compress	2%	2%	5%	7%	0.7
compress2	3%	9%	12%	25%	0.5
fulsom	1%	17%	17%	31%	0.6
gamteb	0%	-0%	0%	6%	0.0
hidden	1%	3%	2%	3%	0.7
hpg	4%	1%	4%	3%	1.3
infer	3%	8%	9%	8%	1.1
parser	1%	19%	22%	28%	0.8
rsa	3%	-1%	2%	$< 1\%$	
symalg	1%	1%	1%	$< 1\%$	
ghc	1%	17%	17%	27%	0.6
ghc -O	2%	12%	14%	24%	0.6

Table 3.4: Effect of prefetching

code, and at the level of C for the run-time system—with its number of read and write references and misses. I concentrated on L2 data misses because data misses are much more frequent than instruction misses, and L2 misses are much more costly than L1 misses. L2 misses were concentrated around a small number of parts of the system, some of which were in the programs' compiled code, and some of which were in the garbage collector.

3.4.6 Avoiding Data Write Misses

Most write misses in Glasgow Haskell programs are unnecessary. Almost all heap writes made by Glasgow Haskell programs are sequential, due to GHC's execution mechanism [89] which does many small allocations in a linear path through memory, and due to its copying garbage collector.

Write misses occur only for the first word in a cache line. There is no need to read the memory block into the D1 cache upon a write miss, as is done in a write-allocate cache; the rest of the line will soon be overwritten. It would be better to write the word directly to the D1 cache and invalidate the rest of the cache line. Unfortunately x86 machines have no such write-invalidate instruction.

Prefetching mitigated this problem. Because writes are sequential it is simple to insert prefetches to ensure memory blocks are in the cache by the time they are written to. I performed some preliminary experiments with the Athlon's `prefetchw` instruction, fetching ahead 64 bytes each time a new heap data structure was allocated or copied by the garbage collector. The changes required were simple, and increased code sizes by only 0.8–1.6%. Figure 3.4 shows the results: columns 2–4 give the speed improvements when the prefetching was applied to just the garbage collector, just program allocations, and both. The improvements are quite respectable: programs ran up to 22% faster, and almost none slowed down.

If a write-invalidate instruction existed that cost no more than a normal write, one can estimate the potential speed-up it would provide by multiplying the proportion of write misses by the proportion of execution time taken up by L2 data cache stalls, which gives the expected speed-ups shown in column 5 of Table 3.4. ([80] explains the calculation in detail.) The prefetching technique—which is applicable to any program using copying garbage collection, not just Glasgow Haskell programs—obtained half or more of this theoretical figure for almost all programs, as shown by column 6 which gives the ratio between theoretical and actual speed-ups. This is pleasing since more prefetches are performed than necessary (one per data structure allocated/copied, about six per cache line), and prefetching increases memory bandwidth requirements.

3.5 Related Work

A great many profiling tools have been built over the years. Many DBA profiling tools are trace-driven, i.e. their input is a trace of information from an event of interest, and the output is some kind of summary based on the input. For example, an address trace provides the input for cache/memory simulators; a branch taken/not-taken trace provides the input for a branch prediction simulator; a trace of heap allocations provides the input for a heap profiler. Some DBA profiling tools use sampling to obtain interesting timings about programs.

Uhlig and Mudge wrote an extremely thorough survey of more than 50 trace-driven cache/memory simulation tools [111]; it is so good that I will not attempt to mention any of the tools it covers, but only give a brief summary of its three criteria for categorising these tools. First, the trace collection method, one of: hardware probes, microcode modification, instruction-set emulation, static code annotation, or single-step execution. Cachegrind uses instruction-set emulation.[1] Second, the trace reduction method, one of: compression, compaction, filtering, sampling, or none. Cachegrind uses none—trace reduction is only necessary for tools that store address traces for processing off-line. Third, the trace processing method, one of: basic, stack, and forest. Cachegrind uses basic processing, and so can only simulate one cache configuration at a time. From this survey it is clear that trace-driven cache simulators are not a new idea; indeed, several have been built with some of the DBI frameworks described in Section 2.7.3. However, Cachegrind compares quite favourably with the tools described in the survey, not requiring any special software or hardware, and being extremely easy to use, relatively fast, fairly accurate, and giving line-by-line hit/miss information.

Cachegrind was based directly on Cacheprof, by Julian Seward. Cacheprof had the same goal as Cachegrind: line-by-line source code annotation of cache hit and miss counts. However, it used static assembly code-level instrumentation, being implemented as a GCC wrapper which instrumented the assembly code emitted by GCC with calls to the cache simulation functions. This was a major disadvantage, because all program code, including libraries, had to be recompiled to be included in the simulation. If library code was not instrumented, the simulation would only be partial, and far less trustworthy. Cacheprof is now defunct, having been superseded by Cachegrind, which is faster, easier to use, and has a more detailed cache simulation.

Josef Weidendorfer's Calltree [116] is a Valgrind tool that extends Cachegrind. As well as simulating the caches, it collects call tree information, and can provide greater context for cost centres, e.g. on

[1] Their terminology is different to mine.

a thread-by-thread basis. It comes with a graphical viewer called KCachegrind which provides an impressive number of ways of viewing the collected information. It is available [115] free-of-charge under the GNU GPL.

Several works have been published about the cache utilisation of declarative languages, and ways to improve it; [80] mentions some of them.

3.6 Conclusion

This chapter described Cachegrind, a cache profiling tool, which was built with Valgrind. Cachegrind represents a classic archetype of trace-driven profiling tools—all the standard features of profiling tools are present, particularly analysis code attached to particular pieces of code, and calls to C functions that update counters and simulation state. This chapter showed how it was used to speed up Glasgow Haskell programs by up to 22%.

Cachegrind is not a particularly novel tool; many similar trace-driven cache profilers have been built. Nonetheless, it is among the easiest to use since it does not require any recompilation of source files, and it is fairly fast and accurate. Finally, it also shows the importance of Valgrind's support for heavyweight DBA.

CHAPTER 4

A Checking Tool

This chapter describes a novel checking tool, built with Valgrind, which uses location metadata and shadow computation.

4.1 Introduction

This chapter describes Annelid, a bounds-checking tool, which was built with Valgrind.

4.1.1 Checking Tools

The second major group of DBA tools, after profilers, are checkers. Checkers collect information about a running program, and use that information to check certain actions performed by the program. If any of these actions are erroneous (or even suspicious), a checker can issue some kind of diagnostic error message. A programmer can act on these error messages and rewrite the program so it does not exhibit the erroneous behaviour. These tools are basically designed to find bugs. Many such tools have been written, most commonly to find memory errors such as bad memory accesses and memory leaks.

Checking tools use passive analysis code (as described in Section 2.7.1). Some checkers also use active analysis code, and can modify a program's execution so certain objectionable actions do not occur. This may be done in several ways, such as aborting the program altogether, making the action fail, or replacing the action with a less objectionable one. These tools are often designed to prevent security breaches, such as stack-smashing attacks [62].

Many of the problems found by checkers can be prevented at the language level; for example, memory leaks are rarely an issue for programs written in garbage-collected languages. Nonetheless, many programs are still written in relatively unsafe languages such as C, C++, and Fortran, and so checkers are very important tools. They can be particularly useful in detecting hard-to-find bugs that might lurk undiscovered for a long time. Bounds errors are one such class of bugs.

4.1.2 Bounds Errors

Low-level programming languages like C and C++ provide raw memory pointers, permit pointer arithmetic, and do not check bounds when accessing arrays. This can result in very efficient code, but the unfortunate side-effect is that accidentally accessing the wrong memory is a very common programming error.

The most obvious example in this class of errors is exceeding the bounds of an array. However the bounds of non-array data objects can also be violated, such as heap blocks, C structs, and stack frames. I will describe as a *bounds error* any memory access which falls outside the intended memory range. These errors are not difficult to introduce, and they can cause a huge range of bugs, some of which can be extremely subtle and lurk undetected for years. Because of this, tools for preventing and identifying them are extremely useful.

4.1.3 Bounds-Checking

Many bounds-checking tools are available, using a variety of analyses. Indeed, Memcheck, described in Section 2.4.1, does certain kinds of bounds-checking. No single analysis is ideal, and each one has a different set of characteristics, including:

- which regions of memory (heap, static, stack, mapped segments) it works with;
- the number of false positives it produces;
- which parts of a program it covers, in particular how it works with libraries;
- what level of compiler support it requires.

A particularly powerful bounds-checking approach is to use *fat pointers*, whereby a tool tracks, for each pointer, metadata which describes the memory range the pointer can legitimately access. Any accesses through a pointer that are outside its legitimate range are flagged as errors. (Fat pointers are discussed in more detail in the related work of Section 4.6.2.)

4.1.4 Overview of Annelid

Annelid is a prototype tool implementing a novel form of fat pointer bounds-checking. The novelty comes not from the use of fat pointers, but rather the fact that they are maintained entirely dynamically; Annelid is the first tool that does fat pointer-based bounds-checking using DBA. It gains the benefits of DBI shared by all Valgrind tools: it is extremely easy to run, requiring no recompilation; it naturally covers all code executed, including libraries; and it works with programs written in any language. Importantly, the pointer range metavalues are maintained separately from the pointers themselves, so pointer sizes do not change. Also, the coverage of libraries is particularly important, because inadequate handling of library code is a major shortcoming of many previous bounds-checking analyses.

The downside is that Annelid's analysis is neither sound nor complete. It spots many, but not all, bounds errors in the heap, the stack, and in static memory; it gives few false positives. The analysis performs best when debug information and symbol tables are present in the compiled program, but *degrades gracefully* when this information is missing—fewer errors are found, but false positives tend not to increase.

Annelid was conceived as an alternative to Memcheck. While Memcheck works very well, it misses certain classes of errors, such as bounds errors on static memory and the stack (as Section 4.6.1 discusses). The idea was that Annelid would use a stronger checking DBA to find errors that Memcheck misses. As it turned out, Annelid has its advantages and disadvantages, and is unlikely to be a world-beater in its own right. However, it does push the boundaries of the design-space of bounds-checking tools, and might point the way to a future analysis that is a world-beater.

Annelid was first described in [79]. Annelid is currently a prototype tool, providing a proof-of-concept implementation of the bounds-checking DBA. It can run medium-size programs such as the SPEC CPU2000 benchmarks and common Linux programs. It is not part of the Valgrind distribution.

4.1.5 Chapter Structure

This chapter is structured as follows. Section 4.2 shows an example of Annelid's use. Section 4.3 describes the analysis at an idealised high level, and Section 4.4 describes what is actually implemented within Annelid, discusses how that differs from the idealised design, and gives more low-level details. Section 4.5 discusses the shortcomings of the analysis, and ways in which a compiler might co-operate with Annelid so that it can give better results. Section 4.6 describes related work, and discusses how Annelid's analysis compares to others. Section 4.7 discusses future work and concludes.

4.2 Using Annelid

Annelid is invoked from the command line like any other Valgrind tool. To check a program `foo` with Annelid, one would use the following command:

```
valgrind --tool=annelid foo
```

The program then runs under Annelid's control. Annelid instruments every instruction and memory operation so that pointers are tracked, and their use is checked. When Annelid detects a bounds error, it issues an error message indicating which line in the program caused the error.

Figure 4.1 shows a short C program, `bad.c`. This contrived program shows three common errors: two array overruns, and an access to a freed heap block. Figure 4.2 shows the output produced by Annelid. Annelid uses Valgrind's built-in support for error recording, and the error messages are deliberately similar to Memcheck's. Each line is prefixed with the running program's process ID. Each error message consists of a description of the error, the location of the error, a description of the memory object(s) involved, and the location where the memory object was allocated or freed (whichever happened most recently). The functions `malloc()` and `free()` are identified as being in the file `vg_replace_malloc.c` because that is the file that contains Annelid's implementations of these functions, which override the standard ones.

The program was compiled with `-g` to include debug information. If it had not been, the code locations would have been less precise, identifying only the code addresses and file names, not ac-

```
#include <stdlib.h>

int static_array[10];

int main(void)
{
   int  i;
   int* heap_array = malloc(10 * sizeof(int));

   for (i = 0; i <= 10; i++) {
      heap_array  [i] = 0;    // overrun when i==10
      static_array[i] = 0;    // overrun when i==10
   }
   free(heap_array);
   heap_array[0] = 0;         // block has been freed
}
```

Figure 4.1: Program with bounds errors: `bad.c`

tual line numbers. Also, the second error involving the static array would not have been found, as Section 4.3.6 discusses.

4.3 How Annelid Works: Design

This section provides a high-level, idealised description of the bounds-checking analysis. Certain details are kept vague, and subsequently fleshed out in Section 4.4.

4.3.1 Overview

The basic idea is simple. Every pointer has a range of addresses it can legitimately access. The range depends on where the pointer comes from. For example, the legitimate range of a pointer returned by `malloc()` is that of the just-allocated heap block. That range is called the pointer's *segment*. All memory accesses are checked to make sure that the memory accessed is within the accessing pointer's segment, and any violations are reported.

Pointers are often used in operations other than memory accesses. The obvious example is pointer arithmetic; for example, array elements are indexed using addition on the array's base pointer. However, if two pointers are added, the result should not be used to access memory; nor should a non-pointer value be used to access memory. Thus the analysis also needs to know which program values are non-pointers. The result is that every value has a run-time type, and the analysis needs a simple type system to determine the type of all values produced by the program. These types are location metadata, and they are propagated by shadow computation.

```
==16884== Invalid write of size 4
==16884==    at 0x8048398: main (bad.c:11)
==16884==  Address 0x40D1C040 is 0 bytes after the expected range,
==16884==   the 40-byte heap block allocated
==16884==    at 0x400216E7: malloc (vg_replace_malloc.c:161)
==16884==    by 0x8048375: main (bad.c:8)
==16884==
==16884== Invalid write of size 4
==16884==    at 0x80483A2: main (bad.c:12)
==16884==  Address 0x80495E8 is 0 bytes after the expected range,
==16884==   a 40-byte static array in the main executable
==16884==
==16884== Invalid write of size 4
==16884==    at 0x80483C5: main (bad.c:15)
==16884==  Address 0x40D1C018 is 0 bytes inside the once-expected range,
==16884==   the 40-byte heap block freed
==16884==    at 0x40021CE9: free (vg_replace_malloc.c:187)
==16884==    by 0x80483BE: main (bad.c:14)
```

Figure 4.2: Bounds errors detected for `bad.c`

4.3.2 Metadata

The analysis requires maintenance of two kinds of metadata. First is the record of the segments in the program, each of which has the following form.

- X, a *segment-type*, describes a segment; this includes its base address, size, location (heap, stack, or static), and status (in-use or freed).

Each segment is given a segment-type X.

Second is the location metadata shadowing each register and memory word, describing the run-time type of the word. This metadata has one of the following three forms.

- n, a *non-pointer-type*, describes a value which is known to be a non-pointer.
- $p(X)$, a *pointer-type*, describes a value which is known to be a pointer to a segment X.
- u, an *unknown-type*, describes a value for which the type is unknown.

Each word-sized value is shadowed by one of n, u or $p(X)$. In principle, every value produced, of any size, can be assigned a type. In practice, the shadow value tracking can be done at word-sized granularity because all memory accesses are through word-sized pointers.

4.3.3 Checking Accesses

Every load and store is checked. When accessing a memory location m through a value x the analysis looks at the type of x and behaves accordingly.

- n: Issue an error about accessing memory through a non-pointer.
- u: Do nothing.
- $p(X)$: If m is outside the range of X, issue an error message about accessing memory beyond the legitimate range of x; if X is a freed segment, issue an error message about accessing memory through a dangling pointer.

Note that in all cases the memory access itself happens unimpeded, but only after the check has been done. This is so an error message will be issued before any erroneous memory access occurs which may crash the client program.

4.3.4 Life-cycle of a Segment

There are four steps in dealing with each segment.

1. Identify when the segment is allocated, in order to create a segment-type describing it.
2. Upon allocation, set the metavalue of each word within the segment appropriately.
3. Identify each pointer to the segment, and set the metavalues of these pointers to the appropriate pointer-type.
4. Identify when the segment is freed, in order to change the status of its segment-type to "freed".

The first two steps occur when a segment is allocated. The third step can occur at various times, depending on the what region of memory the segment belongs to. The fourth step occurs when a segment is freed.

The way these aspects are handled differs between the heap, static and stack segments. In addition, the exact meaning of the term "segment", and thus the kinds of errors found, differs between the three memory areas. The following sections discuss these differences. Section 4.4.2 describes how Annelid deallocates segment-types.

4.3.5 Heap Segments

Heap segments are the easiest to deal with. For the heap, each heap segment represents one heap block. The four steps for dealing with heap blocks are as follows.

1. By intercepting `malloc()`, `calloc()`, `realloc()`, `new`, and `new[]`, the analysis knows for every heap segment its address, size, and where it was allocated. It can thus easily create the new segment-type and store it in a data structure.
2. All the words within a newly allocated heap block have their metavalues set to n, since they should definitely not be used to access memory, being either zeroed (for `calloc()`) or uninitialised.

3. The pointer returned by the allocation function has its shadow set to $p(X)$, where X was the segment-type just created. At this point, this is the only pointer to the segment.

4. By intercepting `free()`, `realloc()`, `delete`, and `delete[]`, the analysis knows when heap segments are freed; it can look up the segment-type in the data structure by its address (from the pointer argument) and change the segment-type's status to "freed".

Note that the analysis as described only detects overruns past the edges of heap blocks. If the heap block contains a struct which contains two adjacent arrays, overruns of the first array into the second will not be caught. This could be improved by using debug information to identify pointer types. Then with some effort, field accesses could be identified by looking at offsets from the block pointer. Section 4.3.6 discusses the difficulties in identifying array overruns in these cases. Section 4.4.8 explains how to handle programs that use custom allocators.

Finally variable-length arrays are an interesting case. Consider the following C code.

```
struct foo { int a; float b; int c[0]; };
struct foo f = malloc(sizeof(foo) + 10*sizeof(int));
f.c[1] = 0;
```

The analysis will handle this valid code without a problem, since the entire heap block is considered a single segment, and `f.c[1]` is within that segment. In comparison, this is a case that source analysis bounds-checking tools may get wrong, since they might think that `f.c[1]` is out of bounds.

4.3.6 Static Segments

Identifying static segments is more difficult. First, identifying static segments by looking only at machine code at run-time is close to impossible. So the analysis relies on debug information to do this; if this information is not present, all pointers to static data objects will have the type u, and no accesses to static memory will be checked.

The analysis can create segment-types for each static data object, e.g. arrays, structs, integers; thus for static memory, each segment represents a single data object. The four steps for dealing with static segments are as follows.

1. If debug information is present, the analysis identifies static data objects in the main executable at start-up, and for shared objects when they are mapped into memory.

2. Newly loaded static memory can contain legitimate pointers. Those the analysis can identify, from debug information, are shadowed to point to the appropriate static segments, as identified from their initial value. The remaining words have their segment-type set according to their type as specified in the debug information. Or if that information is missing, according to their value; if a word is clearly a non-pointer—e.g. a small integer—it is given the type n, otherwise it is given the type u.

```
# load a[0]                           # load a[5]
movl $0x8049400, %eax                 movl $0x8049400, %eax
movl (%eax), %edx                     movl 20(%eax), %edx

# load a[5]                           # load a[5]
movl $0x8049414, %eax                 movl $0x8049400, %eax
movl (%eax), %edx                     movl $5, %ebx
                                      movl (%eax,%ebx,4), %edx

# load a[5]                           # load a[10]
movl $5, %eax                         movl $0x8049428, %eax
movl 0x8049400(,%eax,4),%edx          movl (%eax), %edx
```

Figure 4.3: Hard-wired static array accesses

3. Identifying pointers to static segments is harder than identifying pointers to heap segments. This is because compilers have a lot of control over where static data objects are put, and so can hard-wire absolute addresses into the compiled program.

   One possibility is to rely on a simple assumption: that any constant value that looks like a pointer to an object, is a pointer to that object (even if it does not point to the start of the object). Consider the x86 machine code snippets in Figure 4.3 used to load elements of an integer array `a[10]`, located at address `0x8049400`, into register `%edx`. The assumption works for the constants in all these cases except the last one; it is a bounds error, because the pointer value used falls outside the range of `a[]`. However, detecting this error is not possible in general, given that another array `b[]` might lie directly after `a[]`, in which case an access to `a[10]` is indistinguishable from an access to `b[0]`.

   Unfortunately, this approach sometimes backfires. For example, the program `gap` in the SPEC CPU2000 benchmarks initialises one static pointer to a static array with `p = &a[-1]`; `p` is then incremented before being used to access memory. This is not legal ISO C, but it works with most C compilers. When compiled with GCC, the address `&a[-1]` falls within the segment of another array lying adjacent; thus the pointer is shadowed with the wrong segment-type, and false positives occur when `p` is used to access `a[]`. A similar problem occurs with the benchmark `gcc`, in which an array access of the form `a[var - CONST]` (which is legal ISO C) occurs, where `var` $\geq$ `CONST` at run-time; the address `a[-CONST]` is resolved by the compiler, seemingly giving an address in an adjacent array.

   The alternative is to be more conservative in identifying static pointers, by only identifying those that point exactly to the beginning of static arrays. Unfortunately, this means some static array accesses will be missed, such as in the second snippet on the left-hand side of Figure 4.3, and so some errors will be missed. In particular, array references like `a[i-1]` are quite common and will not be handled if the compiler computes `a[-1]` at compile-time.

   As well as appearing in the code, static pointers can also occur in static data. These can be identified from the debug information, and their segment-type set appropriately.

4. When shared objects are unmapped from memory, all static arrays within the unmapped range can be marked as freed. This is safe because static memory for a shared object is always contiguous, and never overlaps with heap or stack memory.

As with heap segments, arrays within static structs could be identified with some effort. However, as with top-level arrays, direct accesses that exceed array bounds—like the `a[10]` case mentioned previously—cannot be detected.

One further complication is the use of C-style unions, whereby a data object's segment can change in size over time if its type changes. Since the analysis cannot in general determine which type a union actually has at any point, it must be conservative and assume the largest possible segment size. This can cause it to miss some bounds errors.

4.3.7 Stack Segments

The stack is the final area of memory to deal with. One could try to identify bounds errors on individual arrays on the stack, but a simpler goal is to identify when a stack frame's bounds are overrun/underrun. In this case, each stack segment represents one stack frame. The four steps in dealing with stack segments are as follows.

1. When a function is entered, function arguments are above the stack pointer (the x86 stack grows towards zero), the return address is at the stack word pointed to by the stack pointer, and local variables used by the function will be placed below the stack pointer. If the analysis knows the size and number of arguments, it can create a segment-type X for the stack frame, which is bounded at the high end by the last function argument, and is unbounded at the low end. This segment-type can be stored in a stack of segment-types.

2. The values within the segment should be shadowed with n when the segment-type is created, as they are uninitialised and should not be used as pointers. The exception is the function arguments, whose shadow values have already been initialised and should be left untouched.

3. The only pointer to a new stack segment is the stack pointer; its shadow is set to $p(X)$, where X is the segment-type just created.

4. When the function is exited, the segment-type can be removed from the segment-type stack, and marked as deallocated.

These segments allow the analysis to detect two kinds of errors. First, any access using the stack pointer (or a register assigned the stack pointer's value, such as the frame pointer) that exceeds the high end of the stack frame will be detected. Second, and more usefully, any dangling pointers to old stack frames will be caught, even if new stack frames have been built that cover the address range of the old stack frames; this is possible because the segment associated with the dangling pointer will be marked as having been freed. (By comparison, Memcheck and Addrcheck will only detect the use of such dangling pointers while the pointed-to memory remains unaddressable, i.e. not if the stack grows past that point again.) Such dangling pointers can occur if the address of a stack variable is saved in a global variable, or returned from the function. Even better, it will detect any use of dangling pointers in multi-threaded programs that have multiple threads sharing stacks. Such bugs can be extremely difficult to track down.

Knowing the size and number of function arguments requires access to the debug information. If the debug information is missing, an approximation can be used: instead of a segment that is bounded at one end, use an unbounded segment. The first kind of error mentioned above, violations of the stack's edge, will not be detected. However, the second error type mentioned, that of dangling pointer use, will be. This is because any access to a freed segment triggers an error, regardless of whether that access is in range. This approximation could also be used for stack frames of functions like `printf()` that have a variable number of arguments.

As with heap blocks and static memory, the analysis could track individual variables within stack frames, but it would be very fiddly.

All this assumes function entries and exits are paired nicely, and can be identified. Section 4.4.4 discusses what happens in practice.

4.3.8 Shadow Computation Operations

The analysis uses shadow computation, as described in Section 2.4.4. The following paragraphs give the details of the three kinds of metavalue propagation.

Copying Metavalues For machine instructions that merely copy existing values around, the metadata is correspondingly copied. Note that $p(X)$ metavalues must be copied by reference rather than by value, so that if the segment pointed to, X, is freed, all $p(X)$ types that point to it see its change in status.

New Static Metavalues Machine instructions that mention constants effectively introduce new values. The type of each static value is found in the same way that the types of values in newly loaded static memory are given, as was described in Section 4.3.6; this will be n, u, or the appropriate pointer-type.

New Dynamic Metavalues Figure 4.1 shows the shadow operations for several common binary operations; the type of the first operand is shown in the leftmost column, and the type of the second operand is shown in the top row.

The first operation is addition, shown in Table 4.1(a). Adding two non-pointers results in a non-pointer. Adding a non-pointer to a pointer results in a pointer of the same segment; this is crucial for handling pointer arithmetic and array indexing. Adding two pointers together produces a non-pointer, and an error message is issued; while not actually incorrect, it is such a dubious (poorly typed) operation that it is worth flagging. Finally, if either of the arguments are unknown, the result is unknown. Thus unknown values tend to "taint" known values, which could lead to a large loss of

$+$	n	u	$p(Y)$
n	n	u	$p(Y)$
u	u	u	u
$p(X)$	$p(X)$	u	n*

(a) Add

$\times$	n	u	$p(Y)$
n	n	n	n
u	n	n	n
$p(X)$	n	n	n*

(b) Multiply

&	n	u	$p(Y)$
n	n	u	$p(Y)$
u	u	u	u
$p(X)$	$p(X)$	u	n*

(c) Bitwise-and

^	n	u	$p(Y)$
n	u	u	u
u	u	u	u
$p(X)$	u	u	u

(d) Bitwise-xor

$-$	n	u	$p(Y)$
n	n	u	n*
u	u	u	u
$p(X)$	$p(X)$	u	$n/?$

(e) Subtract

'*': An error message is issued when this happens.
All u results are range tested and possibly converted to n.

Table 4.1: Basic shadow operations

accuracy quite quickly. However, before the metadata operation takes place, the analysis performs the real operation and checks its result. As it does for static values, the analysis uses a range test, and if the result is clearly a non-pointer the type given is n. This is very important to prevent unknown-ness from spreading too much.

Multiplication, shown in Table 4.1(b), is simpler; the result is always a non-pointer, and an error message is issued if two pointers are multiplied. The analysis could issue error messages if a pointer was multiplied by a non-pointer (early versions of Annelid did this), but in practice this occasionally happens legitimately; similarly, division sometimes occurs on pointers, e.g. when putting pointers through a hash function. Several times I had to stop issuing error messages on pointer arithmetic operations that seemed ridiculous, because real programs occasionally do them. Generally, this should not be a problem because the result is always marked as n, and any subsequent use of the result to access memory will be flagged as an error.

Bitwise-and, shown in Table 4.1(c), is more subtle. If a non-pointer is bitwise-and'd with a pointer, the result can be a non-pointer or a pointer, depending on the non-pointer value. For example, if the non-pointer has value `0xfffffff0`, the operation is probably finding some kind of base value, and the result is a pointer. If the non-pointer has value `0x000000ff`, the operation is probably finding some kind of offset, and the result is a non-pointer. The analysis deals with these possibilities by assuming the result is a pointer, but also doing the range test on the result and converting it to n if necessary. The resulting shadow operation is thus the same as that for addition.

For bitwise-xor, shown in Table 4.1(d), the analysis does not try anything tricky; it simply uses a range test to choose either u or n. This is because there are not any sensible ways to *transform* a pointer with bitwise-xor. However, there are two cases where bitwise-xor could be used in a non-transformative way. First, the following C code swaps two pointers using bitwise-xor.

```
p1 ^= p2;
```

```
p2 ^= p1;
p1 ^= p2;
```

Second, in some implementations of doubly-linked lists, the forward and backward pointers for a node are bitwise-xor'd together, to save space.[1] In both these cases, the pointer information will be lost, and the recovered pointers will end up with the type u, and thus not be checked.

Most other operations are straightforward. Increment and decrement are treated like addition of a non-pointer to a value, except no range test is performed. Address computations (on x86, using the `lea` instruction) are treated like additions. Shift/rotate operations give a n or u result, depending on the result value—the analysis does not simply set the result to n just in case a pointer is rotated one way, and then back to the original value. Negation and bitwise-not give an n result.

Subtraction The remaining operation is subtraction. It is somewhat similar to addition, and is shown in Table 4.1(e). Subtracting two non-pointers gives a non-pointer; subtracting a non-pointer from a pointer gives a pointer; subtracting a pointer from a non-pointer is considered to be an error.

The big complication is that subtracting one pointer from another is legitimate, and the result is a non-pointer. If the two pointers involved in the subtraction point to the same segment, there is no problem. However consider the following C code.

```
char p1[10];
char p2[10];
int diff = p2 - p1;
p1[diff] = 0;
```

This uses the pointer `p1` to access the array pointed to by `p2`, which is a different segment. ISO C forbids such inter-array pointer subtraction, but in practice it does occur in real C code because it works on most machines. Also, it may be valid in other programming languages, and if the analysis is to be language-independent, it must handle this case.

The problem is that the addition of a non-pointer with a pointer can result in a pointer to a *different segment*, if the non-pointer is the result of a pointer difference. The most accurate way to handle this is to generalise the type n to $n(X, Y)$, which is like n for all operations except addition and subtraction, in which case the analysis requires that $p(X)+n(X, Y)$ gives $p(Y)$, $p(X)-n(Y, X)$ gives $p(Y)$, and so on. Also, pointer differences should be added transitively, so that $n(X, Y)+n(Y, Z)$ gives $n(X, Z)$. However, this is a complex solution for a problem that does not occur very often. Section 4.4.6 describes how Annelid handles this case in practice.

Propagation of Unknown It is tempting to be less rigorous with preserving u types. For example, one might try making the result of $u + p(X)$ be $p(X)$. After all, if the u operand is really a non-pointer, the result is appropriate, and if the u operand is really a pointer, the result will undoubtedly

[1] The pointers can be recovered by using the bitwise-xor'd values from the adjacent nodes; their pointers can be recovered from their adjacent nodes, and so on, all the way back to the first or last node, which holds one pointer bitwise-xor'd with `NULL`.

be well outside the range of $p(X)$, so any memory accesses through it will be erroneous, and should be caught. By comparison, strict u-preservation will cause the analysis to miss this error.

At first, Annelid did this, being as aggressive as possible, and assuming unknowns are pointers when possible. However, in practice it causes false positives, usually from obscure sequences of instructions that would fool the shadow operations into assigning the wrong segment-type to a pointer, e.g. assigning a heap segment-type to a stack pointer.

Each time such a false positive occurred, I reduced the aggressiveness; after it happened several times, it became clear that trying to second-guess what was happening was not the best way to proceed. Instead, I switched to being more conservative with u values, and using range tests throughout.

There is no way of handling these shadow operations that is clearly the best. The required approach is to try alternatives and find one that strikes a good balance between finding real bounds-errors and avoiding false positives.

4.4 How Annelid Works: Implementation

This section describes how Annelid works, emphasising in particular how the implementation differs from the idealised design presented in Section 4.3.

4.4.1 Metadata Representation

The four kinds of metadata are represented as follows.

- A segment-type X is represented by a dynamically allocated *segment structure* containing its base address, size, a pointer to an "execution context" (a stack-trace from the time the segment is allocated, which is updated again when the segment is freed), and a tag indicating which part of memory the segment is in (heap, stack, or static) and its status (in-use or freed). Each structure is 16 bytes. Execution contexts are handled by Valgrind's core; they are stored separately, in such a way that no single context is stored more than once, because repeated contexts are very common. Each execution context holds a stack trace of depth four by default, although this can be changed via a command line option.

- A non-pointer-type n is represented by a small constant `NONPTR`.

- An unknown-type u is represented by a small constant `UNKNOWN`.

- A pointer-type $p(X)$ is represented by a pointer to the segment structure representing the segment-type X. Pointer-types are easily distinguishable from non-pointer-types and unknown-types because the segment structures never have addresses so small as `NONPTR` and `UNKNOWN`.

Each register and word of memory is shadowed by a shadow word that holds `NONPTR`, `UNKNOWN`, or a pointer to a segment structure. Just like Memcheck, shadow memory is set up up lazily, in 64KB chunks.

Sub-word writes to registers and memory destroy any pointer information in those words; the type for that word becomes either n or u, depending on a range test. Thus, any byte-by-byte copying of pointers to or from memory will cause that information to be lost. Fortunately, `glibc`'s implementation of `memcpy()` does word-by-word copying. If it did not, Annelid could just use Valgrind's support for function replacement to override `memcpy()` with its own version that did not use byte-by-byte copying.

Similarly, metadata is not stored across word boundaries in memory. With respect to metadata, an unaligned word-sized write is handled as two sub-word writes; therefore any pointer information will be lost, and the two aligned memory words partially written to will be set to `NONPTR` or `UNKNOWN`, depending on range tests. Fortunately, compilers avoid unaligned writes as much as possible because they are usually slow on modern machines, so this does not come up very often.

4.4.2 Segment Structure Management

Segment structure storage is straightforward. Segment structures are stored in a skip list [92], which gives amortised $\log n$ insertion, look-up, and deletion, and is much simpler to implement than a balanced binary tree.

The freeing of segment structures is much more complicated. When a memory block, such as a heap block, is freed, the segment structure for that block is retained, but marked as being freed. In this way, Annelid can detect any accesses to freed blocks via dangling pointers. However, Annelid needs to eventually free segment structures representing freed segments, otherwise it will have a space leak. There are two possible ways to do this.

The first way is to use garbage collection to determine which segment structures are still live.[2] Unfortunately, the root set is extremely large, consisting of all the shadow registers and all of shadow memory. Therefore, collection pauses could be significant.

The second way is to use segment structure recycling. Annelid would start by allocating each new segment structure as necessary, and then placing it in a queue when its segment is freed. Once the queue reaches a certain size (e.g. 1,000 segments), Annelid can start recycling the oldest structure segments. If the queue size drops to the threshold value, Annelid would re-start allocating new segments structures until it grew bigger again.

Thus Annelid would maintain a window of tracked freed-segments. It is possible that a segment structure could be recycled, and then memory could be accessed through a danging pointer that points to the freed segment. In this case, the error message produced will mention the wrong segment. Or, in very unlucky cases, the pointer will be within the range of the new segment and the error will be missed. The chance of this last event happening can be reduced by ensuring old freed segments are only recycled into new segments with non-overlapping ranges.

In practice, since accesses through dangling pointers are not that common, with a suitably large threshold on the freed-queue this should happen extremely rarely. Also, recycled segments could be marked, and error messages arising from them could include a disclaimer that there is a small chance that the range given is wrong.

Alternatively, since the $p(X)$ representation is a pointer to a segment structure, which is word-

[2] This is a simplified garbage collection, as the traversals will only be one level deep, since segment structures cannot point to other segment structures.

aligned, there are two bits available in the pointer which could be used to store a small generation number. If the generation number in the pointer does not match the generation number in the segment structure itself, an error message can be issued.

One other characteristic of recycling is worth mentioning. If the program being checked contains memory leaks, the segments shadowing the leaked heap blocks will also be leaked and lost by Annelid. This would not happen with garbage collection.

So the trade-off between the two approaches is basically that garbage collection could introduce pauses, whereas recycling has a small chance of causing incorrect error messages. Currently Annelid uses recycling, which was easier to implement.

4.4.3 Static Segments

Section 4.3.6 described how being too aggressive in identifying static array pointers can lead to false positives, e.g. when dealing with array accesses like `a[i-1]`. By default Annelid uses aggressive static pointer identification, but a command line option can be used to fall back to conservative identification.

4.4.4 Stack Segments

Section 4.3.7 discussed how stack segments could be handled by the analysis. There would be no problem if Annelid knew exactly when stack frames were built and torn down; more precisely, if Annelid knew when functions were entered and exited. Unfortunately, in practice this is quite difficult. This is because functions are not always entered using the `call` instruction, and they are not always exited using the `ret` instruction; some programs do unusual things with jumps to achieve the same effects. However, unusually enough, tail recursion will not really cause problems, as long as the function returns normally once it completes. The recursive tail calls will not be identified, and it will be as if every recursive function invocation is part of a single function invocation. In contrast, non-recursive tail calls may cause problems because the size of the new stack frame may differ from the size of the old frame.

It might not be a problem if some entry/exit pairs were not detected; then multiple stack frames could be treated by Annelid as a single frame. This could cause some errors to be missed, but the basic analysis would still work. However, even detecting matching function entries with exits is difficult. The main reason is the use of `longjmp()` and similar techniques which allow a program to effectively exit any number of function calls with a single jump. The obvious way to store stack segments is in a stack data structure. With `longjmp()`, it becomes necessary to sometimes pop (and mark as freed) multiple segments from this segment stack. If calls to `longjmp()` could be reliably spotted, this would not be a problem. However, GCC has a built-in non-function version of `longjmp()` that cannot be reliably spotted. Since Annelid cannot even tell when a `longjmp()` has occurred, it does not know when to pop multiple frames. If Annelid misses the destruction of any stack frames, it will fail to mark their segment-types as freed, and thus not spot any erroneous accesses to them via dangling pointers.

Stack-switching also causes big problems. If a program being checked switches stacks then Annelid should switch segment stacks accordingly. But detecting stack switches by only looking at the dynamic instruction stream is difficult, since it is often hard to distinguish a stack switch from a large stack allocation.

I experimented with various heuristics in an attempt to overcome these problems. For example, after a `longjmp()` occurs, the stack pointer will have passed several old frames in a single step. This evidence can be used to determine that these frames are now dead. However, I have not managed to find any heuristics robust enough to deal with all the difficulties. As a result, Annelid currently does not track stack segments at all. This is a shame, as detecting stack errors was one of the motivations for building Annelid.

4.4.5 Range Tests

As mentioned in Section 4.3.8, the analysis can convert many u result types to n if they are definitely not pointers. In Annelid this test succeeds if a result value is less than `0x01000000`, or greater than `0xff000000`.

This is slightly risky, as a (strange) program could use `mmap()` to map a file or create a memory segment below `0x01000000`. Since Valgrind has complete control over memory allocation, Annelid could ensure this never happens. Alternatively, Annelid could track the lowest and highest addressable addresses, and declare any value outside this range as a non-pointer (with a suitable safety margin).

4.4.6 Pointer Differences

Section 4.3.8 suggested handling pointer differences between different segments precisely by generalising the n type to a $n(X, Y)$ type. This approach turned out to be very painful. It made the addition and subtraction operations more complex; also, pointer differences are often scaled, so $n(X, Y)$ types would have to be propagated on multiplication, division, and shifting. Finally, having to free $n(X, Y)$ structures complicated segment structure storage.

A much easier solution was to introduce a new run-time type b (short for "bottom"). Any value of type b is not checked when used as a pointer for a load or store, and the result of any type operations involving b as an operand is b. Values of type b are never changed to type n via a range test.

This solution is simple but blunt. Fortunately it is not needed often; importantly, intra-segment pointer differences (within a single array), which are more common, are handled accurately.

4.4.7 System Calls

Valgrind does not trace into the OS kernel. Annelid uses the callbacks provided by the core (Section 2.4.2) to do normal range checks on memory ranges that are read and/or written by system calls, and so can find bounds errors in system call arguments.

Most system calls return an integer error code or zero; for these Annelid sets the return type to n. Some system calls can produce pointers on success, notably `mmap()` and `mremap()`. Annelid's current approach is to give these results the value u. I originally tried tracking segments returned by `mmap()` like other segments, but abandoned this because they are difficult to deal with, since they are often resized, and can be truncated or split by other calls to `mmap()` that overlap the range. This should be no great loss, as programs tend to use `mmap()` in straightforward ways, and overruns of mapped segments should be rare.

4.4.8 Custom Allocators

Annelid handles the standard allocators called via `malloc()` and friends. Custom allocators can be handled with a small amount of effort, by inserting client requests into the program being checked. These are macros that pass information to Annelid about the size and location of allocated and deallocated blocks. They are just like the ones Memcheck uses (see Section 2.4.5).

4.4.9 Leniency

Some common programming practices cause bounds to be exceeded. Most notably, `glibc` has heavily optimised versions of functions like `memcpy()`, which read arrays one aligned word at a time. On 32-bit x86 machines, these functions can read up to three bytes past the end of an array. In practice, this does not cause problems. Therefore, by default Annelid allows aligned, word-sized reads to exceed bounds by up to three bytes, although there is a command line option to turn on stricter checking that flags these as errors.

4.4.10 Performance

Performance figures for Annelid on the SPEC CPU2000 suite were given in Table 2.3. The slow-downs were in the range 15.0–90.3, with 38.1 as the median. As mentioned, this analysis is heavy-weight. Therefore, the overhead is high and programs run much slower than normal. However, the slow-down experienced is not dissimilar to that with many thorough memory checking DBA tools. Also, Annelid's analysis code has not been optimised very much, so there is room for improvement. The code expansion factor (see Table 2.4) was 10.5–15.0, with a median of 12.2.

4.4.11 Real World Results

So does Annelid actually find useful bugs in real programs? It is not easy to say. It finds all the deliberate bugs seeded in test programs that I expect; and the test suite is quite thorough, exercising all shadow arithmetic operations and all kinds of bounds errors. However, writing one's own test cases is not the best way to test a program, so this evidence is not hugely convincing.

Working up to real programs is not easy. Analysing the SPEC CPU2000 benchmarks, Annelid found some accesses to already-freed segments in `vortex`; these bugs are also found by Memcheck. False positives occurred for `gap` and `gcc` when aggressive static pointer detection was on, due to the use of negative array indices explained in Section 4.3.6. No other bugs were found.

If a real program is run under Annelid, and Annelid reports no errors, it is often difficult to know if Annelid has missed any. One can deliberately seed errors that it should find in these larger programs, and see if Annelid finds them. I have done with with several of the SPEC CPU2000 benchmarks, and

Annelid detected the errors as expected. However, this is also not particularly convincing evidence.

Looking at Annelid's internals as it runs, although around 99% of the heap values are `UNKNOWN`, typically more than half of non-stack accesses are checked, so there is some evidence that unknowns are not tainting everything, which is encouraging.

4.4.12 Crucial Features

Annelid's analysis is very heavyweight. It was very much designed to be implemented as a Valgrind tool, and it was reasonably easy to do so. As Table 2.2 showed, Annelid is implemented in 3,727 lines of C code, including blank lines and comments.

The most crucial Valgrind features that made Annelid easy to build were the same as those that were crucial were for Memcheck (Section 2.4.5)—those supporting shadow computation, such as support for pervasive, interconnected analysis code, and support for location metadata. Without this support, Annelid would have been much harder to write. Other features useful for Memcheck were also useful for Annelid, particularly the ability to replace functions (especially `malloc()` and friends), memory event callbacks, and error recording.

One area where Valgrind fell short was the reading of debug information. The core can read filenames, function names and line numbers, but ignores other debug information. Annelid needed extra type information in order to handle static arrays well (see Section 4.3.6), and so Valgrind's core required modification to read this extra debug information.

Another area where Valgrind fell short was with its system call wrappers. The wrappers can tell a tool plug-in what pointer values were used to read or write memory blocks within system calls, but they do not say what memory location or register those pointer values were held in, which meant that the run-time type of the pointers was not available. Valgrind's core again required modification to provide this extra information.

4.5 Shortcomings

Like all error-checking analyses, Annelid's is far from perfect. How well it does depends on the circumstances. Happily, it exhibits "graceful degradation"; as the situation becomes less favourable (e.g. debug information is not present), more and more $p(X)$ metavalues will be lost and seen instead as u. Thus it will detect fewer errors, but will not give more false positives.

4.5.1 Optimal Case

In the best case, the program will have all debug information and symbol tables present. In that case, even if the design was implemented optimally, the analysis would have problems in the following cases.

- Certain operations, such as swapping pointers with the bitwise-xor trick, cause $p(X)$ metavalues to be downgraded to u; erroneous accesses using those pointers will then not be caught. One could imagine beefing up the type system to handle such cases, but the cost/benefit ratio would be very high.

- Directly out-of-bounds accesses to static and stack data objects (e.g. accessing `a[10]` in an array of ten elements) cannot be detected if they happen to fall within a nearby data object. Also, constant pointers that do not point to the start of arrays must either be ignored (i.e. marked as unknown values), potentially missing errors, or handled, potentially causing false positives by incorrectly identifying which array they point to.

- C-style unions must be handled conservatively, by using a segment of the largest possible size. This can cause errors to be missed.

4.5.2 Implementation

Annelid suffers from a few more shortcomings, mostly because the design was too difficult to implement fully.

- Pointer-types are lost if pointers are written unaligned to memory.

- Likewise, pointer-types are lost if pointers are copied byte-by-byte between registers and/or memory words. (As Section 4.4.1 mentioned, `glibc`'s implementation of `memcpy()` does word-by-word copying, so this shortcoming does not affect it.)

- The use of b for inter-segment pointer differences will cause some errors to be missed.

- Annelid uses debug information about types only in simple ways. For example, it does not try to break up heap blocks into sub-segments. Also, the only static data objects it constructs segment structures for are arrays (not for structs or basic data types).

- Stack segments are not handled at all.

4.5.3 No Debug Information

If debug information is not present, no static checking can be performed, as Annelid cannot recognise pointers to static data objects (although it could check the bounds of entire static data segments, since that information is known without debug information). Also, if Annelid checked stack frames, it would have to fall back to using unlimited-size segments, as discussed in Section 4.3.7.

4.5.4 No Symbols

If a program has had its symbol tables stripped, error checking might degrade further. This is because, if Annelid did stack checking, it would rely on symbols being present for detecting function entry and exit points, as Section 2.4.3 explained.

4.5.5 Avoiding Shortcomings

A lot of the shortcomings arise because the information available in a program binary at run-time is less than that present in the original program. Debug information and symbol tables retain some of this information, but there are still some things that Annelid would like to know about. The obvious way to improve the situation is to involve the compiler producing the programs; a lot of checking can be done purely dynamically, but some things clearly require static help.

First, a compiler could promise, in certain circumstances, to avoid generating code that causes problems for Annelid. For example, it could ensure that all pointer reads and writes are aligned, and it could ensure where possible that array accesses are done via the array's base pointer, rather than pre-computing offsets.

Second, a compiler could use client requests (see Section 2.3.12) to embed extra information into programs for Annelid to use. This information might indicate that a particular memory access is intended to be to a particular segment, or might indicate when a `longjmp()` occurs. Similarly, users could manually embed information via client requests, although this option is only really practical for events that are rare in a program's source code, such as stack switches.

Finally, some bounds errors that Annelid cannot find should arguably be found by a compiler. In particular, directly out-of-bounds accesses to static and stack arrays (e.g. accessing element `a[10]` of a ten-element array) could easily be found by the compiler.

Annelid's analysis has some very nice characteristics, but it clearly also has some significant shortcomings. Getting cooperation from a compiler could help. However, if compiler cooperation is being used, one might as well combine Annelid's analysis with one of the source analyses described in Section 4.6.3. Such a hybrid tool could do accurate source analysis on the parts of the program instrumented appropriately, and fall back on a dynamic binary analysis like Annelid's for the rest of the program.

4.6 Related Work

This section describes several tools that find bounds errors for C and C++ programs, and compares them to Annelid. No single checking analysis is best; each has its strengths and weaknesses, and they complement each other. Each subsection describes tools that use a particular analysis.

Note that many tools have been written that perform various kinds of memory checking, particularly commercial ones. This section describes a representative sample, but by no means lists all of them.

4.6.1 Red-zones

Many bounds-checking tools dynamically checks accesses to objects on the heap. The simplest approach is to replace the standard versions of `malloc()`, `new`, and `new[]` with versions that produce heap blocks with a few bytes of padding at their ends, called *red-zones*. These red-zones are filled with a distinctive values, and should never be accessed by a program. When the heap block is freed with `free()`, `delete` or `delete[]`, the red-zones are checked, and if they have been written to, an error message is issued. The documentation for mpatrol [97] lists many tools that use this technique, which is very simple, but has the following shortcomings.

1. It only detects small overruns/underruns, within the red-zones—larger overruns or completely wild accesses could access the middle of another heap block, or non-heap memory.

2. It only detect writes that exceed bounds, not reads.

3. It only reports errors when a heap block is freed, giving no information about where the overrun/underrun occurred.

4. It does not detect accesses to freed heap blocks via dangling pointers, unless they happen to hit another block's red-zone (even then, identifying the problem from the error message will be difficult).

5. It does not work with heap blocks allocated with custom allocators (although the technique can be built into custom allocators).

6. It only works with heap blocks—stack and static blocks are pre-allocated by the compiler, and so red-zones cannot (without great difficulty) be used for them.

This technique has too many problems to be really useful. All these problems are avoided by analyses that track pointer bounds, such as Annelid's.

Electric Fence [87] uses a slightly different `malloc()` replacement that uses entire virtual pages as red-zones. These pages are marked as inaccessible, so that any overruns/underruns cause the program to abort immediately, whereupon the offending instruction can be found using a debugger. This avoids problems 2 and 3 above, and mitigates problem 1 (because the red-zones are so big). However, it increases virtual memory usage massively, making it impractical for use with large programs.

A better approach is used by Memcheck (Section 2.4.1) and Purify [52]. They too replace `malloc()` *et al* with versions that produce red-zones, but they also maintain addressability metadata about each byte of memory, and check this metadata before all loads and stores. Because the red-zones are marked as inaccessible, all heap overruns/underruns within the red-zones are spotted immediately, avoiding problems 2 and 3 above. If the freeing of heap blocks is delayed, this can mitigate problem 4. These tools also provide hooks that a custom allocator can use to tell them when new memory is allocated, alleviating problem 5.

Purify is also capable of inserting red-zones around static variables in a pre-link step, in certain circumstances, as the Purify manual explains:

> "Purify inserts guard zones into the data section only if all data references are to known data variables. If Purify finds a data reference that is relative to the start of the data section as opposed to a known data variable, Purify is unable to determine which variable the reference involves. In this case, Purify inserts guard zones at the beginning and end of the data section only, not between data variables."

Similarly, the Solaris implementation of Purify can also insert red-zones at the base of each new stack frame, and so detect overruns into the parent frame. It is unclear exactly how Purify does this, but it may be that the way the SPARC stack is handled makes it much easier to do than on x86.

Red-zones and addressability tracking works very well, which accounts for the widespread use of Memcheck and Purify. However, the remaining shortcomings—1 and particularly 6 (even with Purify's partial solution)—are important enough that tools tracking pointer bounds are worth having.

4.6.2 Fat Pointers

Another major group of bounds-checking analyses use fat pointers, where each normal pointer is augmented with bounds metadata, typically the minimum and maximum address it can be used to access. The standard approach is to use a dynamic source analysis implemented using static source instrumentation, where the analysis code is added by a compiler or pre-processor. All accesses through fat pointers are checked, which gives very thorough checking, and avoids all the problems of red-zones described in Section 4.6.1. But there are other significant disadvantages.

In earlier implementations (e.g. [8]), every pointer was replaced with a struct containing the pointer, plus the bounds metadata. This change in pointer representation has two major problems.

1. Changing the size of a fundamental data type will break any code that relies on the size of pointers, for example, code that casts pointers to integers or vice versa, or C code that does not have accurate function prototypes.

2. Support may be required in not only the compiler, but also the linker (some pointer bounds cannot be known by the compiler), and possibly debuggers (if the fat pointers are to be treated transparently).

Jones and Kelly describe a better implementation in [60]. It avoids both problems by storing each pointer's metadata separately from the pointer itself, and thus largely preserves backward compatibility with existing programs. It cannot handle programs that move pointers outside array bounds and then back in again, so small numbers of source code changes are often needed in practice. It is available [108] as patches for GCC. CRED [98] is an extension of these patches that can handle out-of-bounds pointers, and can be made to run faster by only checking string operations. Patil and Fischer [86] also store metadata separately, in order to perform the checking operations on a second processor.

The improved approach still has the following problems due to the instrumentation being done statically and at source.

1. It requires compiler support, or a pre-compilation transformation step.

2. All code must be instrumented to use fat pointers, including libraries, which can be an enormous hassle. Alternatively, parts of the program can be left uninstrumented, so long as interface code is produced that converts fat pointers to normal pointers and vice versa when moving between the two kinds of code. Producing this code requires a lot of work, as there are many libraries

> used by normal programs. If this work is done, two kinds of errors can still be missed. First, pointers produced by the library code may lack the bounds metadata and thus not be checked when they are used in the "fat" code. Second, library code will not check the bounds data of fat pointers when performing accesses.

Annelid's analysis uses the same basic idea of tracking a pointer's bounds, but the implementation is entirely different—instead of a dynamic source analysis implemented via static source instrumentation, Annelid uses DBA implemented via DBI. As Sections 1.1.2 and 1.1.4 noted, this means it has various inherent pros and cons. The pros are: it is language-independent, it does not require source code, the client does not require any special preparation, and it covers all code in a program and so handles libraries without difficulty. Also, because Annelid tracks both pointers and pointees it can detect the use of dangling pointers, unlike the Jones/Kelly approach. It can also detect some run-time type errors. The cons are: it is platform-specific, and has lower-level information to work with than source-based methods, which reduces accuracy. The overhead is also greater than that of the source-based analyses.

4.6.3 Mixed Static and Dynamic Analysis

CCured [78] is a tool for making C programs type-safe, implemented as a source code transformer. It does a sophisticated static analysis of C source code, then adds bounds metadata and inserts dynamic bounds-checks for any pointers for which it cannot prove correctness statically. It suffers from all the problems described in Section 4.6.2, because it involves static analysis and changes the size of pointers. These additional checks slow performance; published figures range from 10–150%. Also, on larger programs, one "has to hold CCured's hand a bit" for it to work; getting such programs to work can apparently take several days' work [77]. Again, non-coverage of library code is a problem.

Compuware's BoundsChecker Professional tool [34] inserts memory checks into source code at compile-time. It seems to be similar to CCured, but without the clever static analysis to avoid unnecessary checks, and so can suffer much larger slow-downs.

By comparison, the analysis described in this chapter is entirely dynamic, and does not require any recompilation or source code modification. It does find fewer errors, though.

4.6.4 Static Analysis

Some tools detect bounds errors via static source analysis. Purely static tools are good because they consider all paths, and can be sound, but many memory bugs cannot be detected by static analysis because many memory bounds are not computable statically.

One example static analysis tool is ARCHER [120], which uses path-sensitive, interprocedural, bottom-up symbolic analysis. It uses a constraint solver to check every array access, pointer dereference, or call to a function expecting a size parameter. It is unsound but highly practical, and has been used to find tens or hundreds of bugs in large software systems such as the Linux and OpenBSD kernels. A similar tool is PREfix [22], which detects various kinds of errors, including memory errors.

4.6.5 Runtime Type Checking

Some tools perform run-time type checking that is similar to Annelid's analysis. Burrows *et al*'s tool Hobbes [21] performs a DBA that maintains a run-time type metavalue for every value in a program. It warns about run-time type violations, e.g. if two pointers are added. It can also detect bounds errors if they lead to run-time type errors. Implemented using DBI via an x86 binary interpreter, its slow-down factor is in the range 50–175. RTC [66] does a similar dynamic source analysis of C programs, using static source instrumentation to insert the analysis code. Slow-downs when using it are in the range 6–130. As well as type checks, both tools use heap block red-zones to find some overrun errors.

4.7 Conclusion

This chapter described Annelid, a bounds-checking tool, which was built with Valgrind. Annelid implements a classic checking analysis—bounds-checking—in a novel way, being the first tool to do fat pointer bounds-checking entirely dynamically. It can find bounds errors for heap and static memory reasonably well, although the implementation falls short of the idealised design in various ways, particularly in the lack of stack checking.

Perhaps the most important lesson to learn from Annelid is that accurate bounds-checking using fat pointers is a problem that is not really suited to a purely dynamic, binary-level approach. It seems that some source-level information is required to do bounds-checking well. Nonetheless, Annelid shows again the importance of Valgrind's support for location metadata and shadow computation, and the ideas involved in Annelid's DBA could one day form part of a better, future hybrid static/dynamic bounds-checking analysis.

CHAPTER 5

A Visualisation Tool

This chapter describes a novel visualisation tool, built with Valgrind, which uses location metadata and shadow computation.

5.1 Introduction

This chapter describes Redux, a data flow visualisation tool, which was built with Valgrind.

5.1.1 Program Comprehension Tools

The third and final major group of DBA tools, after profilers and checkers, are program comprehension tools. These tools collect information about how a program works that is not obvious from its source code. This information may be used by a programmer to improve performance or correctness of a program; or for other purposes such as decompiling a program, or comparing it to another. Despite being a large class of tools, very few program comprehension tools have been implemented using DBI frameworks.

5.1.2 Visualising Programs

One particular sub-class of comprehension tools are visualisation tools. That is, tools that produce some kind of diagram to communicate something about how a program works. There are a number of different ways to visualise a program, and a number of different levels at which it can be done. One thing that can be useful to visualise is a program's data flow.

5.1.3 Overview of Redux

Redux is a prototype tool that generates *dynamic data flow graphs* (DDFGs), which represent the entire computational history of a program. Redux generates these graphs by tracing a program's execution and recording the inputs and outputs of every value-producing operation (instructions and system calls) that takes place, building up a complete computational history of the program. A program's behaviour, as seen from the outside world, is entirely dictated by the system calls it makes during execution (ignoring timing issues).[1] Therefore, at program termination, by printing the parts of the graph reachable from system call inputs, Redux shows the computations that directly affect the program's behaviour.

Redux's DBA is novel because DDFGs are novel. DDFGs cut through all uninteresting book-keeping details, giving just the *essence* of a program's computation. This is shown by the fact that programs that compute the same thing in highly different ways have very similar or identical DDFGs, as later sections show. DDFGs can be used for debugging and program slicing, and have a range of other possible uses.

Redux gains the benefits of DBI shared by all Valgrind tools: it is extremely easy to run, requiring no recompilation; it naturally covers all code executed, including libraries; and it works with programs written in any language.

Redux was first described in [81]. Redux is currently a prototype tool, intended to provide a way of experimenting with DDFGs, and has only been used with small programs. It is not part of the Valgrind distribution.

5.1.4 Chapter Structure

This chapter is structured as follows. Section 5.2 shows an example of Redux's use and introduces DDFGs. Section 5.3 describes how Redux works. Section 5.4 shows how programs computing the same thing in very different ways can have similar or identical DDFGs. Section 5.5 discusses possible uses of Redux. Section 5.6 describes difficulties and problems with Redux. Section 5.7 discusses related work, and Section 5.8 concludes.

5.2 Using Redux

Redux is invoked from the command line like any other Valgrind tool. To visualise the data flow of a program `foo` with Redux, one would use the following command:

```
valgrind --tool=redux foo
```

The program then runs under Redux's control. Redux instruments every value-producing operation so that each time the operation is executed, a node recording the inputs and outputs is added to the growing DDFG.

When the program terminates, Redux prints out a text description of the portion of the constructed DDFG that is reachable from system call nodes. This output can then be passed to a graph-drawing program for rendering.

[1] I will use "behaviour" in this sense throughout this chapter.

5.2.1 Dynamic Data Flow Graphs

A dynamic data flow graph (DDFG) is a directed acyclic graph (N, E). N is a set of nodes representing value-producing operations (with multiple executions of the same operation represented distinctly). E is the set of edges denoting dynamic data flow between nodes; it also contains some edges denoting state dependencies that are required to accurately represent the program's behaviour.

5.2.2 Factorial

Figure 5.1 shows two C programs that compute the factorial of five—one iteratively, one recursively—and the resultant DDFGs produced by Redux. There are three node types shown. Those labelled "c.5L" and "c.1L" represent constants. The 'c' (short for "code") indicates that the constants appeared literally in the code; the letter "L" (short for "long") indicates that they have 32 bits. Operation nodes (e.g. "dec : 4L"; "dec" is short for "decrement") contain an operation name colon-separated from the operation's result; the operands can be seen from the input edges. The "_exit" node shows the program's exit code, as returned by the `_exit()` system call; system call nodes are shown with darker borders.

All value-producing operations—mostly arithmetic/logic operations and system calls—are represented with nodes. Other computations that do not produce values are not shown, such as loads from and stores to memory, register-to-register moves, and jumps and function calls (direct and indirect). Only inputs and outputs of nodes are shown; the locations of the inputs and outputs are not shown, nor are the address of the instructions that produced them. Also, no control flow is shown in the DDFG (it will be considered further in Section 5.6.3). Module boundaries are also irrelevant.

Importantly, the printed graphs shown in Figure 5.1 only show the nodes that were reachable from system call nodes. Although these programs performed many more value-producing operations than those shown, only those shown had a direct data flow effect on the program's behaviour; the rest were mere book-keeping.

Even from these two small examples, the informative nature of DDFGs is clear, as the differences in the data flow of the two programs is immediately evident. They would be extremely useful in explaining to someone the difference between iteration and recursion, for example.

5.2.3 Hello World

Figure 5.2 shows the Hello World C program and two versions of its DDFG. Consider first the larger DDFG on the left-hand side. This section will first explain the DDFG's components, and then analyse it, to see what it tells us about the program's execution. It will then consider the smaller DDFG.

Static constants are written with an "s." prefix. Dotted state dependency edges show the system call temporal order, which is preserved in the DDFG since many system calls have side-effects that affect the program's behaviour, and so they cannot be shown out of order. In contrast, the sequencing of the other operation nodes in the DDFG is unimportant, except that the obvious data flow constraints must be observed.

System call arguments are shown in the same way as the operands of arithmetic operations. How-

```
int faci(int n) {
   int i, ans = 1;
   for (i = n; i > 1; i--)
      ans = ans * i;
   return ans;
}

int main(void) {
   return faci(5);
}
```

```
int facr(int n) {
   if (n <= 1)
      return 1;
   else
      return n * facr(n-1);
}

int main(void) {
   return facr(5);
}
```

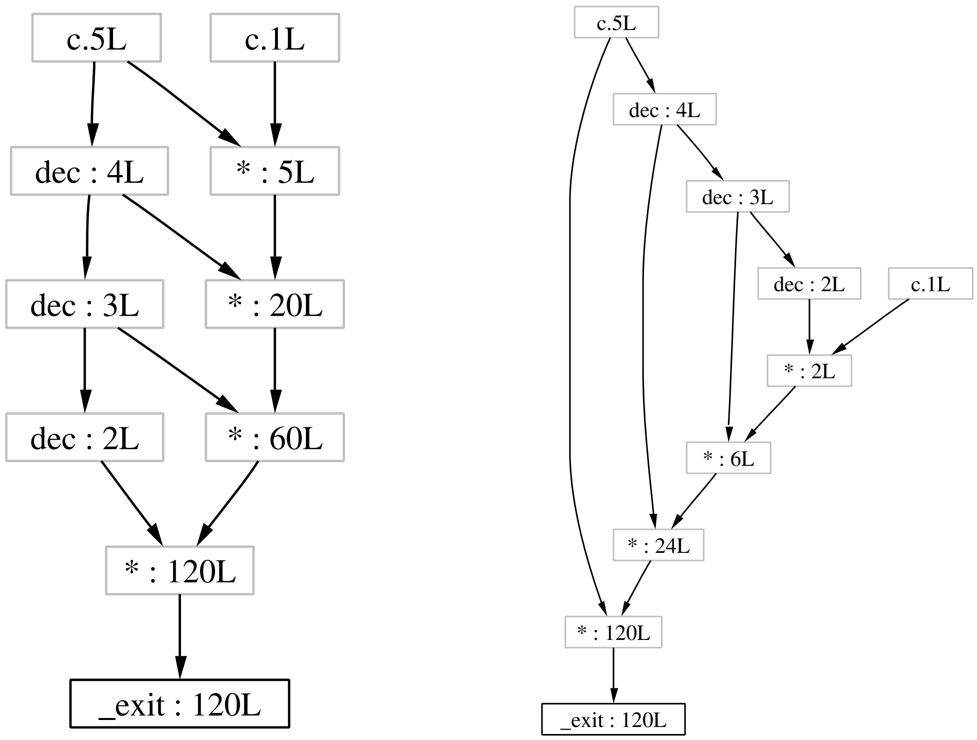
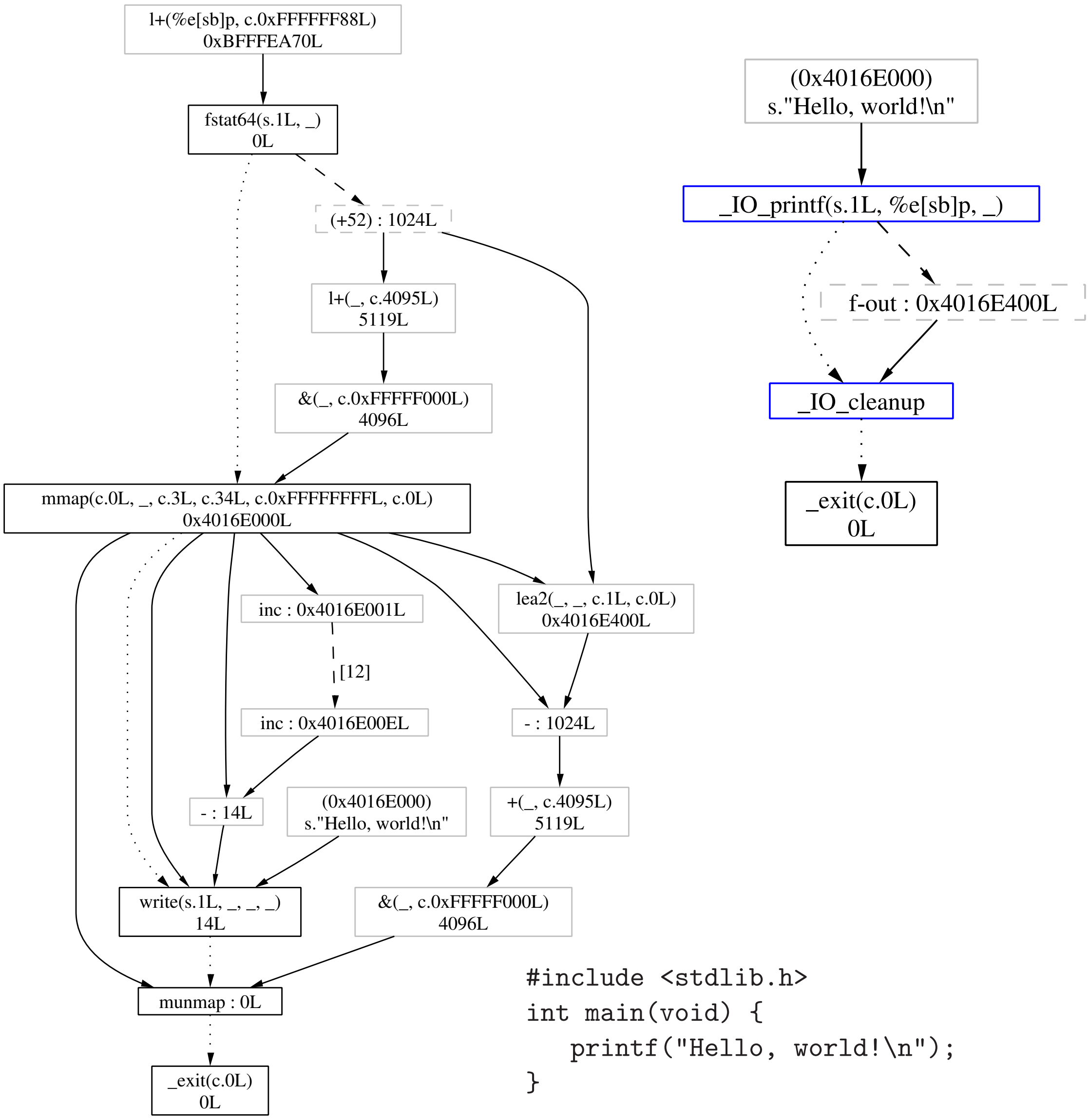


Figure 5.1: DDFGs for iterative and recursive `fac(5)` in C

ever, many system calls also have indirect arguments, accessed via direct pointer arguments; these *memory inputs* are important too. In this example, the `"Hello, world!\n"` memory input is shown as an extra argument to the `write()` system call. Its address is given in parentheses to show that it is paired with the direct pointer argument. System calls can also produce *memory outputs* via pointers. They are shown with dashed edges to dashed nodes, which contain the offset of the word within the memory block. The `fstat64()` call has a memory output; this is the `.st_blocksize` field 52 bytes into the output `struct stat64` buffer. Redux clearly has to know about system calls and their arguments. Redux knows itself the names of system calls and arguments; but the knowledge that some of these are memory input arguments comes from Valgrind, via callbacks that indicate when memory inputs are read (as Section 2.4.5 discussed).

The top node contains an inlined argument "%e[sb]p". By default, the Redux does not track the computational histories of the stack pointer (`%esp`) and frame pointer (`%ebp`). This is because




```
#include <stdlib.h>
int main(void) {
   printf("Hello, world!\n");
}
```

Figure 5.2: DDFGs for Hello World

the computations producing these pointers are usually long, uninteresting chains of small constant additions and subtractions. The “l+” nodes represent `lea` x86 instructions, which programs often use for integer addition.

Even for this tiny program, several measures have been taken to make the DDFG as compact and readable as possible: non-shared constants are inlined into their descendent nodes; a chain of fourteen increment (“inc”) nodes is abbreviated, with a dashed edge indicating the number of elided nodes; and the string argument to `write()` is pretty-printed in a compact way.

Now for the analysis. One can interpret the program’s operation quite precisely from the DDFG. The function `printf()` checks that standard output (file descriptor 1) is writable with the `fstat64()` system call. It then allocates a buffer with `mmap()` (at `0x4016E000`). The string `"Hello, world!\n"` is copied into the buffer, which is passed to `write()`. The other arguments to `write()` are 1 (standard output again) and the string length (14) which is the difference between the string's start and end addresses; the end address was found by incrementing a variable for each character copied into the buffer (shown by the abbreviated chain of "inc" nodes). Finally, the buffer is freed with `munmap()`, and `_exit()` terminates the program.

Finally, Redux can also create more abstract DDFGs, by using its `--fold` option to specify functions whose nodes and edges should be conflated. The right-hand side of Figure 5.2 shows Hello World's DDFG after `_IO_printf()` and `_IO_cleanup()` have been folded.[2] The function `_IO_cleanup()` is invoked by the C library after `main()` exits; the name of this function was found by looking through the `glibc` source code. Function output ("f-out") nodes are similar to system call memory output nodes—they show values produced by the function (unlike system call nodes, this includes its return value) that are subsequently used in a computation reachable from a system call argument.

5.3 How Redux Works

This section describes how Redux builds and prints DDFGs.

5.3.1 Overview

The basic idea is simple. As the program progresses, one node is added to the DDFG for each value-producing instruction executed. In addition, each register and memory word is shadowed by a pointer into the DDFG, which shows how it was computed. These pointers are location metadata, and they are propagated by shadow computation.

At termination, Redux draws only system call nodes, and those nodes in the DDFG that directly affected the system call arguments. These are all the nodes that directly affected the program's behaviour.

5.3.2 Metadata

Redux's main data structure is the DDFG itself, which grows as the client performs value-producing operations. Graphs are built data dependence-wise, i.e. each node points to its inputs. This is the natural way to do things, since an operation's operands are created before the operation takes place, and each operation has a fixed number of operands (except for system calls with memory inputs) whereas an operation result can be used any number of times. However, the graphs are drawn data flow-wise, i.e. each node points to the nodes that use its value. This is more intuitive to read.

Initially, the graph is empty, register shadows are initialised to point to a special "unknown register value" node, and each word of static memory has its shadow initialised (see Section 5.3.6 for details).

DDFG nodes hold a tag indicating their type (e.g. "+", "inc") and pointers to their operands. During execution, a new node is built for every value-producing operation executed. Using *sh*(`%reg`) to denote the shadow pointer for register `%reg`, the instruction:

[2] `_IO_printf()` is the name that appears in `glibc`'s symbol table, which Redux relies on to detect entry to the function; `printf()` is just an alias.

```
addl %eax, %ebx
```

will be instrumented to update the shadow registers as follows:

$$sh(\texttt{\%ebx}) := +(sh(\texttt{\%ebx}), sh(\texttt{\%eax}))$$

where +(a,b) represents a pointer to a "+" node with operands a and b. Instructions that only move existing values, such as moves, loads, and stores, are instrumented so that the shadows are copied appropriately. For example:

```
movl %ebx, %ecx
```

is instrumented to update the shadow registers as follows:

$$sh(\texttt{\%ecx}) := sh(\texttt{\%ebx})$$

Consider the following instructions.

```
1: movl $3,   %eax
2: movl $5,   %ebx
3: addl %eax, %ebx
4: incl %ebx
```

For instructions 1 and 2, two constant ('c') nodes are built; *sh*(`%eax`) and *sh*(`%ebx`) are set to point to them, as shown in Figure 5.3(a); the left-hand side shows the values of the registers and shadow registers, the right-hand side shows the created nodes. For instruction 3, a "+" node is built, with the two constant nodes as its operands, and *sh*(`%ebx`) is updated to point to it (Figure 5.3(b)). Each node stores the result of its operation, and the "+" node's result (8) is equal to the value in `%ebx`. This is an invariant: the result in a node pointed to *directly* by a register or memory word's shadow is always equal to the value of the register or memory word. For instruction 4, an "inc" node is built, its operand being the "+" node, and *sh*(`%ebx`) is updated again (Figure 5.3(c)). Again the invariant holds—`%ebx`'s value (9) matches the "inc" node's value, but no longer matches the "+" node's value, which *sh*(`%ebx`) now only indirectly points to.[3]

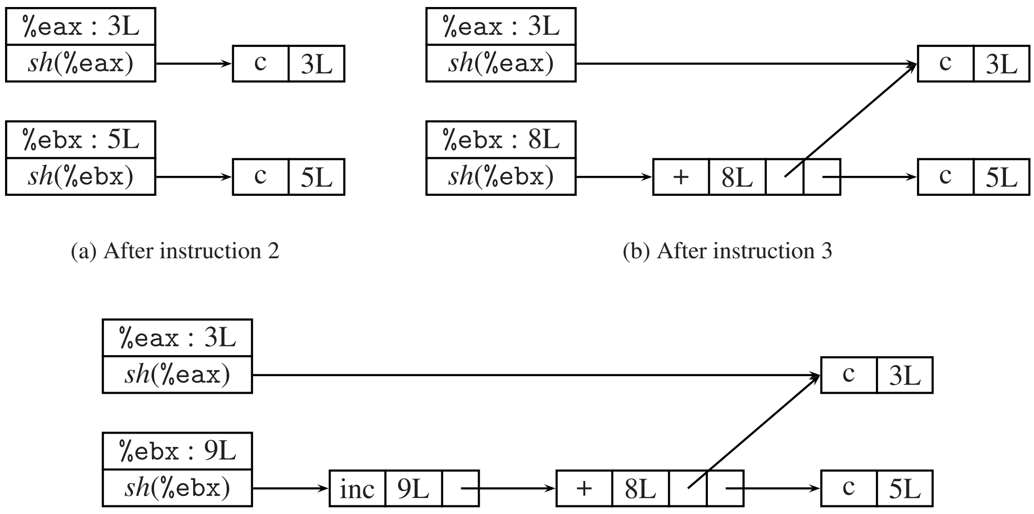
The shadows of deallocated memory words are set to point to a special "unknown memory" node. If these nodes appear in a program's graph, it very probably indicates a bug in the program.

5.3.3 System Calls

System calls are not handled very differently to arithmetic operations. When one takes place, the arguments (and their shadows) can be found from the registers (and shadow registers) easily.

Memory inputs are trickier, but not too difficult. Redux uses callbacks from the Valgrind core to find out when a system call reads memory. When this happens, Redux gathers the shadow words for the memory input block into an aggregate "chunk" node which is made an additional input to the

[3]The shadow computation is actually done at the level of UCode, rather than the level of the original x86 code, so this example is not quite representative. However, the ideas and results are the same.



(a) After instruction 2

(b) After instruction 3

(c) After instruction 4

Figure 5.3: Building a DDFG

system call node. Memory outputs are handled similarly—when the core tells Redux that a system call has written memory, Redux builds new system call memory output nodes for each word written by the call.

System call nodes in the DDFG are also recorded in a separate global data structure, so that they can be traversed at the end to dump the interesting parts of the DDFG to file. The exception is that system calls taking place before `main()` is entered are not included. This is because they are typically the same or very similar for every program, to do with setting up a program's environment, and are usually not interesting. This exception could be removed if necessary, though.

5.3.4 Sub-word Operations

Register and memory shadowing is done at per-word (32-bit) granularity. However, some instructions use one- or two-byte operands. This requires special "read byte" and "read word" nodes for extracting sub-words, and "split" nodes for combining them. Consider the instruction that moves the least-significant byte of register `%eax` into `%ebx`:

```
movb %al, %bl
```

It is instrumented to update the shadow registers as follows:

$$\mathit{sh}(\texttt{\%eax}) := \text{split4B}(B_0(\mathit{sh}(\texttt{\%eax})),\ B_1(\mathit{sh}(\texttt{\%eax})),\ B_2(\mathit{sh}(\texttt{\%eax})),\ B_3(\mathit{sh}(\texttt{\%eax})))$$
$$\mathit{sh}(\texttt{\%ebx}) := \text{split4B}(B_0(\mathit{sh}(\texttt{\%eax})),\ B_1(\mathit{sh}(\texttt{\%ebx})),\ B_2(\mathit{sh}(\texttt{\%ebx})),\ B_3(\mathit{sh}(\texttt{\%ebx})))$$

where a $B_n(x)$ node represents the extraction of the nth byte of node x. In other words, the shadows of both `%eax` and `%ebx` are split, and byte zero of the split `%ebx` shadow is updated to be equal to byte zero of the split `%eax` shadow.

Note that Redux updates *sh*(`%eax`) for this operation by splitting it in-place. This is not essential, but if part of `%eax` is used in a subsequent 1-byte operation, having done this Redux avoids having to split it again. This reduces the number of nodes built by around 15%–25%. Unfortunately, `%eax` may then be used in a 4-byte operation, the node for which will now have an unnecessarily split operand, which makes the DDFG larger and uglier. But Redux can remove these unnecessary splits in the rewrite stage (see Section 5.3.7).

Split nodes are fiddly and complicate the implementation significantly, but they are hard to avoid. The other possibility would be to shadow every byte of memory separately, and use "merge" nodes for word-sized operations. This would bloat memory use even more, and most operations are done at the word level, so the current approach seems the best.

5.3.5 Memory Management

Just like Memcheck and Annelid, Redux sets up shadow memory lazily, in 64KB chunks. As for node allocations, Redux manages its own memory pool. Nodes are allocated sequentially from 1MB superblocks, which is very simple and fast. No garbage collection is done; there has been no need yet, since the programs looked at so far have been small and required only a few megabytes of nodes (Section 5.6.4 discusses the use of Redux with bigger programs).

If garbage collection of nodes were implemented, reference counting would probably be most suitable, rather than mark-sweep or copy collection. This is because the root set during execution is huge—every shadow register, and every shadow memory word—so tracing would be prohibitively slow. Also, there are no cycles to cause problems. Redux could allocate perhaps three bits in each node to track how many references it has, and the count could saturate at eight; if a node was referenced more than eight times it would never be deallocated. In practice, only a tiny fraction of nodes have this many references.

5.3.6 Lazy Node Building

One important optimisation reduces the number of nodes built. For each word of static memory initialised at start-up, Redux does not build a node, but instead tags the lowest bit of its shadow. Because Redux's allocator ensures nodes are word-aligned, the bottom two bits of real node pointers are always zero, so this tagging distinguishes these pointers from ordinary pointers. When reading a node from a shadow word, Redux first checks if the bottom bit is marked, and if so, unmarks the bit and builds a constant node for that word. Thus, nodes are built for static words only when they are used. For Hello World, this avoids building almost 700,000 unnecessary nodes, because a lot of static memory—particularly code—is never used as data.

5.3.7 Rewriting and Printing

Once the program terminates, Redux performs three passes over the parts of the DDFG reachable from the root set of system call nodes. The first pass counts how many times each node is used as an input to another node; those used only once can be rewritten more aggressively. The second pass performs rewriting, mostly making peephole simplifications of the graph that make it more compact and prettier. The third pass prints the graphs.

For the factorial examples in Figure 5.1, 80 nodes were built for the operations within `main()`, but only nine are shown (the "_exit" node is built outside of `main()`). By comparison, for the empty C program that just returns zero, Redux builds eight nodes within `main()`, one for the constant zero, and seven book-keeping nodes for building up and tearing down the stack frame, which do not appear in the drawn DDFG. Outside `main()`, 21,391 nodes were built; Valgrind traces pretty much *everything*, including the dynamic linker (which links functions from shared objects on demand), which accounts for most of these.

Redux supports two graph formats. The first is that of dot, a directed graph drawer that produces PostScript graphs, which is part of AT&T's Graphviz package [7]. The second is that of aiSee, an interactive graph viewer [1]. Dot's graphs are prettier, but aiSee's interactivity gives more flexibility; for example, the node layout algorithm can be changed. The two programs have very similar input languages, so supporting both is not difficult. All the examples in this dissertation were drawn by dot.

5.3.8 Crucial Features

Redux's analysis is very heavyweight. Nonetheless, it was reasonably easy to implement within Valgrind. As Table 2.2 showed, Redux is implemented in 5,503 lines of C code, including blank lines and comments.

As was the case for Memcheck and Annelid, the most crucial features Valgrind provided was support for location metadata and shadow computation.

5.4 Essences

This section considers possible ways of comparing programs, and shows how programs that perform the same computation in very different ways can have similar or identical DDFGs.

5.4.1 Program Equivalence

There are various ways to think about whether two programs are "equivalent". At one extreme, if one considers only their visible behaviour, the only important thing is which system calls are made, their order, and what their inputs were. For a given input I, if two programs executed the same system calls in the same order with the same inputs, they are equivalent with respect to that input I (ignoring timing issues, which may or may not be important).

The other extreme is to consider two programs equivalent with respect to input I if they execute the same instruction sequence when given that input I. This idea of equivalence is so rigid it is almost useless; two programs would probably have to be identical to fit this definition.

Using DDFGs to compare programs gives us a definition of equivalence, based on data flow, that is somewhere in between these two extremes. I call this level of detail shown in DDFGs the *essence* of a program. The reason for choosing this word should hopefully be clear after the following three

```
int faca(int n, int acc) {
   if (n <= 1)
      return acc;
   else
      return faca(n-1, acc*n);
}

int main(void)
{
   return faca(5, 1);
}
```

Figure 5.4: Accumulator recursive `fac(5)` in C

examples.

5.4.2 Factorial in C

Figure 5.1 showed the DDFGs for computing the factorial of five in C, using iteration and naive recursion. The graphs have the same nodes, because they perform the same operations, but different shapes, because their data flow is different. However, one can define the factorial function tail-recursively so that it is equivalent—i.e. it has the same data flow—to the iterative version, as in the program in Figure 5.4. The DDFG for this program is identical to that of the iterative factorial program on the left-hand side of Figure 5.1.

5.4.3 Factorial on a Stack Machine

Encouraged by this result, I rewrote the iterative factorial program in a small stack machine language, and ran it on an interpreter (written by Harald Søndergaard and Peter Stuckey). The program, and the resulting DDFG, are shown in Figure 5.5. The function `read_file()` was folded; it reads the stack machine program from file, and produces the outputs seen in the "f-out" nodes—the integers 5, 1 and 1 which were converted from the ASCII characters '5', '1' and '1' read from the program file (underlined in Figure 5.5).

It is immediately obvious that the part of the graph computing the factorial is almost identical to those seen previously. The only difference is that instead of using a `decl` x86 instruction to decrement the loop counter, the stack machine program uses a `subl` x86 instruction with 1 as the argument.

Redux sees only pure data flow; the workings of the interpreter such as the pushes and pops, and loads and stores to memory of intermediate results, are completely transparent. The DDFG shows the essence of the computation which is (almost) identical to that of the C version.

5.4.4 Factorial in Haskell

Finally, I tried the same experiment in the lazy, purely functional language Haskell [88], using the Glasgow Haskell Compiler [47]. The program and graph are shown in Figure 5.6. Unlike the

```
push 5
store 2
push 1
store 1
load 2
store 0
labl 100
load 0
push 1
psub
bz 101
load 1
load 0
mul
store 1
load 0
push 1
sub
store 0
goto 100
labl 101
load 1
halt
```

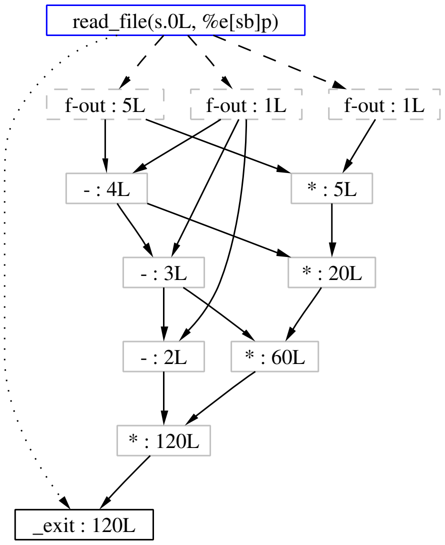


Figure 5.5: DDFG for iterative `fac(5)` on an interpreted stack machine

previous programs, this one does not compute the factorial of five and return the value. Instead it computes the factorial of five using naive recursion and accumulator recursion, adds the two results, and prints the sum. This is because Haskell programs always return zero to the operating system unless they halt with an exception. I also chose to fold the functions `startupHaskell()` and `shutdownHaskellAndExit()`, which initialise and terminate the Haskell run-time system, and are not of interest here.

The two factorial computations are in the graph's top right-hand corner. They are almost identical to those in Figure 5.1; again only small differences exist, such as the use a subtraction instruction instead of a decrement instruction. Most of the rest of the graph shows the conversion of the answer 240 into the characters '2', '4' and '0' that are written with `write()`. Once again, despite the program using a very different execution method, the essence of the computation is the same.

5.5 Possible Uses

DDFGs are intriguing, but it is not immediately obvious how they should be used—they are a solution looking for a problem. This section describes two specific uses that have been implemented, and several other possible uses.

5.5.1 Debugging

Standard debuggers do not allow backwards execution. This is a shame, because the usual debugging approach is to find a breakpoint after a bug has manifested (e.g. when a variable has an incorrect value), and then repeatedly set breakpoints earlier in the execution, restarting the program each time, until the erroneous line is identified.

Redux cannot be used for backwards execution, but it does provide a history of all previous com-

```
main = putStrLn (show (facr 5 + faca 5 1))

facr 0 = 1
facr n = n * facr (n-1)

faca 0 acc = acc
faca n acc = faca (n-1) (acc*n)
```

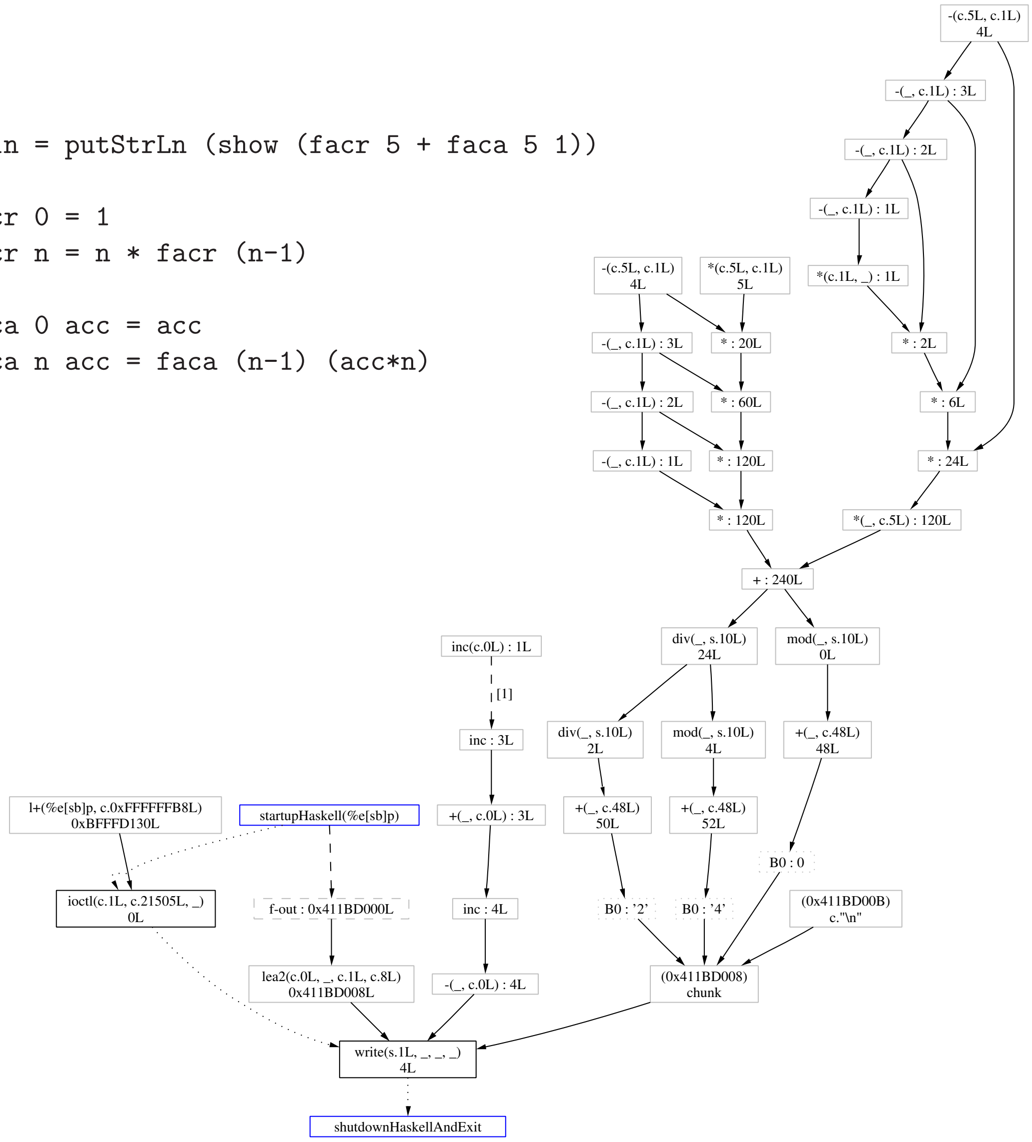


Figure 5.6: DDFGs for iterative and recursive `fac(5)` in Haskell

putations, and can present that information in a highly readable way. Instead of a invoking a "print" command on an incorrect variable (in memory or a register) at a breakpoint, a user could invoke a "why?" command that prints out the sub-graph reachable from the specified variable. This graph would show the entire computational history of the variable, which would hopefully make it simple to identify the bug.

As an example of this, when I first implemented the Haskell factorial program in Figure 5.6, `faca` was defined wrongly. I generated the graph without running the program normally first, and the result for the final "*" node for `faca` was zero instead of 120. My immediate reaction was to look back at the code to see what the problem was, but I instead looked more carefully at the graph. Tracing back from the erroneous "*" node with value zero, I could see the shape of the computation was identical to that in Figure 5.1, but that the result of every node was zero. It was instantly obvious that the problem was that I had used zero as the initial accumulator value instead of one. It was also clear that the problem was not, for example, caused by an incorrect number of recursive calls to `faca5`. This example is also notable because laziness means that Haskell programs are notoriously difficult to debug.

Implementing simple support for sub-graph inspection within Redux was quite straightforward—printing a sub-graph for a specific variable part-way through execution is barely different to printing the graph at the program's end. All it required was a way to specify breakpoints. They were added using client requests (see Section 2.3.12). In this case, the client request indicated the address of the variable of interest. This technique requires recompiling the client program to specify a new variable to inspect, which is unfortunate. A better way would be to specify breakpoints via command line arguments, or interactively. This is quite possible within Valgrind by better utilising the debug information, but it has not been implemented.

5.5.2 Dynamic Program Slicing

From debugging via sub-graph inspection, it is a short step to dynamic program slicing [6]. The sub-graph reachable from a variable represents all the operations that contributed to its current value. If Redux annotates every node with the address of its originating instruction, the addresses of the nodes in that sub-graph form a (non-executable) dynamic data slice [5] for that variable, with respect to the program input. The slice is the "projection" of the sub-graph onto the code.

I did this. Adding instruction addresses to nodes was simple. I modified Redux to produce the slice information in a format readable by Cachegrind's source annotation script (see Section 3.3.4). Figure 5.7 shows two program slices. The first is for the exit code of the iterative factorial C program from Figure 5.1. The two numbers on each line indicate how many constant and non-constant nodes originated from it. A '`.`' represents zero.

Compare that to the second slice, which is for the return value of the stack machine implementation of `fac(5)` from Figure 5.5. Most of the nodes arose from the conversion of digit characters to integers, which were hidden in Figure 5.5 by the folding of `read_file()`. The rest of the nodes come from the actions dealing with the stack machine's `sub` and `mul` instructions. This comparison emphasises just how well Redux can see through a program's book-keeping.

One disadvantage of this approach compared to the sub-graph printing is that not all relevant source code may be available. In the stack machine example, some nodes were not represented because they came from `glibc`, for which the source code was not present.

The most obvious use of this is again for debugging. Sometimes the sub-graph alone might be difficult to interpret, and having a direct connection to the program source might make a variable's computational history much easier to understand. It could also be used in any other way program slicing can be used [109], such as program differencing, software maintenance, or regression testing, but other slicing techniques that have lower overheads might be more appropriate.

Slice of C version:

```
.  .  int fac(int n) {
1  .     int i, ans = 1;
.  3     for (i = n; i > 1; i--)
.  4        ans = ans * i;
.  .     return ans;
.  .  }
.  .
.  .  int main(void) {
1  .     return fac(5);
.  .  }
```

Slice of stack machine version:

```
-- line 171 ----------------------------------------
 .   .  if (neg) ++s;
13  18  for(n = 0; isdigit(*s); n = n*10 + *s++ - '0')
 .   .    ;
-- line 242 ----------------------------------------
 .   .   case add:  stk[sp+1] = stk[sp] + stk[sp+1]; ++sp; break;
 .   3   case sub:  stk[sp+1] = stk[sp+1] - stk[sp]; ++sp; break;
 .   .   case psub: r = stk[sp+1] - stk[sp];
-- line 247 ----------------------------------------
 .   .     ++sp;break;
 .   4   case mul: stk[sp+1] = stk[sp+1] * stk[sp]; ++sp; break;
 .   .   case div: r = (int)stk[sp+1] / stk[sp]; stk[sp+1] = r;
```

Figure 5.7: Slices for the return value of factorial programs

5.5.3 Other Uses

These examples are just a starting point. The following list has several suggestions. Some could not be achieved with Redux in its current form, or might not be feasible in practice, but they give an idea of the range of uses DDFGs might have.

- *Program comprehension.* As a generalisation of debugging, DDFGs could be used not to find bugs, but to generally improve understanding of exactly what a program is doing, and where its data flows.

- *De-obfuscation.* Redux can see through some simple kinds of obfuscation. For example, running a program through some kind of interpreter to hide what it is doing will not be of any use against Redux, as the stack machine interpreter example in Section 5.4.3 showed. For more highly obfuscated programs, the DDFG could provide a good starting point for understanding what the program is doing. A DDFG might even be useful for analysing cryptographic code, e.g. by finding tell-tale patterns of computation and extracting a key somehow.

- *Value seepage.* Debugging with sub-graphs considers the past history of a value. The dual to this is to consider a value's future: how is it used in the rest of the program? Where does it reach? Such information could be useful for understanding programs better, especially security aspects. This is the same idea used in forward slicing [55]. It also has a similar feel to Perl's *taintedness* tracking [114], whereby values from untrusted sources (e.g. user input) cannot be used as is, but must be "laundered" in some way, otherwise a run-time error occurs. Redux would have to be changed to support this, because of the way the graphs are built—each node points back to nodes built in the past, and does not record how its value is used in the future.

- *Program comparison.* Since Redux sees the essence of a program's computation, it could be used to provide a semi-rigorous comparison of programs. This might be useful for determining whether two programs share part of their code, or use the same algorithm to compute something. The comparisons described in Section 5.4 are preliminary, but quite promising; this looks like an area worthy of more study.

- *Decompilation.* The standard approaches to decompilation (e.g. [27, 76]) are purely static. When a decompiler cannot decompile part of a program in this way, dynamic information such as that in a DDFG might be helpful.

- *Limits of parallelism.* Since the DDFG represents the bare bones of a computation, it gives some idea of the level of parallelism in a program's computation, independent of exactly how that computation is programmed. This parallelism is at a somewhat higher level than instruction-level parallelism [113]. Speaking very generally, a wider DDFG implies more inherent parallelism in a program.

- *Test suite generation.* If Redux tracked conditional branches, for each branch that is always or never taken, it might be possible to work back through the DDFG from the conditional test's inputs, and determine how the program's input should be changed so that the branch goes the other way. This could be useful for automatically extending test suites to increase their coverage. While this is a nice idea, in practice it would be very difficult, not least because of the problems Redux has with tracking conditional branches, described in Section 5.6.3.

Even if these suggestions turn out not to be practical, Redux is useful just for having shown how program essences represent the very heart of a program's computation. Redux also provides an excellent demonstration of the power of Valgrind, and the use of location metadata and shadow computation in general.

5.6 Difficulties

What Redux does—tracking the entire computational history of a program—is quite ambitious. There are multiple practical difficulties involved. This section describes some of them, and how they have been tackled, with varying levels of success.

```
int main(int argc, char* argv[]) {
   if (<complex condition using argc, argv[]>)
      return 1;                 // DDFG: _exit(c.1L)
   else
      return 0;                 // DDFG: _exit(c.0L)
}
```

Figure 5.8: A program with a complex condition

5.6.1 Normalisation

Since I claim that the DDFG represents the essence of a program's computation, some kind of normalisation should take place, so that small unimportant differences can be ignored. One example from Section 5.4 was the difference between a +(x,1) node and an inc(x) node. An x86-specific example is the `lea` instruction, which is often used not for calculating addresses, but as a quasi-three-address alternative to the two-address `add` instruction. Also, the instruction `xorl %reg,%reg` (or `subl %reg,%reg`) is often used to zero a register `%reg`, because it is a shorter instruction than `movl 0x0,%reg`.

Currently, a few transformations are hard-wired into the graph rewriting and printing passes. For example, an `xorl %reg,%reg` node is transformed into a constant node "z.0L" (the 'z' indicates the value comes from a special instruction that always produces zero). A more general approach would be to have some kind of mini-language for specifying transformations of sub-graphs.

5.6.2 Loop Rolling

Redux already does some very basic loop rolling for one case—long chains of repeated "inc" or "dec" nodes. It is important to avoid bloating the drawn graphs with boring chains. This loop rolling is hard-wired into the graph printing phase; no nodes are actually removed from the graph. A small step further is to do the same for chains of nodes that add or subtract the same constant; this case occurs in the graphs of some programs.

A bigger challenge is to find a more general method for rolling up loops. It is not clear how to do this; loops with very simple data flow dependencies and few operations are not hard, but the difficulty jumps greatly as the loops grow and/or their data flow becomes more tangled. Representation of rolled loops is another issue; a good representation is not obvious even for the simple factorial loops in Figure 5.1.

5.6.3 Conditional Branches

So far, I have not considered conditional branches at all. Often this is the best thing to do; ignoring them helps cut the DDFG back to a program's essence. However, sometimes the conditions are critical. Consider the C program in Figure 5.8. Redux would ignore the condition, the most interesting part of the program, and produce a useless graph with a single node, "_exit(c.0L)" or "_exit(c.1L)".

Unfortunately, only a tiny fraction of conditionals are interesting, and choosing which ones to

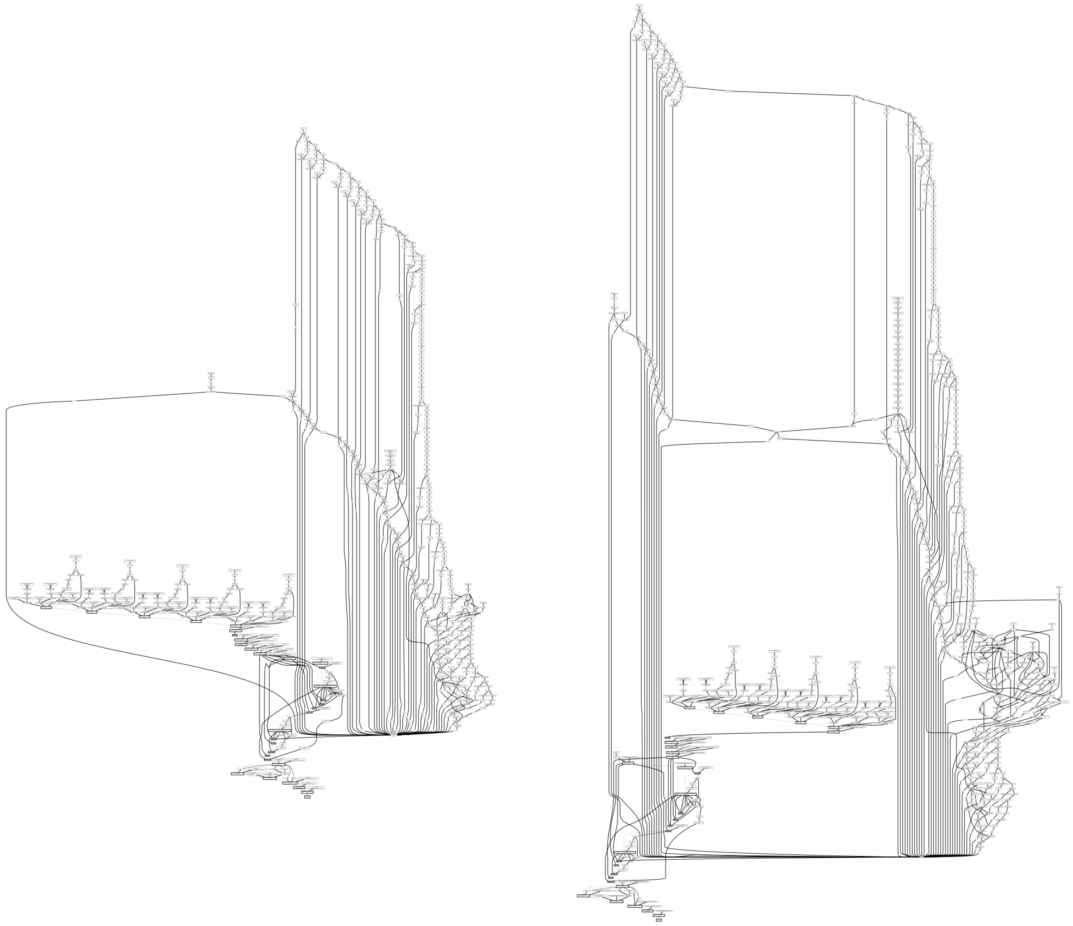


Figure 5.9: DDFGs for bzip2 when compressing a two-byte file and a ten-byte file

show is difficult. One way to show them would be to annotate edges with a node representing the branch (or branches) that had to be taken for that edge to be created. Each branch node would have as inputs the values used in the conditional test. In the small example above, the edge between the "c.0L" or "c.1L" node and the "_exit" node would be annotated by a branch node that represents the complex condition. Thus the interesting part of the computation would be included.

To do this properly one must know the "scope" of a conditional, i.e. know which conditional branches each instruction is dominated by. Unfortunately, it is not clear how to determine this from the dynamic instruction stream that Redux sees. One possibility would be to modify a C compiler to insert client requests that tell Redux the start and end of conditional scopes. But even if this was done, it is likely that the graphs would be bloated terribly by many uninteresting conditions.

5.6.4 Scaling

The most obvious and pressing problem with Redux is that of scaling. It works well for very small programs, but it is unclear how useful it could be on larger programs. There are two main aspects to this problem.

First, the bigger difficulty is drawing and presenting the graphs. Figure 5.9 shows two DDFGs for the compression program bzip2 (a 26KB executable when stripped of all symbol information); the left graph is for compressing a two-byte file, the right graph is for compressing a ten-byte file. On a 1400MHz Athlon with 256MB of RAM, running bzip2 under Redux took about 0.8 seconds for both cases, but the graph drawer dot took 8 seconds to draw the first graph, and over two minutes to draw the second graph. The interactive graph viewer aiSee has similar difficulties. The problem is that the graphs are highly connected; long edges between distant nodes slow things down particularly. Presenting the graphs in an intelligible way is also important. The graphs in Figure 5.9 are already unwieldy. Much effort has already been put into making the graphs more compact. The `--fold` option, used for Figures 5.2, 5.5 and 5.6, is a good start for mitigating these problems—Figure 5.6 is four times larger without folding.

Second, recording all this information is costly. This is unavoidable to some extent, but there is a lot of room in the current implementation to reduce overhead, by being cleverer with analysis code to build fewer nodes, adding garbage collection, and so on. The memory space required for nodes, which can be proportional to the running time of the program, could be removed by streaming them to disk. This could be done linearly because all nodes are constant and always refer to previously created nodes. I have not concentrated on this issue yet because the scaling problems of drawing the graphs are more limiting at the moment.

One likely way to deal with both problems is to be more selective. Currently, all reachable nodes are shown by default, and functions can be folded if the user specifies. Perhaps this should be inverted so that the default is to show a small amount of information, and the user can then specify the interesting parts of the program. Being able to interactively zoom in on parts of the graph would be very useful. Support for outputting graphs in aiSee format was added for this reason, because it allows groups of nodes to be folded up. However I have not yet had much chance to experiment with this facility. Alternatively, automatic factoring or compacting of similar sub-graphs would be very useful, if done well; it is hard to say yet how well this could be done.

5.6.5 Limitations of the Implementation

Because Redux is a prototype, it has not been tried on a great range of programs. It cannot yet handle programs that use floating point arithmetic, nor multi-threaded programs. There are no fundamental reasons for this, it is mostly a matter of implementation effort. It also has not been tried on many programs larger than those mentioned, such as bzip2.

5.7 Related Work

Static data flow graphs (SDFGs) are commonly used in static analysis of programs. A DDFG is a partial unfolding of a SDFG, but with non-value producing operations (such as assignments) removed, and with control flow "instantiated" in a way that omits all control flow decisions. These differences are crucial; a DDFG omits a lot of information and thus is less precise than an SDFG, which means that its potential uses are very different. For example, the SDFGs for the factorial programs examined in Section 5.4 would be much more varied than their DDFGs, making them less useful for finding the essence of a program. If Redux could perform loop rolling as described in Section 5.6.2, the rolled DDFGs would be more similar to SDFGs, but still fundamentally different. SDFGs are also typically done at the level of source code, rather machine code.

The dynamic dependence graph [6, 4] is quite similar to the DDFG, but with additional control nodes and edges, and expressed at the level of source code rather than machine code. The additional control representation makes the graph a much less abstract representation of a program. Agrawal *et al* used dynamic slicing (but not using the dynamic dependence graph, as far as I can tell) in their SPYDER debugger, which annotated code in a way similar to that described in Section 5.5.1; unlike Redux, SPYDER's slices included control statements. Choi *et al* used a similar dynamic graph in their Parallel Program Debugger (PPD) [26].

Other similar visualisation tools to Redux are for tracing, visualising and debugging execution of Haskell programs. Hat [25] transforms Haskell programs in order to trace their execution. Three text-based tools can be used to view the traces; one of them, Hat-Trail, allows backwards exploration of a trace. The Haskell Object Observation Debugger (HOOD) [46] is a library containing a combinator `observe` that can be inserted into programs to trace intermediate values, particularly data structures. It also presents the trace information in a text-based way. Both these tools have been inspired by the fact that more conventional debugging techniques are more or less impossible in Haskell because it is lazy. They all present information at the level of source code.

5.8 Conclusion

This chapter described Redux, a visualisation tool, which was built with Valgrind. Redux implements a novel DBA to create dynamic data flow graphs of programs, and it is the only visualisation tool created with a DBI framework that I am aware of. These graphs show the computational history of the program, and reduce programs to a minimal *essence*; programs that compute the same thing in multiple ways have identical or very similar graphs.

Redux does not represent a solved problem—the uses of DDFGs are not completely clear, and the implementation has significant practical shortcomings. Nonetheless, Redux shows again the importance of Valgrind's support for location metadata and shadow computation, and the DDFGs it produces—even if none of the proposed uses for them end up being practical—are useful just for having shown how program essences represent the very heart of what programs are computing.

CHAPTER 6

Describing Tools

This chapter describes a novel and useful system of semi-formal descriptions of dynamic binary analysis tools, and strongly emphasises the importance of metadata in dynamic analysis.

6.1 Introduction

The previous chapters described a DBI framework for building DBA tools, and three very different DBA tools built with the framework. This chapter steps back and considers what can be said about all DBA tools. The details are largely independent of whether the DBA is implemented using DBI or static binary instrumentation.

6.1.1 Tool Differences and Similarities

All DBA tools have a small number of possible aims; most often it is to obtain information in order to improve program performance or correctness. These goals are very broad, and each DBA tool can target only a small number of aspects, such as cache performance, or bounds-checking. The information they record varies widely, and so do the ways in which they use that information. The end result is that there are a wide variety of DBA tools, using many different techniques to achieve their goals.

However, despite the obvious differences between different tools, *all these tools work in the same basic way*. Basically, all tools instrument interesting parts of the client program with analysis code. This analysis code takes raw inputs, possibly produces *metadata* about the running program by analysing the inputs, and performs I/O actions to present useful information to the user (be it a human or another program). Roughly speaking, metadata is information that is not available from the normal program state; for example, the contents of a register is not metadata, but a count of how many times the register has been accessed is. That definition of metadata is enough for now, but one of the things this chapter explores is the question of what exactly metadata is.

It makes sense to view DBA tools in terms of analysis code and metadata, since code and data are the common characteristics of all programs. Niklaus Wirth expressed this simply in the title of his 1975 book [117]:

Algorithms + Data Structures = Programs.

Paraphrasing this for dynamic binary analysis gives:

Analysis Code + Metadata = Dynamic Binary Analysis Tools.

And of these two concepts, metadata is the more important. After all, analysis code is just a means to an end; metadata (and its use) *is* the end.

6.1.2 Tool Descriptions

This observation leads to a question: if tools all work in the same basic way, can one precisely characterise these similarities, so their workings can be more easily understood, and comparisons between tools can be made? The answer is *yes*.

This chapter describes a system of semi-formal DBA tool descriptions. Each description has two halves. One half describes the metadata, the other half describes the analysis code that is required to create and maintain the metadata. Each of the two halves has a formal component, and an informal, plain language component. This formal/informal split is crucial, and is similar to an interface/implementation split for a program. The descriptions primarily describe DBA tools that use passive analysis code that does not change a program's semantics; most kinds of active analysis are outside its scope, except for one exception (function replacement) that is important for describing the Valgrind tools in the previous chapters.

The descriptions are most useful for understanding tools, both before and after they are written. This is because they encourage a directed way of thinking about tools that emphasises their most important aspects, and draws out subtleties that may not be otherwise clear. They can also be used for comparing tools, and identifying what features a framework such as Valgrind should provide to make tool-writing easier. Also, they show that analysis code and metadata are the fundamental concepts underlying DBA, and provide a way of characterising the limit of what can be achieved with DBA tools. Finally, they lead to a precise definition of dynamic analysis, and a better understanding of dynamic analysis in general, based around the idea that metadata is the approximation of a program's past.

However, they are not a formal system, they involve an element of subjectivity, and they cannot be used for any kind of automated reasoning. They represent a good first attempt at identifying a unifying principle underlying all tools, but there may be room for future improvement, such as a greater level of formality. Nonetheless, they are still very useful in their current form, as this chapter shows.

6.1.3 Chapter Structure

This chapter is structured as follows. Section 6.2 gives a quick overview of the descriptions. Section 6.3 introduces preliminary definitions used in the descriptions. Section 6.4 presents the descriptions more formally. Section 6.5 gives descriptions of some simple DBA tools. Section 6.6 shows how more complex descriptions can be constructed. Section 6.7 gives full descriptions for the Valgrind tools described in the previous chapters. Section 6.8 discusses the limits of DBA tools. Section 6.9 considers what dynamic analysis really is, and gives a precise definition. Section 6.10 considers the benefits and shortcomings of the descriptions. Section 6.11 discusses related work, and Section 6.12 concludes.

6.2 The Big Picture

DBA tools can be very complex, as Chapters 2–5 have shown. And yet, all tools can be described simply by their analysis code and metadata. Section 6.1.1 claimed that metadata is the more important of the two concepts. Correspondingly, when describing a tool, the metadata is the more instructive of the two. Consider Raymond's [93] rephrasing of Brooks' [16] quote:

> Show me your code and conceal your data structures, and I shall continue to be mystified.
> Show me your data structures, and I won't usually need your code; it'll be obvious.

This idea certainly applies when describing tools. For this reason, the semi-formal descriptions in this chapter are metadata-centric, i.e. structured around metadata. This makes the tools they are describing much easier to understand. For the same reason, the descriptions of the Valgrind tools given in Chapters 2–5 were deliberately based around the metadata those tools maintain.

This section provides a broad overview of the descriptions, and gives a simple example, which should make the detailed explanations in Sections 6.3 and 6.4 more comprehensible, before more examples are given in Section 6.5.

6.2.1 Description Basics

Each description has four parts. The following list gives a high-level introduction to them; it uses Memcheck and Cachegrind as examples, because together they track the most common kinds of metadata.

1. *Formal description of metadata (**M**-part).* This part describes what program/machine entities the tool "attaches" metadata to. Only three of these attachment points, called *M-hooks*, are distinguished.

   (a) Global metadata, e.g. Memcheck's record of the heap, or Cachegrind's simulated cache state.

   (b) Per-location (register or memory) metadata, e.g. Memcheck's A (addressability) bits.

   (c) Per-value metadata, e.g. Memcheck's V (validity) bits.

   The M-hooks involved are formally described in the **M**-part, but the form of the metadata is not.

   A tool can attach metavalues to more than one kind of M-hook. If per-location or per-value metadata is used, a metavalue is typically attached to every location or value in the program.

Note that per-location and per-value metadata could actually be done using global metadata—consider a global table holding one metavalue per register, compared with per-register metadata. Nonetheless, they are treated separately because they are very common, and distinguishing them in this way makes many descriptions more concise and easier to understand.

2. *Informal description of metadata (**M**′-part).* This part describes the form of each kind of metadata that was specified in the **M**-part. This is an informal, plain language description, because metadata can take almost any form. For example, Memcheck's per-memory-byte A bits are mere booleans, whereas Cachegrind's global cache state is a complex data structure.

3. *Formal description of analysis code (**I**-part).* This part describes exactly which parts of the client's code are instrumented with analysis code. The instrumentation points are called *I-hooks*. There are three built-in I-hooks:

   (a) program start;

   (b) instructions;

   (c) program end.

   For example, Cachegrind instruments every instruction, and instructions that access memory are instrumented differently from those that do not. At a program's end, it dumps the gathered per-source line counts to file. In comparison, Memcheck instruments many, but not all, instructions, and the instrumentation for each instruction depends on its opcode. Memcheck also instruments functions (such as `malloc()`) and system calls (such as `mmap()`) that affect the addressability of memory.

   Custom I-hooks can be constructed from the built-in I-hooks, e.g. for functions, system calls and memory operations.

   As well as describing which I-hooks are instrumented, this part also describes exactly what are the raw inputs to the analysis code, and whether the analysis code updates metavalues or does I/O. However, the exact form of the analysis code is not specified; roughly speaking, it describes the analysis code's interface.

   For example, the inputs to Cachegrind's analysis code functions include the instruction's address, and its data address and size (for those instructions accessing memory); and Cachegrind only performs I/O at the program's end. Memcheck's shadow computation updates V bits by taking the V bits of the values used by each instruction as inputs. Also Memcheck can perform I/O as the program proceeds, when issuing error messages.

   Finally, this part also makes clear which decisions are made statically, at instrumentation-time, and which decisions are made dynamically, at run-time.

4. *Informal description of analysis code (**I**′-part).* This part describes what the analysis code does with the inputs specified in the **I**-part; roughly speaking, it describes the analysis code's

implementation. Again, this description is informal, using plain language, because the analysis code can do almost anything.

For example, the informal description of Cachegrind's analysis code would include a description of how the simulated cache is updated, and how the per-source line hit/miss counts are updated. For Memcheck, the description could describe how the V bit computations work.

The formal **M**-part and **I**-part are the parts of a description that do not vary much between different tools. The informal **M′**-part and **I′**-part are the parts of a description that do vary greatly between different tools. This split captures the differences and similarities between tools outlined in Section 6.1.1.

6.2.2 A First Example

The following example describes a simple tool that counts the number of jump instructions executed by a program, and prints the total out when the program terminates.

M $\boldsymbol{g}.mv$

M′ $\boldsymbol{g}.mv$ is an integer, initialised to zero.

I $instr(\boldsymbol{i}) : is\_jmp(\boldsymbol{i}.opcode) \;\Rightarrow\; inc(!\boldsymbol{g}.mv)$
$end(\_) \;\Rightarrow\; io(\boldsymbol{g}.mv)$

I′ *is_jmp* succeeds if the opcode is that of a jump instruction.

inc increments its argument.

io prints the final count.

Full details of the syntax are given in Sections 6.3 and 6.4, but the following text should be enough to give a basic understanding.

The **M**-part is the formal description of metadata. The $\boldsymbol{g}.mv$ indicates that the tool maintains a global metavalue. The **M′**-part is the informal description of metadata. It indicates that the global metavalue is an integer, initialised to zero.

The **I**-part is the formal description of analysis code. The left-hand side of the first *rule*, before the $\Rightarrow$ symbol, states that every instruction that satisfies the predicate *is_jmp*—which succeeds or fails depending on the instruction's opcode—is instrumented. The right-hand side of the first rule states that the analysis code updates the global metavalue with the function *inc*. The ! symbol indicates that the metavalue $\boldsymbol{g}.mv$ is updated by *inc*. The second rule states that the program's termination is instrumented to call a function *io*, which is passed the global metavalue; the name *io* indicates that it performs some I/O. The **I′**-part is the informal description of analysis code. It describes exactly what *is_jmp*, *inc* and *io* do.

This is all fairly straightforward. One non-obvious thing worth mentioning is that in the **I**-part, everything on the left-hand side of each rule represents what happens at instrumentation-time, and everything on the right-hand side represents what happens at run-time. In particular, in the first rule, the *is_jmp* predicate is evaluated at instrumentation-time. It could be done instead at run-time, as the following rule describes.

I $\quad$ *instr*(***i***) $\Rightarrow$ *if is_jmp*(***i***.*opcode*) { *inc*(!***g***.*mv*) }

This is inefficient, as the test is done—always giving the same result for each distinct instruction—every time the instruction executes, rather than just once at instrumentation-time. This issue of instrumentation-time vs. run-time evaluation is important, and recurs several times in the rest of this chapter.

6.3 Preliminaries

This section introduces basic concepts and notation used in the descriptions.

6.3.1 M-hooks and Built-in I-hooks

As Section 6.2 said, metadata can be attached to M-hooks, and analysis code can be attached to I-hooks. Each hook has built-in *attributes*; if a hook is thought of as a kind of structure, attributes are the hook's fields. Attributes serve as inputs to analysis code. The descriptions are expressed in terms of M-hooks and I-hooks, and their attributes.

Table 6.1 gives the types for all hooks and their attributes. Column 1 gives the name of the hook; column 2 gives what kind of hook it is (M or I). Column 3 gives the names of each hook's attributes, and columns 4–6 give each attribute's type, binding time (static, dynamic, or mixed; explained below), and a brief description. The following list describes the table's entries in detail.

- *Global.* This M-hook represents global attributes of the program and the machine. It is always referred to as ***g***. The attribute *rs*$\langle n \rangle$ represents the machine's register file and *ms*$\langle n \rangle$ represents every memory location in the machine's address space. The parameter n indicates the granularity; for example *rs*$\langle 1 \rangle$ is the list of every register byte, whereas *rs*$\langle 4 \rangle$ is the list of every register (assuming 32-bit registers). All attributes that are lists have a name that ends in 's' to indicate plural. The binding time of *rs*$\langle n \rangle$ and *ms*$\langle n \rangle$ is *static*—known at instrumentation-time—as they do not change during execution (note that the contents of these register and memory locations do change, but the identities of the locations themselves do not).

  The attribute *debug* represents debug and symbol information, which can be looked at for file, function, line information, etc. The attribute *t*, the current time, is *dynamic*—only known at run-time, and ever-changing. The attribute *mv*, the global metavalue, is also dynamic. It is optional, since a tool does not have to track global metadata. Its type (if present) depends on the tool.

- *RegLoc*$\langle n \rangle$. This M-hook represents a register location, identified by *name*, which holds a Value$\langle n \rangle$, *contents*, that is n bytes wide. RegLocs can have a metavalue, *mv*; this metavalue describes the location itself, rather than the value stored in the location.

- *MemLoc*$\langle n \rangle$. This M-hook represents a memory location, at address *addr*, which holds a Value$\langle n \rangle$, *contents*, that is n bytes wide, and n-aligned. MemLocs can have a metavalue, *mv*;

Element	Hook	Attribute	Type	Binding	Description
Global	M	*rs*$\langle n\rangle$	List(RegLoc$\langle n\rangle$)	s	all register locations
		ms$\langle n\rangle$	List(MemLoc$\langle n\rangle$)	s	all memory locations
		debug	DebugInfo	s	debug and symbol information
		t	Time	d	current time
		mv	?	d	global metavalue (optional)
RegLoc$\langle n\rangle$	M	*name*	String	s	register location name
		contents	Value$\langle n\rangle$	d	contents of location
		mv	?	d	metavalue (optional)
MemLoc$\langle n\rangle$	M	*addr*	Addr	s	address
		contents	Value$\langle n\rangle$	d	contents of location
		mv	?	d	metavalue (optional)
Value$\langle n\rangle$	M	*v*	Bits$\langle n\rangle$	d	the actual bits
		mv	?	d	metavalue (optional)
Instr	I	*opcode*	Opcode	s	opcode
		isize	Int	s	instruction size
		dsize	Int	s	data (operand) size
		addr	Addr	s/d	instruction address
		loc	CodeLoc	s/d	code location
		rs$\langle n\rangle$	List(RegLoc$\langle n\rangle$)	s	RegLocs accessed
		ms$\langle n\rangle$	List(MemLoc$\langle n\rangle$)	sd	MemLocs accessed
		us$\langle n\rangle$	List(Value$\langle n\rangle$)	sd	Values used in computation
		as$\langle n\rangle$	List(Value$\langle n\rangle$)	sd	Values used as auxiliaries
		ds$\langle n\rangle$	List(Value$\langle n\rangle$)	sd	Values defined
Start	I	–			
End	I	–			

Hook:

M	M-hook (can have a metavalue)
I	I-hook (instrumentable code)

Binding:

s	static (known at instrumentation-time)
d	dynamic (known at run-time)
s/d	either (see text for explanation)
sd	mixed (see text for explanation)

Table 6.1: M-hooks, I-hooks, and their attributes

as with RegLocs, this metavalue is about the location itself, rather than the value stored in the location.

- *Value*$\langle n\rangle$. This M-hook represents actual n-byte values that are stored in locations, and that appear as literal constants in code. The actual bits of the value are in *v*. Values can have a metavalue, *mv*. Values can be copied, and are destroyed if overwritten.

  If Values are shadowed with metavalues, every single Value in the program should be shadowed. For stored Values, if the Value is copied from one location to another, the metavalue should also be copied. If a Value is overwritten, its metavalue should also be overwritten. These conditions are all met by shadow computation. If these conditions do not hold, then the metavalue probably really belongs to the RegLoc or MemLoc.

  From a particular implementation viewpoint, such as that of Valgrind, there is no intrinsic difference between attaching metadata to Values and attaching metadata to all RegLocs and MemLocs.[1] This is why Chapters 1, 2, 4 and 5 speak about location metadata in relation to shadow computation. However, there is a significant conceptual difference when it comes to describing tools, and so this chapter distinguishes between location metadata and value metadata.

For each literal Value in the code, its metavalue must also be constant, and any description of a tool that shadows Values with metavalues must provide a function name *const_value_mv* for computing these metavalues.

- *Instr.* This I-hook represents an instruction executed by the program (not, by contrast, an instruction in the instruction set). It includes the instructions in the program's code, and also the instructions in the operating system kernel executed on behalf of the program, for example system call instructions. Each instruction can be instrumented.

  The attributes *opcode*, *isize* and *dsize* are all static and straightforward. The attribute *addr* is tricky because of its binding time. From the point of view of static binary instrumentation it is a static attribute for instructions within the main executable (which must be loaded at a fixed address), but a dynamic attribute for instructions within shared objects (which can be loaded at any address). For DBI *addr* is known at instrumentation-time; however, an instruction's address can change if it is part of a shared object that gets unloaded and then reloaded into a different location (as part of the client's normal execution, i.e. independent of any instrumentation system). The attribute *loc* identifies an instruction's location, which is mostly useful for looking up debug and symbol information. Its form is deliberately vague in Table 6.1 because it can vary; for DBI, it may be the same as *addr*, for static binary instrumentation it may be a location in an object file.

  The attributes *rs*$\langle n \rangle$ and *ms*$\langle n \rangle$ represent the register and memory locations accessed by an instruction; *us*$\langle n \rangle$ represents the Values used in direct computation by the instruction, *as*$\langle n \rangle$ represents the auxiliary Values used by the instruction (e.g. for address computations), and *ds*$\langle n \rangle$ represents Values written by the instruction. The subscript specifies, in bytes, the location sizes. Again, the 's' in the attribute names indicates that they are lists.

  An example will make things clearer. Consider the following instruction.

  ```
  addl %eax, (%ebx)
  ```

  Recall that the second operand is the destination. The location and value attributes (using word-granularity) are as follows, where `%reg` denotes a RegLoc$\langle 4 \rangle$, *`%reg` the Value$\langle 4 \rangle$ within the RegLoc$\langle 4 \rangle$, (`%reg`) the MemLoc$\langle 4 \rangle$ pointed to by `%reg`, and *(`%reg`) the Value$\langle 4 \rangle$ in that MemLoc$\langle 4 \rangle$.

  rs$\langle 4 \rangle$ = [`%eax`, `%ebx`]

[1] This assumes that every value computed by a program is stored in a location, at least temporarily, so that any metavalue can be correspondingly stored in the shadow location. This requires that all implicit intermediate values computed by the original machine code, such as addresses computed by complex addressing modes, are made explicit. This property is satisfied by Valgrind's UCode—all implicit values are made explicit and stored in shadow virtual registers.

$$
\begin{aligned}
ms\langle 4\rangle &= [\ (\texttt{\%ebx})\ ] \\
us\langle 4\rangle &= [\ {*}\texttt{\%eax},\ {*}(\texttt{\%ebx})\ ] \\
as\langle 4\rangle &= [\ {*}\texttt{\%ebx}\ ] \\
ds\langle 4\rangle &= [\ {*}(\texttt{\%ebx}),\ {*}\texttt{\%eflags}\ ]
\end{aligned}
$$

The attributes would change with the granularity; for example, with per-byte granularity $rs\langle 1\rangle$ would contain eight sub-register bytes: `%eax`[31..24], `%eax`[23..16], `%eax`[15..8], `%eax`[7..0], `%ebx`[31..24], `%ebx`[23..16], `%ebx`[15..8], and `%ebx`[7..0].

The binding time of $rs\langle n\rangle$ is static, since the variables accessed by instructions are known at instrumentation-time. However the remainder have a *mixed* binding time. They are partially static, because it is known at instrumentation-time if each element is an empty list or not. However, the number of elements, and what those elements are, changes every time the instruction executes, and so the attributes are also partially dynamic.

Instructions are the most important of the three built-in I-hooks. They can be used to build custom I-hooks that can be used for instrumenting functions, system calls, memory operations, etc., as Section 6.6 explains.

- *Start* and *End*. These I-hooks represent the operations of starting and ending the program. They have no attributes, but they are useful to instrument.

Note that *mv* is the only attribute that tools can write to, due to the assumption of passive analysis code mentioned in Section 6.1.2.

A '.' denotes field access. For example, $\boldsymbol{i}.opcode$ is the opcode of the instruction represented by $\boldsymbol{i}$. The '.' symbol is overloaded so that it can also be applied to a list, giving a list of attributes. For example, $\boldsymbol{g}.rs\langle 4\rangle.contents.v$ is the list of all the actual bit values in all the registers.

6.4 Descriptions

Now that the basic elements of the descriptions—metadata, M-hooks and I-hooks, and attributes—have been covered, this section presents the exact form of the descriptions. It begins by reviewing their four-part structure, then describes each of the four parts in more detail.

6.4.1 Basic Ideas

As Section 6.2 said, the tool descriptions have four parts. They answer the following four questions.

M Which M-hooks have metavalues, if any?

M′ What is the form of the metavalues?

I Which I-hooks are instrumented with metadata updates and I/O actions, and what are their input attributes?

I′ What is the form of the predicates, functions and I/O action(s)?

Parts **M** and **I** can be expressed concisely, precisely, and formally. This is because there are only a limited number of M-hooks to attach metavalues to, a limited number of I-hooks to instrument, and a limited number of raw attribute inputs to analysis code.

In comparison, parts **M′** and **I′** cannot, in general, be expressed both concisely and precisely. This is because there are an unlimited number of metadata and analysis code forms. Therefore, concise, informal, plain language descriptions are used to answer questions **M′** and **I′**. This might sound like it would give vague results, however, it will soon be clear that parts **M** and **I** typically specify a tool to such a degree that the plain language descriptions just "fill in the gaps". Any concise description of a complex tool has to leave out some details, and it turns out that this is the right place to do so.

As mentioned in Section 6.1.2, this formal/informal separation is *crucial*. It is much more obvious to do things without this separation, but any resulting descriptions lack a unifying consistency that makes it clear how different tools relate. (My first attempts at coming up with meaningful tool descriptions suffered from exactly this problem.) Clearly separating the formally describable parts from the rest results in descriptions that are much more concise, precise, and useful. It also helps when considering the limits of DBA tools, as Section 6.8 will show.

As a preliminary, intuitive analogy, consider the separation of a function's type signature from its body. The function `map` (using Haskell syntax):

```
map :: (a -> b) -> [a] -> [b]
```

is a good example. The type signature restricts the function so much that one barely needs to see the function body to know what it does. One often sees a similar effect with the separation of the formal and informal parts of the descriptions.

The following sections describe the four parts of each description in more detail.

6.4.2 Formal Description of Metadata

The formal description of metadata (**M**-part) is a single line which states which M-hooks have metadata attached. It can be omitted if the tool tracks no metadata. The terms $\boldsymbol{g}.mv$, $\boldsymbol{r}\langle n\rangle.mv$, $\boldsymbol{m}\langle n\rangle.mv$, $\boldsymbol{v}\langle n\rangle.mv$ are used to represent metadata attached to Global, RegLoc$\langle n\rangle$, MemLoc$\langle n\rangle$ and Value$\langle n\rangle$ M-hooks respectively. Where present, the subscripts indicate the metavalue granularity, i.e. how many bytes each metavalue represents. Consider the following two examples.

M $\boldsymbol{r}\langle 4\rangle.mv$
M $\boldsymbol{g}.mv$, $\boldsymbol{v}\langle 1\rangle.mv$

Example one states that every (word-sized) register has a metavalue. Example two states that there is a global metavalue, and also that every byte value (in both registers and memory) has a metavalue. Sometimes it is useful to break a metavalue into parts, e.g. $\boldsymbol{g}.mv_a$ and $\boldsymbol{g}.mv_b$.

6.4.3 Informal Description of Metadata

The informal description of metadata (**M′**-part) is written in plain language. It should describe the form of all metavalues, including their initial values, and the *const_value_mv* function (which computes metavalues for constant code Values) if needed. It can be omitted if the tool tracks no metadata. The following is a very simple example.

M′ $\boldsymbol{g}.mv$ is an integer, initialised to zero.

The level of detail is arbitrary, but usually depends on the purpose of writing the description. For example, to broadly explain how a tool works, a short description will probably suffice. To explain in enough detail for someone else to reimplement a tool, more detail might be necessary.

6.4.4 Formal Description of Analysis Code

The formal description of analysis code (**I**-part) consists of one or more formal *rules* that specify an instrumentation schema. Each rule has the following two-part form.

$$i\_hook(\boldsymbol{x}) : static\_ops \quad \Rightarrow \quad dynamic\_ops$$

This means *all I-hooks matching the left-hand side should be instrumented with the analysis code on the right-hand side*. A key point is that the left-hand side (i.e. the text before the ⇒ symbol) represents what happens at instrumentation-time, the right-hand side represents what happens at run-time.

On the left-hand side, *i_hook* must be one of the built-in I-hooks—*instr*, *start* or *end*—or a custom I-hook (custom I-hooks are discussed in Section 6.6). The variable $\boldsymbol{x}$ is bound to the I-hook, and can be used to access the I-hook's attributes in the rest of the rule. The *static_ops* part can be empty, or contain predicates which select a subset of all matching I-hooks for instrumentation. It can also assign the special ***tmp*** variable, which represents the hard-wiring of statically known values into analysis code, which is a mild form of (non-automatic) partial evaluation. Because the left-hand side describes what is happening at instrumentation-time, all attributes mentioned on the left-hand side must be static attributes of $\boldsymbol{x}$.

The right-hand side can contain assignments, functions, I/O actions, and if-statements. Because the right-hand side describes what is happening at run-time, the inputs can be any mixture of static and dynamic attributes of the I-hook variable bound on the left-hand side, and of the global $\boldsymbol{g}$ variable.

For example, consider the following three rules.

$$\begin{aligned}
&end(_) \quad \Rightarrow \quad io(\boldsymbol{g}.mv) \\
&instr(\boldsymbol{i}) : is_jmp(\boldsymbol{i}.opcode) \quad \Rightarrow \quad f(!\boldsymbol{g}.mv, \boldsymbol{i}.rs\langle 1 \rangle) \\
&instr(\boldsymbol{i}) : \boldsymbol{tmp} := g(\boldsymbol{i}.opcode) \quad \Rightarrow \quad if\ is_ok(\boldsymbol{i}.as\langle 4 \rangle)\ \{\ \boldsymbol{g}.mv := h(\boldsymbol{tmp})\ \}
\end{aligned}$$

The first rule specifies that the program's end should be instrumented with the I/O action *io*, which takes the global metavalue $\boldsymbol{g}.mv$ as input. Because no I-hook attributes are used, no I-hook variable needs to be bound on the left-hand side, hence the '_'.

The second rule specifies that all instructions executed that satisfy the predicate *is_jmp* should be instrumented to call *f*, which updates the global metavalue using the register bytes involved in the instruction; the ! symbol indicates that a variable is modified by a function.

The third rule specifies that all instructions should be instrumented with analysis code that performs a run-time check before overwriting the global metavalue with the result of *h*; ***tmp*** is used to cache something computed, once per instruction, at instrumentation-time, so it does not have to be recomputed every time each instruction is executed.

These examples only show assignments to *mv* variables; assignments to non-*mv* variables are also possible for custom I-hooks, as Section 6.6 shows. To distinguish between functions, predicates and I/O actions, predicates will be prefixed with "*is_*" (although they are always obvious from context) and I/O actions will always be prefixed with *io*. The rules do not say how *f*, *g*, *h*, *io*, *is_jmp* or *is_ok* are implemented, or what they do; that is covered in the informal description of analysis code. However, well-chosen names can make their actions obvious.

6.4.5 Informal Description of Analysis Code

The informal description of analysis code ($\mathbf{I}'$-part) is written in plain language. It should describe the form of all predicates, functions and I/O actions named in the rules. This covers what they do, and possibly how they are implemented (e.g. as a C function, or as inline machine code), if that is notable. As was the case for the informal description of metadata, the level of detail required is arbitrary.

6.5 Descriptions of Simple Tools

This section gives multiple example descriptions. It starts with example tools that use no metadata, and then moves onto examples using global, per-location and per-value metadata.

6.5.1 Tools Using No Metadata

Many tools do not record any metadata, but merely react to particular events. It is questionable if they should be described as DBA tools, as they do not do any "analysis" as such. Section 6.9 will return to this point, but for the moment they are a good place to start.

One of the simplest possible tools just prints a string every time an instruction is executed.

I $\quad$ $instr(\_) \;\Rightarrow\; io()$

$\mathbf{I}'$ $\quad$ *io* prints "instruction executed!"

The **M**-part and $\mathbf{M}'$-part are omitted because the tool maintains no metadata. Every instruction is instrumented with the same analysis code, which is an I/O action that takes no inputs. In this case, the analysis code would probably be a call to a C function; this could have been mentioned in the informal description of analysis code.

A slightly more elaborate tool prints the opcode of every instruction executed. This can be implemented in several ways, with small but important differences. The first way is as follows.

I $\quad$ $instr(\boldsymbol{i}) \;\Rightarrow\; io(\boldsymbol{i}.opcode)$

$\mathbf{I}'$ $\quad$ *io* prints a string representing the passed opcode.

Note that *io* presumably contains a case statement which switches on the value of $\boldsymbol{i}.opcode$; the decision about what to print for each instruction is repeatedly made at run-time, even though the result for each instruction is always the same. This could be implemented in the following, more efficient manner.

I $\quad instr(\boldsymbol{i}) : \boldsymbol{tmp} := f(\boldsymbol{i}.opcode) \quad \Rightarrow \quad \boldsymbol{tmp}()$

I′ $\quad f$ returns one of a family of functions that print opcodes.

The idea here is to have a family of opcode-printing functions, and instrument each instruction with the appropriate function; the decision about what to print for each instruction is made (once) at instrumentation-time. This partial evaluation is possible because *opcode* is a static attribute. The special ***tmp*** variable, local to the rule, is used to communicate a value computed statically (on the left-hand side) for an instruction to its analysis code (on the right-hand side). Note that this rule is higher-order, because ***tmp*** is a function.

A third, fairly efficient way of implementing this tool is possible.

I $\quad instr(\boldsymbol{i}) : \boldsymbol{tmp} := opcode2str(\boldsymbol{i}.opcode) \quad \Rightarrow \quad io(\boldsymbol{tmp})$

I′ $\quad$ *opcode2str* converts an opcode to a string pointer.

io prints a given string.

Here the ***tmp*** variable communicates a string pointer obtained at instrumentation-time to *io*. This version is more efficient than the version that chooses the string to print at run-time, but will be slightly less efficient than the version that has a separate I/O action for each opcode, because of the overhead of passing the argument to *io*.

These examples show how decisions can be made at instrumentation-time or at run-time. It is important to distinguish the two cases; generally, static evaluation is preferred, where possible, for efficiency reasons.

As another example, consider the tool that prints “instruction executed!” for every instruction executed within the first five seconds of a program starting.

I $\quad instr(\boldsymbol{i}) \quad \Rightarrow \quad \textit{if } is\_lt5(\boldsymbol{g}.t)\ \{\ io()\ \}$

I′ $\quad$ *is_lt5* succeeds if the time argument is less than 5 seconds.

io prints “instruction executed!”

This example shows a conditional test that must be performed at run-time—and thus appears on the right-hand side—as it involves a non-static input, $\boldsymbol{g}.t$.

From these examples, three things are already clear. First, the descriptions are concise. Second, often the rules are so suggestive that the informal description of analysis code is barely necessary. Third, the distinction between instrumentation-time actions and run-time actions is important for efficiency.

6.5.2 Tools Using Global Metadata

Consider a simple tool that counts the number of instructions executed by a program.

M $\boldsymbol{g}.mv$

M′ $\boldsymbol{g}.mv$ is an integer, initialised to zero.

I $instr(\_) \;\Rightarrow\; inc(!\boldsymbol{g}.mv)$
$end(\_) \;\Rightarrow\; io(\boldsymbol{g}.mv)$

I′ *inc* increments its argument.

io prints the final count.

The first rule is extremely suggestive; without any other inputs, the only sensible action for *inc* is to increment $\boldsymbol{g}.mv$.

Many tools have more complex global metadata. One example is a tool that records the number of times each opcode is executed. This information could be used to categorise code types, e.g. as integer-dominated, or floating-point dominated.

M $\boldsymbol{g}.mv$

M′ $\boldsymbol{g}.mv$ is a table containing one integer per opcode, each one initialised to zero.

I $instr(\boldsymbol{i}) \;\Rightarrow\; inc\_opcode(!\boldsymbol{g}.mv, \boldsymbol{i}.opcode)$
$end(\_) \;\Rightarrow\; io(\boldsymbol{g}.mv)$

I′ *inc_opcode* increments the appropriate entry in the table.

io prints the final per-opcode counts.

The inputs to *inc_opcode* specify that $\boldsymbol{g}.mv$ is a function of the global trace of executed opcodes. As written, the rule uses a single function *inc_opcode*, but it could be changed to be a family of functions, one per opcode and selected at instrumentation-time, like the opcode printer in Section 6.5.1.

A more interesting tool is one that gathers the dynamic instruction counts for each line of source code, and produces information from which the source code can be annotated with the counts. This tool gives useful profiling information, and also gives execution coverage.

M $\boldsymbol{g}.mv$

M′ $\boldsymbol{g}.mv$ is a table holding one integer per line of source code, initially empty.

I $instr(\boldsymbol{i}) : \boldsymbol{tmp} := lookup(!\boldsymbol{g}.mv, get\_src\_line(\boldsymbol{g}.debug, \boldsymbol{i}.loc)) \;\Rightarrow\; inc\_line(!\boldsymbol{g}.mv, \boldsymbol{tmp})$
$end(\_) \;\Rightarrow\; io\_lines(\boldsymbol{g}.mv)$

I′ *get_src_line* finds the source line in the debug and symbol information for the given instruction.

lookup finds the entry in $\boldsymbol{g}.mv$ that represents the given source line, creating and adding it if it is not already present, and returns a pointer to it.

inc_line increments the counter for the entry in $\boldsymbol{g}.mv$ pointed to by $\boldsymbol{tmp}$.

io_lines prints the final per-line counts to file, augmenting any previous coverage information already present from previous runs. This information can be used to annotate source code.

The debug information look-up done by *get_src_line* is not fast, which is why it pays to do it at instrumentation-time, once per instruction, rather than at run-time.

If only pure coverage was needed, $\boldsymbol{g}.mv$ could be changed to hold a boolean per source line, and *inc_line* changed to set the appropriate boolean when $\boldsymbol{i}$ is executed.

The metadata could be tracked at the level of instructions, rather than the level of source code lines. However, since the metadata is to be used to annotate source code, there is no point tracking it at instruction level, which would only increase the number of entries in $\boldsymbol{g}.mv$ and require multiple instruction counts to be collapsed into multiple line counts during annotation.

A similar tool is a dynamic invariant detector, such as Daikon [42] or DIDUCE [49]. The following description is of a generic invariant detector.

M $\boldsymbol{g}.mv$

M′ $\boldsymbol{g}.mv$ is a table holding invariant information for all the "interesting" source code lines.

I $instr(\boldsymbol{i}) : is\_interesting(\boldsymbol{i}), \boldsymbol{tmp} := lookup^{\dagger}(!\boldsymbol{g}.mv, get\_src\_line^{\dagger}(\boldsymbol{g}.debug, \boldsymbol{i}.loc)) \;\Rightarrow$
$\quad invar(!\boldsymbol{g}.mv, \boldsymbol{tmp}, \boldsymbol{i}.us\langle n\rangle.v)$
$end(\_) \;\Rightarrow\; io\_invars(\boldsymbol{g}.mv)$

I′ *is_interesting* succeeds if the instruction should have its invariants tracked.

invar updates the invariant information for the node in $\boldsymbol{g}.mv$ pointed to by $\boldsymbol{tmp}$, using the values used by the instruction.

io_invars prints out all the gathered invariants.

Note that *lookup* and *get_src_line* are not described in the **I′**-part; this is because they appeared identically in an earlier description. Predicates, functions and I/O actions repeatedly used like this will be marked with a † symbol. This is not a feature of the descriptions, but is purely to avoid repetition in this text. Also note that $\boldsymbol{i}.us\langle n\rangle.v$ here is a list of values, not just a single value; and that the comma on the left-hand side of the first rule is just a separator that indicates sequencing.

This description is very vague; for example, there is no detail about the form of the invariants. It would be much more precise if it were describing a specific invariant detector. It is interesting to note that the vagueness is all in the informal **M′**- and **I′**-parts, which demonstrates their flexibility.

6.5.3 Tools Using Per-Location Metadata

Consider the following tool which counts how many times each register is accessed.

M $\boldsymbol{r}\langle 4\rangle.mv$

M′ $\boldsymbol{r}\langle 4\rangle.mv$ is an integer, initialised to zero.

I $\quad$ *instr*(***i***) : *is_not_empty*(***i***.*rs*$\langle 4\rangle$) $\;\Rightarrow\;$ *incs*(!***i***.*rs*$\langle 4\rangle$.*mv*)

I′ $\quad$ *is_not_empty* succeeds if the list is not empty.

incs increments the counter(s).

A similar tool counts how many times each page of memory (assumed to be 4KB) is accessed.

M $\quad$ ***m***$\langle 4096\rangle$.*mv*

M′ $\quad$ ***m***$\langle 4096\rangle$.*mv* is an integer, initialised to zero.

I $\quad$ *instr*(***i***) : *is_not_empty*†(***i***.*ms*$\langle 4096\rangle$) $\;\Rightarrow\;$ *incs*†(!***i***.*ms*$\langle 4096\rangle$.*mv*)

Recall that n-byte MemLocs are n-aligned, so each ***m***$\langle 4096\rangle$.*mv* corresponds exactly to a page of memory, and ***i***.*ms*$\langle 4096\rangle$ names the exact memory pages touched by ***i***. Unaligned accesses are possible on the x86, so a single memory access can touch two pages; in that case both pages will be included in ***i***.*ms*$\langle 4096\rangle$.

Note also that ***i***.*ms*$\langle 4096\rangle$ has a mixed static/dynamic binding time. For each instruction, it is known at instrumentation-time whether it accesses any memory pages, and thus the predicate *is_not_empty* can be decided then. But the actual number of pages accessed, and their identities, are not known until run-time.

6.5.4 Tools Using Per-Value Metadata

Section 2.4.4 introduced the idea of shadow computation, whereby every value is shadowed with a metavalue, and shadow operations are used to propagate the metavalues. The following rule describes the generic approach.

M $\quad$ ***v***$\langle n\rangle$.*mv*

M′ $\quad$ ***v***$\langle n\rangle$.*mv* says something about the value. Initialised to an "uninitialised" value.

const_value_mv returns an appropriate metavalue for code constants.

I $\quad$ *instr*(***i***) : *is_not_empty*†(***i***.*ds*$\langle n\rangle$), ***tmp*** := *f*(***i***.*opcode*) $\;\Rightarrow\;$ ***i***.*ds*$\langle n\rangle$.*mv* := ***tmp***(***i***.*us*$\langle n\rangle$.*mv*)

I′ $\quad$ *f* returns one of a family of functions that compute metavalues outputs from metavalue inputs.

This description is vague because it is describing shadow computation in general. Section 6.7 has more examples of descriptions using shadow computation.

6.6 Custom I-hooks

All the examples so far have been dominated by *instr* I-hooks. All passive instrumentation can be described using the built-in I-hooks, but for more complex tools it is very useful to hide certain messy details in the descriptions by raising the level of abstraction. This is done by introducing custom I-hooks for describing specific kinds of instructions and common aggregates such as system calls and memory operations. Custom I-hooks are also the means for describing function replacement, a form of semantics-changing, active analysis code that most Valgrind tools use.

6.6.1 A Simple Example

Custom I-hooks look like normal tool descriptions, except that they are augmented by a **C**-part and possibly a **C′**-part. Section 6.3 explained that the *instr* I-hook includes all instructions within the operating system executed on behalf of a client program. The following custom I-hook is a modified version of *instr* that describes the instrumentation of only the instructions outside the operating system kernel.

C $\quad user\_instr(\_) : \boldsymbol{X} \ \Rightarrow\ \boldsymbol{Y}$

I $\quad instr(\boldsymbol{i}) : is\_user\_instr(\boldsymbol{i}), \boldsymbol{X} \ \Rightarrow\ \boldsymbol{Y}$

I′ $\quad$ *is_user_instr* succeeds for user-level (non-kernel) instructions.

A custom I-hook is a bit like a macro in C. The **C**-part names the custom I-hook. $\boldsymbol{X}$ is a parameter that represents the left-hand side's predicates and assignments, and $\boldsymbol{Y}$ is a parameter that represents the right-hand side's analysis code. $\boldsymbol{X}$ and $\boldsymbol{Y}$ are replaced when the custom I-hook is used, like in macro-expansion; their actual names are not important.

With this custom I-hook definition in place, the following description:

I $\quad user\_instr(\boldsymbol{i}) : true \ \Rightarrow\ io()$

I′ $\quad$ *io* prints a message.

expands out to this:

I $\quad instr(\boldsymbol{i}) : is\_user\_instr(\boldsymbol{i}), true \ \Rightarrow\ io()$

I′ $\quad$ *is_user_instr* succeeds for user-level (non-kernel) instructions.
io prints a message.

Here the $\boldsymbol{X}$ has been replaced by *true*, the $\boldsymbol{Y}$ has been replaced by *io*(), and the **I′**-parts have been combined.

This I-hook can be composed with any other. Custom I-hooks differ from built-in I-hooks in that they can have **M**-, **M′**- and **I′**-parts associated with them.

This is an interesting I-hook, because of its relevance to Valgrind. Valgrind cannot instrument any kernel code, and so any *instr* rule used in an accurate description of a Valgrind tool must account for this. Thus, all descriptions that follow will use *user_instr* rather than *instr*. Thus the predicates on the left-hand side (e.g. *is_user_instr*) describe what gets instrumented, but do not necessarily represent actual code that is run at instrumentation-time.

6.6.2 System Calls

More complicated custom I-hooks have their own attributes, as the following example shows. It describes the instrumentation of system call entry, as supported by Valgrind.

C $\mathit{syscall\_entry}(\boldsymbol{s}) : \boldsymbol{X} \;\Rightarrow\; \boldsymbol{Y}$

C′ $\boldsymbol{s}.\mathit{name}$::String, $\boldsymbol{s}.\mathit{args}$::List(Value$\langle 4\rangle$)

I $\mathit{user\_instr}(\boldsymbol{i}) : \mathit{is\_wrapping\_syscalls}(), \mathit{is\_syscall\_entry}(\boldsymbol{i}.\mathit{opcode}, \boldsymbol{i}.\mathit{us}\langle 4\rangle) \;\Rightarrow$
$(\boldsymbol{s}.\mathit{name}, \boldsymbol{s}.\mathit{args}) := \mathit{sys\_details}(\boldsymbol{i}.\mathit{rs}\langle 4\rangle.\mathit{contents}),\; \boldsymbol{Y}$

I′ *is_wrapping_syscalls* succeeds if the Valgrind tool has declared that it requires system calls to be wrapped (i.e. that system calls are instrumentable). Tools that do not need to instrument system calls will not make this declaration, in order to avoid the (small) associated run-time overhead.

is_syscall_entry succeeds if the instruction does a system call; on x86/Linux this is done with the instruction `int $0x80`.

sys_details converts the system call number in `%eax` to the system call name, and gets the appropriate number of arguments from the other registers.

New attributes are declared in the **C′**-part, and assigned in the rules. The symbol :: indicates an attribute's type. This example shows that custom attributes are the third thing that can be assigned to within rules; the first two things were metavalues and the special ***tmp*** variable. Static attributes can be assigned on the left-hand side, dynamic attributes on the right-hand side. Note that the comma is just a separator indicating sequencing, so the right-hand side of the rule states that *sys_details* is called, and then ***Y*** takes place (whatever action ***Y*** represents when the custom I-hook is actually used). It is important that ***Y*** is the right-most element of the right-hand side, so the assignments of $\boldsymbol{s}.\mathit{name}$ and $\boldsymbol{s}.\mathit{args}$ come before it and can be used within ***Y***. The I-hook used is *user_instr* rather than *instr* so that this custom I-hook can be reused later to describe Valgrind tools, which only instrument user-level code.

This rule nicely captures the role of the Valgrind core in instrumenting system calls with wrappers; the ***Y*** part here corresponds to the calling of the tool-registered callback function, and the *name* and *args* attributes correspond to the arguments collected by the core and passed to the callback. Since Valgrind's approach does not allow tools to statically ignore particular system calls—any filtering can only be done at run-time—***X*** does not appear in the **I**-part.

Valgrind also allows system call exits to be instrumented, and in such a way that the arguments from the entry are still visible. The following rule expresses this.

C $\mathit{syscall\_exit}(\boldsymbol{s}) : \boldsymbol{X} \;\Rightarrow\; \boldsymbol{Y}$

C′ $\boldsymbol{s}.\mathit{name}$::String, $\boldsymbol{s}.\mathit{args}$::List(Value$\langle 4\rangle$), $\boldsymbol{s}.\mathit{ret}$::Value$\langle 4\rangle$

I $\mathit{syscall\_entry}(\_) \;\Rightarrow\; \_$
$\mathit{user\_instr}(\boldsymbol{i}) : \mathit{is\_wrapping\_syscalls}(), \mathit{is\_syscall\_exit}(\boldsymbol{i}) \;\Rightarrow\; \boldsymbol{s}.\mathit{ret} := \mathit{eax}(\boldsymbol{i}.\mathit{rs}\langle 4\rangle.\mathit{contents}),\; \boldsymbol{Y}$

I′ *is_syscall_exit* succeeds conceptually if the instruction is located immediately after a system call. In Valgrind, the instrumentation is actually done as part of the system call wrapper (described in Section 2.3.10).

eax returns the contents of `%eax`, which holds the return value.

The '_' on the right-hand side of the first rule indicates that nothing is done beyond the implicit gathering of $\boldsymbol{s}.name$ and $\boldsymbol{s}.args$. In the second rule, the $\boldsymbol{X}$ is again not present, again because Valgrind can only do run-time filtering on system calls.

As an example of this I-hook in action, consider the following description of the Unix utility `strace` which prints out the name and arguments of every system call performed by a program.

I $syscall\_entry(\boldsymbol{s}) \;\Rightarrow\; io\_entry(\boldsymbol{s}.name, \boldsymbol{s}.args.v)$

$syscall\_exit(\boldsymbol{s}) \;\Rightarrow\; io\_exit(\boldsymbol{s}.ret.v)$

I′ *io_entry* prints the system call name and arguments.

io_exit prints the system call return value.

Note that merging the two rules into a single *syscall_exit* rule would not be accurate, as no output would be produced for the final `_exit()` or `exit_group()` system calls which never return.

6.6.3 Function Replacement

As Section 2.4.5 explained, Valgrind tools can replace functions in the client with their own custom versions. This is heavy-handed, but is necessary for two reasons. First, it is the best way within Valgrind to instrument specific functions in a way that is reliable and also allows the function arguments to be obtained (as Section 2.4.3 described). Second, sometimes it is necessary to change the behaviour of a function slightly, as in the case of Memcheck which replaces `malloc()` and friends so that it can pad heap blocks with red-zones and postpone the recycling of deallocated memory. Function replacement cannot be described completely by the descriptions, but the following is a good-enough approximation of how it is used by Valgrind tools to instrument function entry.

C $fn\_entry(\boldsymbol{f}) : \boldsymbol{X} \;\Rightarrow\; \boldsymbol{Y}$

C′ $\boldsymbol{f}.name$::String, $\boldsymbol{f}.args$::List(Value$\langle n \rangle$)

I′ All functions matching the predicates in $\boldsymbol{X}$ are replaced; the custom versions set $\boldsymbol{f}.name$ and $\boldsymbol{f}.args$ appropriately, and execute $\boldsymbol{Y}$ on function entry.

No **I**-part is given, because it cannot be expressed in the standard form. To make up for this, in addition to its normal contents the **I′**-part should also explain if the replacement function is meant to be instrumented by other rules or not (for Memcheck, some are and some are not), and also if the replacement version differs in any significant way from the original.

Function exits are described similarly, as the following shows.

C $\textit{fn\_exit}(\boldsymbol{f}) : \boldsymbol{X} \quad \Rightarrow \quad \boldsymbol{Y}$

C′ $\boldsymbol{f}.\textit{name}$::String, $\boldsymbol{f}.\textit{args}$::List(Value$\langle n\rangle$), $\boldsymbol{f}.\textit{ret}$::Maybe(Value$\langle n\rangle$)

I′ All functions matching the predicates in $\boldsymbol{X}$ are replaced; the custom versions set $\boldsymbol{f}.\textit{name}$, $\boldsymbol{f}.\textit{args}$ and $\boldsymbol{f}.\textit{ret}$ appropriately, and execute $\boldsymbol{Y}$ on function exit.

Note that the size of arguments and the return value is n; this will usually be word-sized.

This treatment of function replacement is somewhat awkward. Why not just make replacement functions a proper part of the description system? Basically, because they are a fairly Valgrind-specific feature. Besides, the ugliness here is only local, since the *fn_entry* and *fn_exit* I-hooks can be used in other descriptions in a similar way to other custom I-hooks—i.e. the intent and result is the same as that of the *syscall_entry* and *syscall_exit* I-hooks.

6.6.4 Memory Operations

Many tools need to know about memory operations—allocations, deallocations, and other operations, for heap, stack and mapped (including static) memory. The following definition defines an I-hook representing all these operations in a useful way. This description will be used in the descriptions for the Valgrind tools in Section 6.7. The description has several parts, and is broken up with explanatory text; it also omits, for brevity, some parts that are repetitive and very similar to other parts.

This definition introduces four attributes. The attribute *opname* usefully classifies the operations, e.g. “new_mem_stack” (for stack allocation) or “die_mem_heap” (for heap deallocation). The attributes *addr* and *size* indicate the location and size of the memory block affected by the operation. The attribute $\textit{ms}\langle n\rangle$ expresses this in a different way, i.e. the list of memory locations within the affected block; it is redundant with respect to *addr* and *size*, but is useful for some descriptions.

C $\textit{memop}(\boldsymbol{m}) : \boldsymbol{X} \quad \Rightarrow \quad \boldsymbol{Y}$

C′ $\boldsymbol{m}.\textit{opname}$::String, $\boldsymbol{m}.\textit{addr}$::Addr, $\boldsymbol{m}.\textit{size}$::Int, $\boldsymbol{m}.\textit{ms}\langle n\rangle$::List(MemLoc$\langle n\rangle$)

The first rule relates to the stack. The I-hook *user_instr* is used to avoid considering kernel code. The rule for stack growth is given, but that for stack shrinkage is omitted; it is extremely similar, the only notable difference is that the *opname* is “die_mem_stack”. Recall that on x86 the stack grows towards zero.

I $\textit{user\_instr}(\boldsymbol{i}) : \textit{is\_stack\_growth}(\boldsymbol{i}), \boldsymbol{m}.\textit{opname} :=$ “new_mem_stack”, $\boldsymbol{X} \quad \Rightarrow$
    $\boldsymbol{m}.\textit{addr} := \textit{new\_stack\_top}(\boldsymbol{i}.\textit{rs}\langle n\rangle)$,
    $\boldsymbol{m}.\textit{size} := \textit{growth\_size}(\boldsymbol{i}.\textit{opcode}, \boldsymbol{i}.\textit{rs}\langle n\rangle)$,
    $\boldsymbol{m}.\textit{ms}\langle n\rangle := \textit{mem\_locs}(\boldsymbol{m}.\textit{addr}, \boldsymbol{m}.\textit{size}, \boldsymbol{g}.\textit{ms}\langle n\rangle)$,
    $\boldsymbol{Y}$

I′ *is_stack_growth* succeeds if the instruction grows the stack.

new_stack_top returns the top of the newly grown stack.

growth_size returns the number of bytes the stack grew by.

mem_locs selects the memory byte locations within the given range.

The next part relates to the heap. It involves conditionally defined attributes and conditional use of the $\boldsymbol{Y}$ part.

M $\boldsymbol{g}.mv$

M′ $\boldsymbol{g}.mv$ records the location and size of all blocks in the heap. It is initially empty.

I $fn\_exit(\boldsymbol{f}) : is\_heap\_alloc(\boldsymbol{f}.name), \boldsymbol{m}.opname :=$ "new_mem_heap", $\boldsymbol{X} \Rightarrow$
 $if\ is\_non\_zero(\boldsymbol{f}.ret.v)\ \{$
  $\boldsymbol{m}.addr := get\_val(\boldsymbol{f}.ret.v),$
  $\boldsymbol{m}.size := get\_arg(1, \boldsymbol{f}.args.v),$
  $\boldsymbol{m}.ms\langle n\rangle := mem\_locs(\boldsymbol{m}.addr, \boldsymbol{m}.size, \boldsymbol{g}.ms\langle n\rangle),$
  $add\_block(!\boldsymbol{g}.mv, \boldsymbol{m}.addr, \boldsymbol{m}.size),$
  $\boldsymbol{Y}\ \}$

I′ *is_heap_alloc* succeeds for `malloc()`, `new`, `new[]`.

is_non_zero succeeds if its argument is non-zero.

get_val extracts the return value from the Maybe type.

get_arg extracts the nth argument of the list.

add_block records the details of the allocated block in the heap state.

The replacement allocation functions are not themselves instrumented. This is so that that any memory-allocating system calls (usually `mmap()` or `brk()`) used within `malloc()` are ignored; this is to avoid considering any memory allocated within `malloc()` by the allocator itself until it has been returned to the client by `malloc()`. The replacement allocation functions differ from the originals in that they pad the ends of heap blocks with red-zones.

I $fn\_exit(\boldsymbol{f}) : is\_heap\_dealloc(\boldsymbol{f}.name), \boldsymbol{m}.opname :=$ "die_mem_heap", $\boldsymbol{X} \Rightarrow$
 $\boldsymbol{m}.addr := get\_arg(1, \boldsymbol{f}.args.v),$
 $if\ is\_known\_block(\boldsymbol{g}.mv, \boldsymbol{m}.addr)\ \{$
  $\boldsymbol{m}.size := get\_size(\boldsymbol{g}.mv),$
  $\boldsymbol{m}.ms\langle n\rangle := mem\_locs(\boldsymbol{m}.addr, \boldsymbol{m}.size, \boldsymbol{g}.ms\langle n\rangle),$
  $remove\_block(!\boldsymbol{g}.mv, \boldsymbol{m}.addr),$
  $\boldsymbol{Y}\ \}$

I′ *is_heap_dealloc* succeeds for `free()`, `delete`, `delete[]`.

is_known_block succeeds if the pointer given points to an existing heap block.

get_size obtains the block size from the global heap state.

remove_block removes the details of the freed block from the heap state.

> The replacement deallocation functions are not themselves instrumented, for the same reasons that the allocation functions are not.

This is an example of a custom I-hook that introduces some metadata. This metadata is not shown when the I-hook is used within another rule, however, as as an example will soon show.

The rules for `realloc()` are omitted, but they are similar to those shown for heap allocations and deallocations.

The last rules are for mapped memory segments. As before, the rules for `munmap()`, `mremap()`, `mprotect()` and `brk()` are omitted, but similar.

I $\quad$ $\mathit{syscall\_exit}(\boldsymbol{s}) \;\Rightarrow$
$\quad$ $\mathit{if}\ \mathit{is\_equal}(\boldsymbol{s}.\mathit{name}, \text{“mmap”})\ \{$
$\quad\quad$ $\boldsymbol{m}.\mathit{opname} := \text{“new\_mem\_mmap”},$
$\quad\quad$ $\mathit{if}\ (\boldsymbol{X} \wedge \mathit{is\_non\_error}(\boldsymbol{f}.\mathit{ret}.v))\ \{$
$\quad\quad\quad$ $\boldsymbol{m}.\mathit{addr} := \mathit{get\_val}(\boldsymbol{s}.\mathit{ret}),$
$\quad\quad\quad$ $\boldsymbol{m}.\mathit{size} := \mathit{get\_arg}(2, \boldsymbol{s}.\mathit{args}.v),$
$\quad\quad\quad$ $\boldsymbol{m}.\mathit{ms}\langle n\rangle := \mathit{mem\_locs}(\boldsymbol{m}.\mathit{addr}, \boldsymbol{m}.\mathit{size}, \boldsymbol{g}.\mathit{ms}\langle n\rangle),$
$\quad\quad\quad$ $\boldsymbol{Y}\ \}\ \ \}$

I′ $\quad$ *is_non_error* succeeds if the return value is not -1, which indicates that the call failed.

Note that the $\boldsymbol{X}$, which is normally on the left-hand side of custom I-hooks, is on the right-hand side. This is again because no static predicates can be used when instrumenting system calls.

That ends the definition of the custom I-hook *memop*. As an example of the use of *memop* under Valgrind, consider a heap profiler that produces a graph plotting heap size against time for a program's entire execution. It would have the following description.

M $\quad$ $\boldsymbol{g}.\mathit{mv}_h, \boldsymbol{g}.\mathit{mv}_s$

M′ $\quad$ $\boldsymbol{g}.\mathit{mv}_h$ is an integer, initially zero.

$\boldsymbol{g}.\mathit{mv}_s$ is a sequence of (time, heap-size) integer pairs, initially empty.

I $\quad$ $\mathit{memop}(\boldsymbol{m}) : \mathit{is\_heap\_op}(\boldsymbol{m}.\mathit{opname}) \;\Rightarrow\; \mathit{heap\_census}(!\boldsymbol{g}.\mathit{mv}_h, !\boldsymbol{g}.\mathit{mv}_s, \boldsymbol{m}.\mathit{opname}, \boldsymbol{m}.\mathit{size}, \boldsymbol{g}.t)$
$\quad$ $\mathit{end}(\_) \;\Rightarrow\; \mathit{io}(\boldsymbol{g}.\mathit{mv}_s)$

I′ $\quad$ *is_heap_op* succeeds for any heap allocation or deallocation.

heap_census updates the current heap size (growing or shrinking depending on $\boldsymbol{m}.\mathit{opname}$) and records the current (time, heap-size) pair.

io prints a graph of the time-varying heap profile.

These custom I-hooks are quite ugly. The good news is that they encapsulate and hide the ugliness in such a way that the Valgrind tool descriptions in the next section are quite clean.

6.7 Descriptions of Valgrind Tools

This section gives full descriptions for the Valgrind tools described in Chapters 2–5. It shows how the descriptions can be used to describe real, complex DBA tools.

6.7.1 Memcheck

Memcheck is a comprehensive memory checker. It was briefly described in Section 2.4.1. It tracks multiple kinds of metadata, does multiple kinds of checking, and is arguably the most complex tool implemented with Valgrind. So it is a good test for the description system. To make the description easier to read, it is broken into four parts.

Addressability Checker This part uses A (addressability) bits to detect accesses of unaddressable memory, which may be done by normal memory accesses, when jumping to new locations, and in memory blocks read by system calls.

M $\boldsymbol{m}\langle 1\rangle.mv$

M′ $\boldsymbol{m}\langle 1\rangle.mv$ is a boolean; the initial value is *false* ("unaddressable").

I $memop(\boldsymbol{m}) \;\Rightarrow\; \boldsymbol{m}.ms\langle 1\rangle.mv := update\_A\_bits(\boldsymbol{m}.opname)$
$user\_instr(\boldsymbol{i}) : is\_not\_empty^{\dagger}(\boldsymbol{i}.ms\langle 1\rangle) \;\Rightarrow$
  $\mathit{if}\ is\_unaddr(\boldsymbol{i}.ms\langle 1\rangle.mv)\ \{\ io\_warn_1(\boldsymbol{g}.debug, \boldsymbol{i}.loc, \boldsymbol{i}.ms\langle 1\rangle.addr)\ \}$
$user\_instr(\boldsymbol{i}) : is\_jmp^{\dagger}(\boldsymbol{i}.opcode) \;\Rightarrow$
  $\mathit{if}\ is\_unaddr(jmp\_target\_memloc(\boldsymbol{i}.us\langle 4\rangle).mv)\ \{\ io\_warn_2(\boldsymbol{g}.debug, \boldsymbol{i}.loc)\ \}$
$syscall\_entry(\boldsymbol{s}) \;\Rightarrow$
  $\mathit{if}\ is\_mem\_touching\_syscall(\boldsymbol{s}.name)\ \{$
    $\mathit{if}\ is\_unaddr(sys\_touched\_memlocs(\boldsymbol{s}.name, \boldsymbol{s}.args.v, \boldsymbol{g}.ms\langle 1\rangle))\ \{$
      $io\_warn_3(\boldsymbol{g}.debug, \boldsymbol{i}.loc, \boldsymbol{s}.name, \boldsymbol{s}.args.v)\ \}\ \}$

I′ *update_A_bits* returns *true* if the operation makes the memory addressable, or *false* otherwise.

is_unaddr succeeds if any of its arguments are *false* (i.e. unaddressable).

io_warn_1, io_warn_2 and io_warn_3 print warnings about accesses to unaddressable memory; they use the code location and debug information to print the message in terms of source code locations.

jmp_target_memloc returns the memory location being jumped to.

is_mem_touching_syscall succeeds if the system call touches any memory blocks through pointers, e.g. `read()` or `write()`.

sys_touched_memlocs returns the byte-sized memory locations touched by the system call, as determined from the arguments.

The replacement allocation and deallocation functions are not instrumented.

The replacement deallocation functions add red-zones to heap blocks, and postpone memory recycling to improve the chances of catching use-after-free errors.

Validity Checker This part uses V (validity) bits to detect dangerous uses of uninitialised values—in the target addresses of jumps, in the conditions of conditional jumps, in the addresses of loads and stores, and in memory blocks touched by system calls. V bits are set by shadow computation, and by system calls that write blocks of memory.

M $\boldsymbol{v}\langle 1\rangle.mv$

M′ $\boldsymbol{v}\langle 1\rangle.mv$ is a byte, where each bit indicates whether the corresponding bit in the value is valid (i.e. has been initialised), and is initialised to "invalid".

const_value_mv returns a value indicating all the bits of a constant are valid.

I $user\_instr : is\_not\_empty^{\dagger}(\boldsymbol{i}.ds\langle 1\rangle), \boldsymbol{tmp} := f(\boldsymbol{i}.opcode) \;\Rightarrow\; \boldsymbol{i}.ds\langle 1\rangle.mv := \boldsymbol{tmp}(\boldsymbol{i}.us\langle 1\rangle.mv)$
$user\_instr(\boldsymbol{i}) : is\_jmp^{\dagger}(\boldsymbol{i}.opcode) \;\Rightarrow$
  $if\ is\_uninit(jmp\_target\_reg(\boldsymbol{i}.rs\langle 1\rangle).contents.mv)\ \{\ io\_warn_1(\boldsymbol{g}.debug, \boldsymbol{i}.loc)\ \}$
$user\_instr(\boldsymbol{i}) : is\_cond\_jmp(\boldsymbol{i}.opcode) \;\Rightarrow$
  $if\ is\_uninit(eflags(\boldsymbol{i}.rs\langle 1\rangle).contents.mv)\ \{\ io\_warn_2(\boldsymbol{g}.debug, \boldsymbol{i}.loc)\ \}$
$user\_instr(\boldsymbol{i}) : is\_not\_empty(\boldsymbol{i}.ms\langle 1\rangle) \;\Rightarrow$
  $if\ is\_uninit(addrs(\boldsymbol{i}.as\langle 1\rangle).mv)\ \{\ io\_warn_3(\boldsymbol{g}.debug, \boldsymbol{i}.loc, \boldsymbol{i}.dsize)\ \}$
$syscall\_entry(\boldsymbol{s}) \;\Rightarrow$
  $if\ is\_mem\_reading\_syscall(\boldsymbol{s}.opname)\ \{$
    $if\ is\_uninit(sys\_read\_memlocs(\boldsymbol{s}.name, \boldsymbol{s}.args.v, \boldsymbol{g}.ms\langle 1\rangle))\ \{$
      $io\_warn_4(\boldsymbol{g}.debug, \boldsymbol{i}.loc, \boldsymbol{s}.name, \boldsymbol{s}.args.v)\ \}\ \}$
$syscall\_entry(\boldsymbol{s}) \;\Rightarrow$
  $if\ is\_mem\_writing\_syscall(\boldsymbol{s}.opname)\ \{$
    $sys\_written\_memlocs(\boldsymbol{s}.name, \boldsymbol{s}.args.v, \boldsymbol{g}.ms\langle 1\rangle).contents.mv := valid()\ \}$

I′ f returns one of the family of shadow computation update functions. They are mostly implemented as inline code for efficiency. They propagate appropriate validity metavalues; the exact of these updates details are beyond the scope of this description.

jmp_target_reg returns the location bytes of the register specifying the jump target address.

is_uninit succeeds if any arguments are invalid.

io_warn_1–io_warn_4 print appropriate warnings about invalid bits being used.

is_cond_jmp succeeds if the instruction is a conditional jump or conditional move.

eflags returns the location bytes of `%eflags`.

addrs selects the bytes holding the address(es) of the accessed memory location(s).

is_mem_reading_syscall succeeds if the system call reads any memory blocks through pointers, e.g. `write()`.

is_mem_writing_syscall succeeds if the system call writes any memory blocks through pointers, e.g. `read()`.

sys_read_memlocs returns the memory location bytes read by the system call, as determined from the arguments.

sys_written_memlocs returns the memory location bytes written by the system call, as determined from the arguments.

valid returns a fully valid byte metavalue.

The shadow computation updates could certainly be described in more detail. However, omitting the details in this case is no problem, as their workings are a detail that is not needed to understand the basics of how the tool works.

Note that Valgrind's use of UCode is not important here; that is a detail below the descriptions' level of abstraction. This is a good thing, as it shows the descriptions' generality.

Deallocation and Leak Checker This part uses the tracked heap state to detect bad deallocations and search for leaked heap blocks.

M $\boldsymbol{g}.mv$

M′ $\boldsymbol{g}.mv$ augments the information about each heap block maintained for the *memop* custom I-hook with the name of the allocating function (e.g. `malloc()`, `new`, `new[]`).

I $memop(\boldsymbol{m}) : is\_heap\_op^{\dagger}(\boldsymbol{m}.opname) \ \Rightarrow \ heap\_block(!\boldsymbol{g}.mv, \boldsymbol{m}.opname, \boldsymbol{m}.addr, \boldsymbol{m}.size)$
$memop(\boldsymbol{m}) : is\_free\_op(\boldsymbol{m}.opname) \ \Rightarrow$
$\quad if\ is\_no\_such\_block(\boldsymbol{g}.mv, \boldsymbol{m}.addr)\ \{\ io\_bad\_free(\boldsymbol{g}.debug, \boldsymbol{i}.loc, \boldsymbol{m}.opname, \boldsymbol{m}.addr)\ \},$
$\quad if\ is\_mismatched(\boldsymbol{g}.mv, \boldsymbol{m}.opname, \boldsymbol{m}.addr)\ \{$
$\quad\quad io\_mismatched(\boldsymbol{g}.debug, \boldsymbol{i}.loc, \boldsymbol{m}.opname, \boldsymbol{m}.addr, \boldsymbol{g}.mv)\ \}$
$end(\_) \ \Rightarrow \ io\_leaks(\boldsymbol{g}.mv, \boldsymbol{g}.ms\langle 1\rangle.mv, \boldsymbol{g}.ms\langle 4\rangle.contents.v)$

I′ *heap_block* updates the record of the current heap state; it is effectively an augmented version of the *add_block* and *remove_block* functions used in the definition of the *memop* I-hook in Section 6.6.4.

is_free_op succeeds for any heap deallocation.

is_no_such_block succeeds if the given address does not match an existing heap block.

io_bad_free prints a warning about the bad deallocation, including the passed address.

is_mismatched succeeds if the deallocation function is not appropriate for the block.

io_mismatched prints a warning about the mismatched deallocation, including the address and size of the block, and what the used allocation/deallocation functions were.

io_leaks prints the memory leak summary. It detects and reports any heap blocks that have not been freed yet, and for which no pointers remain in memory (heap blocks that could have been freed but were not are considered acceptable). It does this by finding still-addressable locations that point to unfreed heap blocks; blocks that have no pointers to them have definitely leaked. Note that there is some interaction here with the addressability checking part: the A bits are used here to determine which parts of memory should be scanned for pointers. If no leak occurred, a message explaining this is printed.

Overlap Checker This part checks functions like `memcpy()` and `strcpy()` for overlaps between the source and destination. It does not use any metadata.

I $\mathit{fn\_entry}(\boldsymbol{f}) : \mathit{is\_strmem\_fn}(\boldsymbol{f}.\mathit{name}) \;\Rightarrow$
$\quad \mathit{if}\ \mathit{is\_overlap}(\boldsymbol{f}.\mathit{name}, \boldsymbol{f}.\mathit{args}.v, \boldsymbol{g}.\mathit{ms}\langle 1\rangle)\ \{\ \mathit{io\_overlap}(\boldsymbol{f}.\mathit{name}, \boldsymbol{f}.\mathit{args}.v)\ \}$

I′ *is_strmem_fn* succeeds if the function needs overlap checking.

is_overlap determines if there is an overlap. For `memcpy()`, this can be determined just by looking at the arguments. For the rest (e.g. `strcpy()`), it requires looking at the contents of the two blocks.

io_overlap prints a warning about an overlap, including the function name and the memory block addresses.

The replacement functions are instrumented as normal. Apart from the overlap checking, their function is identical to that of the functions they replace.

6.7.2 Addrcheck

Addrcheck is a cut-down version of Memcheck that does not perform validity checking. It was briefly described along with Memcheck in Section 2.4.1. Its description is exactly the same as Memcheck's, from Section 6.7.1, minus the second validity checker part.

In general, the descriptions make it easy to compare tools, and to tell if a tool is subsumed by another. As another example, one could create a cut-down version of Addrcheck that does not do deallocation and leak checking, and the relationship of this new version with Memcheck and Addrcheck would be immediately obvious from the description.

6.7.3 Cachegrind

Cachegrind performs detailed cache profiling. It was described in Chapter 3.

M $\boldsymbol{g}.\mathit{mv}_s, \boldsymbol{g}.\mathit{mv}_c, \boldsymbol{g}.\mathit{mv}_i$

M′ $\boldsymbol{g}.\mathit{mv}_s$ is the simulation of the state of the I1, D1 and L2 caches, initially empty. The cache configuration is found using the `cpuid` instruction, unless a different configuration is specified from the command line.

$\boldsymbol{g}.\mathit{mv}_c$ is the cost centre table, a table holding one cost centre (CC) per source code line. Each line CC holds various integer counters, all initialised to zero: accesses, instruction read misses, data read misses, and data write misses, for each of the I1, D1 and L2 caches. The table is structured by file, function and line. It has an "unknown" CC for lines that do not have any source code information. It is initially empty.

$\boldsymbol{g}.mv_i$ is the instr-info table, a table caching the statically known information about each instrumented instruction. It is initially empty.

I $\quad$ *user_instr*($\boldsymbol{i}$) : *is_empty*($\boldsymbol{i}.ms\langle 1\rangle$),

$\quad\quad$ $\boldsymbol{tmp} := setup\_instr(!\boldsymbol{g}.mv_c,\ !\boldsymbol{g}.mv_i,\ \boldsymbol{i}.addr,\ \boldsymbol{i}.isize,\ 0,\ get\_src\_line^{\dagger}(\boldsymbol{g}.debug,\ \boldsymbol{i}.loc)) \;\Rightarrow$

$\quad\quad\quad$ $non\_mem\_instr(!\boldsymbol{g}.mv_s,\ !\boldsymbol{g}.mv_c,\ \boldsymbol{tmp})$

user_instr($\boldsymbol{i}$) : *is_not_empty*†($\boldsymbol{i}.ms\langle 1\rangle$),

$\quad\quad$ $\boldsymbol{tmp} := setup\_instr(!\boldsymbol{g}.mv_c,\ !\boldsymbol{g}.mv_i,\ \boldsymbol{i}.addr,\ \boldsymbol{i}.isize,\ \boldsymbol{i}.dsize,\ get\_src\_line^{\dagger}(\boldsymbol{g}.debug,\ \boldsymbol{i}.loc)) \;\Rightarrow$

$\quad\quad\quad$ $mem\_instr(!\boldsymbol{g}.mv_s,\ !\boldsymbol{g}.mv_c,\ \boldsymbol{tmp},\ \boldsymbol{i}.ms\langle 1\rangle)$

$end(\_) \;\Rightarrow\; io(\boldsymbol{g}.mv_c)$

I′ $\quad$ *is_empty* succeeds if the list is empty.

setup_instr finds the CC node in $\boldsymbol{g}.mv_c$ that represents the source location of the instruction, and adds the CC if it does not already exist. It then caches the statically known information about the instruction—the instruction size, instruction address, data size, and a pointer to the found CC—in the instr-info table, and returns a pointer to this node of cached information, which is assigned to ***tmp***.

non_mem_instr is used for instructions not accessing memory: it updates the global I-cache state and L2 state, determines if the access hit or missed, and updates the I1 and L2 hit/miss counters in $\boldsymbol{g}.mv_c$ for the instruction's line. The relevant instruction information is passed via ***tmp***, which represents the pointer to the instruction's node in the instr-info table.

mem_instr does the same for memory-accessing instructions, and it also does the appropriate updates for the D1 and L2 data accesses.

io prints out the summary hit/miss statistics (found by summing the hit/miss counts of every cost centre), and also does a per-source-line dump of hit/miss counts, which can be used to annotate the original source code.

This is an example where breaking the global metavalue into parts makes a description clearer.

6.7.4 Annelid

Annelid performs bounds-checking. It was described in Chapter 4.

M $\quad$ $\boldsymbol{g}.mv,\ \boldsymbol{v}\langle 4\rangle.mv$

M′ $\quad$ $\boldsymbol{g}.mv$ is the table of segment structures. It is initially empty.

$\boldsymbol{v}\langle 4\rangle.mv$ is the run-time type, initially set to `UNKNOWN`.

const_value_mv returns `NONPTR` for small constants, and `UNKNOWN` for constants that could be pointers, unless the value points to within a known segment, in which case it will return a pointer to that segment's structure.

I $\textit{memop}(\boldsymbol{m}) : \textit{is\_heap\_op}^{\dagger}(\boldsymbol{m}.\textit{opname}) \;\;\Rightarrow$
$(\boldsymbol{m}.\textit{ms}\langle 4\rangle.\textit{contents}.\textit{mv}, \boldsymbol{m}.\textit{ret}.\textit{mv}) := \textit{heap\_op}(!\boldsymbol{g}.\textit{mv}, \boldsymbol{m}.\textit{opname}, \boldsymbol{m}.\textit{addr}, \boldsymbol{m}.\textit{size})$
$\textit{memop}(\boldsymbol{m}) : \textit{is\_non\_heap\_op}(\boldsymbol{m}.\textit{opname}) \;\;\Rightarrow$
$\boldsymbol{m}.\textit{ms}\langle 4\rangle.\textit{contents}.\textit{mv} := \textit{non\_heap\_op}(!\boldsymbol{g}.\textit{mv}, \boldsymbol{m}.\textit{opname}, \boldsymbol{m}.\textit{ms}\langle 4\rangle.\textit{contents}.\textit{v})$
$\textit{user\_instr}(\boldsymbol{i}) : \textit{is\_not\_empty}^{\dagger}(\boldsymbol{i}.\textit{ds}\langle 4\rangle), \boldsymbol{tmp} := f(\boldsymbol{i}.\textit{opcode}) \;\;\Rightarrow\;\; \boldsymbol{i}.\textit{ds}\langle 4\rangle.\textit{mv} := \boldsymbol{tmp}(\boldsymbol{i}.\textit{us}\langle 4\rangle.\textit{mv})$
$\textit{user\_instr}(\boldsymbol{i}) : \textit{is\_not\_empty}^{\dagger}(\boldsymbol{i}.\textit{ms}\langle 4\rangle) \;\;\Rightarrow$
$\textit{if } \textit{is\_out\_of\_bounds}(\boldsymbol{i}.\textit{as}\langle 4\rangle.\textit{mv}, \boldsymbol{i}.\textit{ms}\langle 4\rangle.\textit{addr}) \; \{$
$\textit{io\_warn}(\boldsymbol{g}.\textit{debug}, \boldsymbol{i}.\textit{loc}, \boldsymbol{i}.\textit{dsize}, \boldsymbol{i}.\textit{ms}\langle 4\rangle.\textit{addr}) \; \}$

I′ *heap_op* does three metadata updates. First, in the global segment table it creates a new segment or marks an old one as free. Second, it sets the shadow type for each word within the newly allocated/deallocated segment (to `NONPTR`). Third, if the operation is an allocation, the metavalue of the register holding the return value is set to point to the just-created segment structure.

is_non_heap_op succeeds for memory operations that do not involve the heap.

non_heap_op sets the shadow type for each word in the newly allocated/deallocated segment to `NONPTR` or `UNKNOWN` (depending on the range). If the operation is an `mmap()` of a code segment with debug information, segment structures for any arrays mentioned are added to the global table; if it was an `munmap()`, any such segment structures are marked as freed.

f returns one of the family of shadow computation update functions that propagate the run-time type metavalues.

is_out_of_bounds fails if the shadow run-time type of a pointer used to access memory shows the access is out-of-bounds.

io_warn prints an out-of-bounds error message.

6.7.5 Redux

Redux records dynamic data flow graphs (DDFGs). It was described in Chapter 5.

M $\boldsymbol{g}.\textit{mv}_t, \boldsymbol{g}.\textit{mv}_s, \boldsymbol{v}\langle 4\rangle.\textit{mv}$

M′ $\boldsymbol{g}.\textit{mv}_t$ is the full DDFG, initially empty except for the special nodes such as the "undefined" node.

$\boldsymbol{g}.\textit{mv}_s$ is the table of pointers to DDFG nodes, one for each system call executed. It is initially empty.

$\boldsymbol{v}\langle 4\rangle.\textit{mv}$ is a pointer into the DDFG, initially pointing to the "undefined" node.

const_value_mv returns a DDFG node representing the constant; each code constant's node is created once at instrumentation-time.

I $\textit{user\_instr}(\boldsymbol{i}) : \textit{is\_not\_empty}^{\dagger}(\boldsymbol{i}.\textit{ds}\langle 4\rangle), \boldsymbol{tmp} := f(\boldsymbol{i}.\textit{opcode}) \;\;\Rightarrow$
$\boldsymbol{i}.\textit{ds}\langle 4\rangle.\textit{mv} := \boldsymbol{tmp}(!\boldsymbol{g}.\textit{mv}_t, \boldsymbol{i}.\textit{us}\langle 4\rangle.\textit{mv}, \boldsymbol{i}.\textit{us}\langle 4\rangle.\textit{v})$

$syscall\_exit(\boldsymbol{s}) \;\Rightarrow$
$\quad(sys\_written\_memlocs^{\dagger}(\boldsymbol{s}.name, \boldsymbol{s}.args.v, \boldsymbol{g}.ms\langle 4\rangle).contents.mv, \boldsymbol{s}.ret.mv) :=$
$\qquad syscall\_node(!\boldsymbol{g}.mv_t, !\boldsymbol{g}.mv_s, \boldsymbol{s}.name, \boldsymbol{s}.args.mv, \boldsymbol{s}.args.v)$
$memop(\boldsymbol{m}) \;\Rightarrow\; \boldsymbol{m}.ms\langle 4\rangle.contents.mv := mem\_node(\boldsymbol{m}.name, \boldsymbol{g}.mv_t, \boldsymbol{m}.ms\langle 4\rangle.v)$
$end(\_) \;\Rightarrow\; io(\boldsymbol{g}.mv_t, \boldsymbol{g}.mv_s)$

I′ f returns one of the family of shadow computation update functions. Each function creates a new node from the inputs and inserts it in the DDFG.

syscall_node is similar, but builds nodes for system calls, including the indirect memory inputs. It inserts a pointer into the syscall DDFG node table, and sets the metavalues of the indirect memory outputs and the return value.

mem_node updates the metavalues of locations affecting by memory operations.

io prints the portions of the DDFG that are reachable from the system call nodes recorded in $\boldsymbol{g}.mv_s$.

6.8 Limits of Dynamic Binary Analysis

Different DBA tools record different amounts and kinds of metadata. One pertinent question: is there a maximum amount of (non-redundant) metadata that can be recorded? The answer is *yes*. Imagine the ultimate DBA tool, where "ultimate" is in terms of the information it gathers at run-time (rather than what it does with it). I will call it Omniscion, a suitably super name for a super tool.

Assume that Omniscion has the same access to run-time information that any real tool does. This implies that Omniscion has access to exactly the attributes introduced in Section 6.3, which have been used throughout the examples. Assume also that Omniscion can adjust its speed up or down, to match that of any tool; this nullifies any potential timing issues caused by the fact that most tools slow a program down.

So what information will Omniscion collect? In other words, what does Omniscion's description look like? It is very simple.

M $\boldsymbol{g}.mv$

I $start(\_) \;\Rightarrow\; f_1(!\boldsymbol{g}.mv, \boldsymbol{g}.t)$
$instr(\boldsymbol{i}) \;\Rightarrow\; f_2(!\boldsymbol{g}.mv, \boldsymbol{g}.t, \boldsymbol{i}.*)$
$end(\_) \;\Rightarrow\; f_3(!\boldsymbol{g}.mv, \boldsymbol{g}.t)$

The rules are straightforward; for each operation Omniscion adds analysis code that augments the global metadata with all the attributes involved in the operation ($\boldsymbol{i}.*$ represents every attribute of $\boldsymbol{i}$). Recording anything else would be redundant, as all metadata can be determined from these attributes. Omniscion only needs global metadata, since all other kinds can be replaced with global metadata, as Section 6.2 explained.

Omniscion's description is missing the **M′**- and **I′**-parts which would describe the form of $\boldsymbol{g}.mv$ and the functions f_1, f_2 and f_3. This is because Omniscion is a thought experiment and the important thing is what it *could* do with the information it collects. Ultimately, every DBA tool is limited by two things:

1. the amount of information it can record;
2. what it can do with that information.

With respect to item 2, all tools are equally limited by the laws of computation and what can be done on a machine. Therefore item 1 is the deciding factor. Since Omniscion takes as input the maximum possible amount of (non-redundant) information, it represents the limit of what DBA tools can achieve. In other words, *anything that cannot be determined by Omniscion cannot be determined by any DBA tool.*

The basic assumption underlying this is that tracking a program's start, end, every input to every instruction, plus $\boldsymbol{g}.t$ for all of these, provides Omniscion with all the information it needs. This is perfectly reasonable—after all, a program's execution is nothing more than the sum of every instruction executed by it and by the operating system kernel on its behalf, and all these instructions are instrumented by Omniscion.

This is a nice result to come out of such a simple line of reasoning. It was enabled by the separation between the formal and informal parts of the descriptions.

6.9 What is Dynamic Analysis?

Now is a good time to consider a fundamental question: what is dynamic analysis? "Dynamic analysis" is a term that is widely used, but does not have a clear definition. Consider the following descriptions of dynamic analysis from Ball [11] and Ernst [41].

> "Dynamic analysis is the analysis of the properties of a running program... [it] derives properties that hold for one or more executions by examination of the running program (usually through program instrumentation)."

> "Dynamic analysis operates by executing a program and observing the executions. Testing and profiling are standard dynamic analyses."

Without unduly picking on Ernst and Ball—they were not attempting to rigorously define dynamic analysis—these descriptions are typically vague. Contrast them with the following, much more precise definition of "static analysis", from Nielson, Nielson and Hankin [83].

> "[Static] Program analysis offers static compile-time techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically at run-time when executing a program on a computer."[2]

[2] Other, less precise definitions of static analysis are in use. Consider the definition from the Free On-Line Dictionary of Computing (FOLDOC) [56]: "A family of techniques of program analysis where the program is not actually executed (as opposed to dynamic analysis), but is analysed by tools to produce useful information. Static analysis techniques range from the most mundane (statistics on the density of comments, for instance) to the more complex, semantics-based techniques." I will use Nielson, Nielson and Hankin's definition—which excludes the suggested comment density statistics, for example, because they do not involve run-time values and behaviours—because it is precise, authoritative, and it can be interestingly contrasted with dynamic analysis.

(Note that this definition is aimed at static analyses used to enable program transformations, which must be safe. Some static analyses that detect bugs, such as those used by the Stanford checker [48], do not need to be safe. Therefore the words ”safe and” can be removed from the above definition to give a more general definition of static analysis.)

Because static analysis and dynamic analysis are duals [41], often ideas from one can be applied to the other with only a small twist. But the twist required here is not obvious—there seems to be no need for prediction and approximation of values and behaviours with dynamic analysis; at run-time the values and behaviours are simply *there*, ripe for the picking. So where is the duality?

In comparison to static analysis, which requires a modicum of theory to achieve interesting and reliable results, things are, in one sense, so easy at run-time that some dynamic tools barely do anything worth describing as “analysis”. Consider a tool like `strace`, mentioned in Section 6.6.2, which pretty-prints system call arguments and return values. Although `strace` is extremely useful, “dynamic observation” would be a more suitable description of what it does. But in comparison, the A and V bits of Memcheck, for example, are sufficiently complex to deserve the epithet “analysis”. Where does the analysis/not analysis dividing line lie between these two? The answer comes from metadata.

Dynamic analysis requires the maintenance of metadata. `strace` maintains no metadata, and lives forever in the present. In contrast, the Valgrind tools described in this dissertation, and many other run-time tools, maintain metadata and thus have a “memory” of a program’s execution (although the overlap checking part of Memcheck, described in Section 6.7.1, does not require metadata, and is thus an example of dynamic checking that does not involve true dynamic analysis). And crucially, maintained state is not truly metadata unless it is potentially reused again later by analysis code to compute more metadata, or in a conditional operation (e.g. Memcheck’s checking of A bits on a memory access). For example, if `strace` was implemented so that its output was collected into a buffer, and then printed out in one fell swoop at the program’s end, would that qualify as dynamic analysis? No; this state is not true metadata, as it is never used in either of the required ways.

So, the first part of the twist is that whereas static analysis involves prediction, dynamic analysis involves remembrance. The second part is the observation that dynamic analysis does, like static analysis, involve approximations (alternatively called abstractions), despite the perfect light of run-time. However, unlike static analysis where approximations are a necessary shortcoming, for dynamic analysis approximations are involved simply because there is no point in recording more information than necessary.

In fact, this is exactly what all metadata is—not just a remembrance of things past, but an approximate one. Consider the following examples.

- A global instruction count is an approximation of the trace of all executed instructions.
- Under Memcheck, a memory byte’s A bit is an approximation of all the memory operations that have affected it, and a value’s V bits are an approximation of all the constant values and operations on which it has a data dependency.
- A global cache state is an approximation of all the memory accesses that have occurred.

- Dynamic invariants are approximations of the values that have been seen at particular program points.
- A representation of the heap state is an approximation of all the heap allocations and deallocations that have occurred.

Even normal data values stored in registers and memory locations are approximations of the past operations on which they have a data dependency.

With these two ideas of remembrance and approximation in mind, the required twist is clear; mirroring the above definition of static analysis:

> Dynamic analysis offers run-time techniques for remembering approximations to the set of values or behaviours seen so far when executing a program on a computer.

Since metadata is exactly "approximations to the set of values or behaviours seen so far", the definition can be expressed more succinctly:

> Dynamic analysis is the gathering and use of metadata.

Or, to summarise in another way: *static analysis predicts approximations of a program's future; dynamic analysis remembers approximations of a program's past.*

6.10 Discussion

This chapter has given descriptions for a number of tools, including several Valgrind tools. From this a number of benefits and shortcomings are clear, as the following sections describe.

6.10.1 Benefits

The descriptions provide numerous benefits. First, they demonstrate some high-level things about DBA tools and dynamic analysis in general.

- They show that: all tools can be described in a unified way, in terms of their analysis code and metadata; that metadata is the key component, and is particularly important in making the description clear; what kinds of metadata are common; and how metadata is threaded through analysis code.
- Through the emphasis on metadata, they lead to a precise definition of dynamic analysis, based on the observation that static analysis predicts a program's future, whereas dynamic analysis remembers a program's past, and metadata is exactly what is remembered.
- They emphasise the importance of Valgrind's support for location metadata (shadow registers and shadow memory), which other DBI frameworks do not provide.

  They also show what other facilities a DBI framework such as Valgrind should have. For example, Valgrind currently allows function entry/exit to be instrumented, but function arguments cannot be easily obtained (except for those of system calls) without using function replacement, which is a shortcoming.

Following on from this, identifying how the framework features map onto concepts in the descriptions can also make easier to write tools, if this mapping is well documented.

- Omniscion demonstrates the theoretical limits of DBA tools, in terms of the information they can gather.

- They emphasise the importance of computing things statically where possible, which is a key implementation detail for making tools efficient.

- A similar description system may well be useful for characterising dynamic analysis done at the level of source code or byte-code. The basic concepts should be the same, although the details would differ—for example, program variables might serve as M-hooks instead of machine registers and memory locations, and program statements or expressions might serve as I-hooks rather than instructions.

Second, they help us write and understand tools.

- They provide a precise way to describe tools. The descriptions describe the tools in great detail, giving an excellent understanding of how they work. By forcing one to specify many things, they are much more precise and directed than plain language descriptions would be. One would be hard-pressed to find a properly written description of a tool that leaves the reader unsure what it does.

- They provide a concise way to describe tools. Compare a few thousand lines of code with a few lines of description. Plain language descriptions would be much longer.

- They provide a foundation for comparing tools. In particular, it shows how some tools subsume others (e.g. Memcheck subsumes Addrcheck), or are composed of multiple parts.

- Perhaps most importantly, writing a tool description forces one to think extremely carefully about the tool. In my experience, writing accurate descriptions is surprisingly difficult and time-consuming. However, thinking about a tool so carefully is extremely useful, particularly when the thinking is directed in a way that draws out the important characteristics of the tool. If done before a tool is written, it can make it much clearer how the tool should be implemented. If done for an existing tool, it can identify how the tool could be improved.

  For example, I once tried implementing an invariant tracker and checker similar to DIDUCE [49] as a Valgrind tool. After some experimentation, I abandoned the effort due to unsatisfactory results. Several months later, when first thinking about the description system, I realised that I had mis-implemented the tool, associating invariant data with *memory locations* rather than *code* (i.e. instructions or source code locations). This confusion probably arose because the existing tools such as Memcheck used location metadata, and I unwittingly based the invariant detector on it. Such a confusion would not have occurred if I had written a description before beginning coding, or had even been thinking in the way that the descriptions encourage.

As an even better example, writing the description for Cachegrind led to it being greatly improved. Before I wrote its correct description, Cachegrind had been in existence for two years, and widely used, including for the Glasgow Haskell experimentation described in Section 3.4. I had even drafted Chapter 3 of this dissertation describing Cachegrind. Upon realising the first description I had written was wrong, I realised that Cachegrind had a single data structure that awkwardly served two distinct purposes: first, to store cost centres; second, to cache the static instruction attributes to save passing them as arguments to the simulation functions. After realising this, I was able to split this single, complicated data structure into two much simpler data structures—the cost centre table and the instr-info table, both of which Section 3.3.1 described. A particular improvement was the association of cost centres to source line codes, rather than individual instructions, as had been the case.

These changes resulted in the following improvements to Cachegrind: source code was reduced in size by 29%, and made substantially simpler in many places; memory consumption was reduced by 10–20%, due to fewer cost centres being required; the size of the output files was reduced by over 90%, and so annotation speed was hugely improved; and the run-time speed was unchanged. Also, the functionality was improved—with the old approach, the hit/miss counts for instructions that were unloaded at run-time had to be merged into a single "discard" cost centre; with the new approach they are preserved accurately. These impressive improvements were a direct result of using the description system.

6.10.2 Shortcomings

Despite their benefits, the descriptions have the following shortcomings.

- They can be messy, particularly for the more complicated DBA tools. However, this reflects the tools themselves, and is hard to avoid.

- They do not provide any way of formally reasoning about tools to the extent of e.g. proving or verifying any aspects of them. Again, this is difficult in general because many interesting tools are complicated in practice, and do things that are very hard to reason about formally.

- There is some subjectivity in the formal parts of the descriptions. For example, the choice of M-hooks and attributes was chosen according to what seemed to fit well. Also, there is no single correct description for any tool, due to the informal parts.

- They may be incomplete, in that there may be some tools that cannot be adequately described with it. The form of the descriptions was decided upon by looking at many example tools, and coming up with a system that fit everything that they needed. This took many revisions and reworkings. If the system is inadequate, it is possible that minor changes would render it adequate. However, even if not, that does not detract from its usefulness for the tools it can describe.

- The system cannot describe some things very well, such as function replacement.
- The level of abstraction in the descriptions means that they cannot exactly portray all the low-level details of a tool. This is hard to avoid given their high level. However, the informal description parts can describe low-level details.
- Writing them is entirely manual. There is no way to automatically check a description to determine if it is accurate.

None of these shortcomings really detract from the benefits, however. The descriptions are a means, rather than an end in themselves, so even though they have flaws they are still useful.

6.10.3 Ideality vs. Reality

In many fields of computer science, there are two basic approaches to a problem, which differ in the chosen starting point. As an example, consider the goal of ensuring programs are correct.

The first approach is to begin with a perfect ideal—for example, a program with no bugs—and to work towards the real world. That is, to find ways to make that ideal practical, and usable in real programs. For example, one can invent a new programming language with a type system or other facilities that make certain kinds of errors impossible. Such approaches usually involve rigorous theoretical backing. When it works, this approach is best, as it usually leads to desirable properties such as soundness. However, the history of computer science is littered with research that begins in ideality, but does not progress far enough towards the real world to result in any tangible benefit. This approach is also often not useful for improving existing programs.

The second approach is to begin with the imperfect real world—for example, real programs already written in languages like C, C++ and Fortran—and find ways to make those programs less imperfect. DBA tools such as Memcheck and Annelid are prime examples of this approach. This more pragmatic approach rarely reaches the ideal, but an improvement on an existing situation is better than no improvement at all.

Some of the systems described in the related work of Section 6.11.2, e.g. Kishon, Hudak and Consel's monitoring semantics, take the former approach. This chapter's description system is very much an example of the latter approach—it takes existing, real-world DBA tools, warts and all, and describes them as best it can. In fact, the entire field of dynamic binary analysis and instrumentation, and all the work presented in this thesis, also take this latter approach. The results may not always be pretty, and they may have flaws, but they are still worthwhile.

6.10.4 Executable Descriptions?

A final note: one possible use for this sort of description system would be to automatically generate some of the tool code for a framework such as Valgrind, similar to the way parser generators like YACC generate parsers from a high-level grammar description. The formal parts of the tool descriptions would be written in a special language that looks a lot like how those parts have been written in this chapter. The informal parts would be written not in plain language, but instead in a programming language such as C; they would correspond to YACC's "user actions". The user actions would be able to refer to attributes like ***i***.*opcode* in some convenient way, just like YACC uses `$$`, `$1`, etc., to refer to elements in a parse tree.

However, the amount of code saved would probably not be substantial, as much of the complexity in tools is typically in the parts that correspond to the informal descriptions of metadata and analysis code, i.e. those parts which would have to be programmed as user actions anyway. Also, tools need to be programmed carefully to minimise their overhead, and it would be difficult for automatic code generation to satisfy this requirement. Valgrind's built-in support for things like shadow registers and shadow memory is a good compromise between making tools easy to write, and giving tools enough control.

6.11 Related Work

There is a great deal of theory behind static analysis. There are whole conferences and books (e.g. [83]) about it. In comparison, there is far less theory behind dynamic analysis. This is partly because dynamic analysis tends to be easier from a theory point of view; working with the actual run-time values and behaviours makes many things simple. Also, because dynamic analysis is not sound in general, one does not have to worry about using theory to ensure this property.

There are two main areas of work related to this chapter's descriptions. The first is in the general description and classification of dynamic analysis and dynamic analysis tools. The second is in actual systems, theoretical and implemented, that involve high-level specifications of dynamic analysis tools.

6.11.1 Classifications

Ernst [41] nicely describes the complementary characteristics of static analysis and dynamic analysis, as covered in Section 1.1.1. Ernst also identifies that both kinds of analysis have in common the characteristic that they both consider only a subset of executions: "those that induce executions of a certain variety" for static analysis, and those observed during execution for dynamic analysis. However, Ernst does not consider the novel future/past duality described in Section 6.9, nor the role of metadata in dynamic analysis.

Ball, in an unpublished summer school presentation [12], hints at the importance of metadata and approximation in dynamic analysis, with the following two bullet points: "Dynamic analysis: abstraction used in parallel with, not in place of, concrete values", and "Dynamic analysis is a problem of data aggregation and compression, as well as abstraction". However, he does not make the connection nearly as strongly as Section 6.9 does.

Zeller [121] gives a nice, brief four-level categorisation of tools based on how many runs of a program they observe. The tools that this chapter has considered fit mostly in the "observation" (1 run observed) category, although they could be in the "Induction" (n runs) category. They are not in "Static" (0 runs) or "Experimentation" (n runs, where later runs are guided by results from earlier runs) categories. Zeller's simple categorisation is useful at a high-level—in a similar way that it is useful to know which M-hooks metadata is attached to—but since it only describes a single

```
syscall::mmap:entry                          // I-hook specification
/execname == "Xsun"/                         // run-time predicate
{
    @[pid, arg4] = count();                  // global metadata update
}

END                                          // I-hook specification
{
    printf("%9s %13s %16s\n", "PID", "FD", "COUNT");
    printa("%9d %13d %16@d\n", @);           // I/O actions
}
```

Figure 6.1: Example DTrace probes

characteristic it is does not give (nor is it intended to give) very much information about a tool.

Reiss [94] describes the features that an ideal dynamic instrumentation framework would provide, such as low usage overhead, low execution overhead, and adequate handling of real programs including multi-threaded programs and libraries. This provides a useful view of dynamic analysis and instrumentation in general, but does not describe individual tools at all.

Delgado [36] gives an ad-hoc taxonomy that categorises run-time monitoring tools that detect, diagnose, and recover from software faults. The taxonomy has three main parts, for the "Specification Language", "Monitor" and "Event-handler" parts of a tool. The classification of each part is done with a multi-level tree-shaped classification scheme. Classifying a tool requires some human judgement. Delgado's scheme covers more types of tools than this chapter's descriptions do; for example, it includes tools that instrument source code. However, it only gives a very high-level overview with little detail about how each tool actually works.

6.11.2 Specification Systems

Sloane's little-known but far-sighted system Wyong [103] generates tools for ATOM [105] from high-level, concise, declarative specifications based on attribute grammars—i.e. exactly the YACC-style tool generation mentioned in Section 6.10.4. Wyong has an impressive set of features, particularly for a tool from 1997. It has a program grammar that describes the static and dynamic structure of a program in terms of function calls, basic blocks, and instructions. The requested raw inputs are gathered automatically. Metadata can be specified at different levels, e.g. global and per-instruction, and metadata storage is handled automatically. Analysis steps are automatically sequenced in the correct order according to data dependencies. The specifications are quite architecture-independent and suitable for retargetting to different systems. They are also highly composable. On the downside, it is unclear but seems unlikely that Wyong would be suitable for building heavyweight tools such as Memcheck; the example in the cited paper is for a lightweight branch counting tool that tracks taken/not taken counts for each instruction. It is also unclear how good the performance of tools built with it is. Finally, Wyong was implemented to use ATOM on Alphas and so is now unfortunately defunct.

DTrace [23] is a powerful instrumentation facility built into the Solaris 10 kernel that is designed for profiling systems. Instrumentation *probes* are specified in a high-level pattern/action language similar to C and awk. Each probe's definition looks quite similar to the rules in this chapter. For example, the pair of simple probes in Figure 6.1 (from the cited paper, with my annotations) measure the number of `mmap()` calls performed by all the X server processes on a machine, and what file descriptors are mapped. The first probe specifies the I-hook of interest—entry to the `mmap()` system call—and uses a run-time predicate to decide whether to run the analysis code (by checking if the process name is `Xsun`, the name of the X server). The instrumentation increments an element (via a built-in function `count()`) in an associative array (a built-in data type for storing global metadata) indexed by the process ID and the mapped file descriptor (`arg4`). The second probe specifies an I/O action performed when the monitoring is ended, which dumps the collected metadata. The exact details of this example are not so important; more notable are the concepts shared with this chapter's descriptions.

So with its concise specification of I-hooks, support for different kinds of metadata (it also supports thread-local metadata), and use of predicates, update functions, and I/O actions, DTrace probes look a lot like the descriptions presented in this chapter. The main difference, apart from DTrace being an actual implemented system, is that DTrace analysis code is less fined-grained than that described here, typically at the level of function or system call entries, not at the level of individual instructions, and it does not support location metadata. Nonetheless, it is interesting and encouraging to see an independently created system overlap this chapter's descriptions, as it provides further evidence that they are a natural and effective way to describe DBA tools.

Kishon, Hudak and Consel [63] define a *monitoring semantics*, a non-standard model of program execution that captures monitoring activity performed by many dynamic analysis tools. The monitoring semantics is a conservative extension of a programming language's normal denotational semantics, and can be derived automatically for any language which has a continuation semantics specification. Individual program monitor specifications have three parts—the monitor syntax, the monitor algebra (which includes the monitor state, i.e. the metadata), and the monitoring functions—and are analogous to this chapter's descriptions. The cited paper gives four example monitor specifications: a tracer, a profiler, a demon and a collecting monitor; it also discusses an interpreter-based implementation. The examples are based around a small language based on the lambda calculus. Their system is quite neat, and compared to this chapter's descriptions, it has a much stronger theoretical basis. However, their system is language-specific and highly idealised, so it cannot be applied to real-life DBA tools.

Polymer [14] is a language for formally specifying program monitors for enforcing security policies. A policy can force a client program to abort execution, prevent certain operations (e.g. function calls) from occurring, or do an alternative computation for the client. Policies can be composed. Polymer has a formal semantics, and a type and effect system which ensures non-interference between policies that are composed together. The policies look somewhat like this chapter's descriptions, although the examples given are all expressed at the level of function calls in source code, rather than at a binary level, and they have little in the way of metadata, as they are basically designed to check certain operations, and either allow or prevent them.

6.12 Conclusion

This chapter has presented a system of detailed descriptions of DBA tools. The descriptions have four parts: formal and informal descriptions of the metadata, and formal and informal descriptions of the analysis code. The separation into formal and informal parts is crucial for making the descriptions useful. Multiple examples were given, including descriptions of all the Valgrind tools described in this dissertation.

The descriptions, despite some shortcomings, provide multiple benefits. First, they show that metadata and analysis code are the key concepts underlying DBA. They demonstrate simply and intuitively—through Omniscion—the limits of DBA. They make it clear what features a DBI framework like Valgrind should provide. They are precise and concise—Valgrind tools containing several thousand lines of C code were meaningfully described in as little as ten or twenty lines. They also lead to a precise definition of dynamic analysis, and a better understanding of dynamic analysis in general, based around the idea that metadata is the approximation of a program's past. Finally, the descriptions provide a directed way of thinking about tools which improves one's understanding how a tool works and what it does. This can (and already has, in the case of Cachegrind) lead to improved tool implementations.

CHAPTER 7

Conclusion

This dissertation has advanced the theory and practice of dynamic binary analysis and instrumentation, with an emphasis on the importance of the use and support of metadata.

7.1 What Just Happened

Chapter 2 described Valgrind, a DBI framework. The chapter emphasised Valgrind's support for heavyweight DBA via a description of Memcheck, a checking DBA tool. In particular, the discussion of heavyweight DBA introduced the concepts of location metadata and shadow computation.

Chapter 3 described Cachegrind, a useful profiling tool. Chapter 4 described Annelid, a novel checking tool. Chapter 5 described Redux, a novel visualisation tool. All these tools were built with Valgrind, and all perform heavyweight DBA, the latter two using location metadata and shadow computation.

Chapter 6 described a system of semi-formal descriptions of DBA tools. The descriptions are useful for improving the understanding of individual tools, which can lead to improved implementations. They are also useful for understanding DBA tools in general. In particular, they emphasise that analysis code and metadata are the primary concepts underlying all DBA tools. The discussion of the imaginary tool Omniscion showed the limits of DBA. The chapter also identified a key feature of dynamic analysis in general, that it considers approximations of a program's past, as opposed to static analysis which considers approximations of a program's future; and that metadata is exactly what these approximations are.

If a single idea is to be taken away from this dissertation, it is that metadata—the approximations of a program's past values and behaviours—is crucial. Metadata is at the very heart of dynamic binary analysis, rich metadata leads to powerful DBA tools, and efficient heavyweight DBA tools can be well supported by DBI frameworks.

7.2 Future Work

Valgrind is a system with great potential for further research and development. There are three main areas for improvement: support for new architectures, support for multiple operating systems, and new kinds of tools. Work on these areas will help bring Valgrind to a wider audience, and also teach us more about dynamic binary analysis and instrumentation.

7.2.1 New Architectures

Valgrind currently only works on x86 machines, which is a big limitation. Many users want it for other architectures, most notably x86-64; PowerPC and SPARC are also requested fairly frequently.

Porting Valgrind to new architectures in a clean way will not be an easy task. Some kind of architecture abstraction layer will be required. This layer will need to factor out at least the three following architecture differences.

- All the pieces of code within Valgrind's core that deal directly with machine registers will have to be factored out, as they will be different for each architecture. This should be fairly straightforward.
- Valgrind's code currently assumes 32-bit words throughout. Making the code 64-bit clean is not particularly difficult, but will require quite some effort. An x86-64 port will be perfect for this.
- By far the hardest part—requiring genuine research rather than just coding effort—is choosing an intermediate representation (IR). UCode was designed for x86, and has a number of x86-specific features. It also has very poor support for instrumenting floating point and SIMD registers.

  A more general-purpose IR is necessary that will be able to express the features of multiple architectures. A key requirement of the IR is that it should be possible to write tools in an architecture-neutral fashion. Complete architecture-neutrality will be hard to achieve; for example, Memcheck currently exploits some of the behaviours of the x86 condition codes to efficiently do its V bit shadow computation. However, as much of each tool plug-in as possible should be architecture-neutral.

  To prevent the IR effectively becoming the union of all the supported instruction sets, some kind of line will have to be drawn; the more unusual instructions will probably be represented with some kind of "opaque" instruction in the IR. The code generator will just copy the original instruction bytes through, but the details will not be expressed directly as opcodes in the IR. These opaque instructions could have a string attached which gives the original instruction form, so that a tool that really did need to know about unusual instructions could find out about them if absolutely necessary.

  One consequence of this "opaque" approach would be that dynamic binary translation—hosting binaries for one architecture on another—would not be possible. Dynamic binary translation is extremely difficult in general, and would be even harder to do in a way that supports heavyweight DBA within Valgrind, and so is not a current design goal for Valgrind.

Paul Mackerras has already completed an experimental port of Valgrind to the PowerPC/Linux platform. He augmented UCode with eight new opcodes to handle PowerPC instructions that could not be expressed in the current UCode. The challenge now is to come up with an improved IR that has the right abstractions for expressing these extra opcodes in a cleaner way.

7.2.2 New Operating Systems

Valgrind currently only works on Linux, which is another big limitation. Many users want it for other Unix-style operating systems, most notably the BSDs and MacOS X.

Just like architecture ports, operating system (OS) ports will be best achieved by introducing an OS abstraction layer that factors out the OS-specific parts, such as those involving threads, signals and system calls. Unlike the architecture abstraction layer, which will affect both the core and tools significantly, the OS abstraction layer will mostly affect the core, and tool plug-ins should not be affected much. The kinds of details that will need to be factored out include interactions with system calls, signals and threads. This work will be fiddly—currently, these parts are the generally the most intrusive and most likely to be broken by changes—but should be possible. Doug Rabson has already completed an experimental port of Valgrind to x86/FreeBSD.

A Windows port would also be extremely popular. However, because Windows has a completely different architecture to the aforementioned Unix-style operating systems, porting Valgrind to Windows would require extremely large changes. It is hard to see this happening any time soon.

7.2.3 New Tools

Many new tools could be built using Valgrind. For example, memory use is often a huge factor in the speed of a program. Imagine a tool that could identify which parts of a program that are responsible for generating large amounts of memory traffic. The metadata tracked here might be the number of bytes of memory copies performed by each line of source code. Or, it might be useful to record for each memory location which line of source code last touched it. Or a combination of such information might be useful.

Whether this tool would be useful remains to be seen. However, one thing is certain—there is a strong trend towards more powerful tools using richer forms of metadata, due to the increasing complexity of both software and hardware; the shadow computation performed by Memcheck, Annelid and Redux are excellent examples. Valgrind's support for heavyweight dynamic analysis makes it an ideal platform for developing the next generation of DBA tools.

Another possibility, as Section 2.4.5 briefly mentioned, is in the area of combined static and dynamic analysis. The idea of combining static and dynamic analysis is one that has been talked about frequently by researchers, but not a great deal of progress has been made. Valgrind could, in conjunction with a compiler that inserts client requests into compiled programs, provide a good platform for real experimentation and progress on this front.

7.2.4 Avoiding Code Blow-up

If the architecture and OS abstraction layers are completed, Valgrind will have three dimensions of genericity, supporting M architectures, N operating systems, and P tools. For this to be remain viable, it is imperative that an $M \times N \times P$ code blow-up is avoided. An $M + N + P$ code increase would be ideal. This will not be possible in practice, but it will be important to do the abstraction layers well enough to keep code expansion as close as possible to this ideal, otherwise code maintenance will overwhelm efforts to further extend Valgrind.

7.3 Final Words

Ever since programming began, programming tools have slowly improved, making the difficult job of writing good programs easier. Dynamic binary analysis tools such as Memcheck, Cachegrind, Annelid and Redux, built with dynamic binary instrumentation frameworks such as Valgrind, are just another step in this progression. I hope that they will help many programmers improve their programs. But I look forward more to the day when the art of programming has progressed such that they are no longer necessary, having been eclipsed by something even better.

APPENDIX A

Glossary

This glossary provides a quick reference for important terms used in this dissertation. Terms appearing in **bold** within definitions have their own entries. Terms introduced or defined by this dissertation (or by the author publications mentioned at the start of this dissertation) are marked with a † symbol. Valgrind-specific terms used by this dissertation are marked with a ♮ symbol. Among the unmarked terms, note that some are used by others in multiple ways, but are used in only one way in this dissertation.

Active Used to describe **analysis code** that affects the semantics of the **instrumented** program. Contrast with **passive**.

†**Analysis code** Any code added to a program for the purpose of doing **dynamic analysis**.

†**Attribute** A characteristic of an **M-hook** or **I-hook**, such as a memory location's address.

♮**Basic block** A straight-line sequence of machine code, whose head is jumped to, and which ends in a control flow transfer such as a jump, call, or return. Note that this definition is different to some uses of the term; in particular, a jump can land in the middle of a previously seen basic block.

†**Binary analysis** The analysis of programs at the level of machine code. Contrast with **source analysis**.

♮**Client** A program running under the control of a **Valgrind tool**.

♮**Client request** A small piece of code, embedded in a **client**, that is a no-op when the client is run normally, but is interpreted specially as a message or query by Valgrind's **core** and/or **tool plug-ins**. Client requests can take multiple arguments, and even return values to the client.

♮**Core** The main part of Valgrind and the major component in every **Valgrind tool**, doing most of the work except for **instrumentation**, which is done by a **tool plug-in**.

†**DBA** Short for **dynamic binary analysis**.

DBI Short for **dynamic binary instrumentation**.

†**DDFG** Short for **dynamic data flow graph**.

†**Dynamic analysis** Run-time techniques for remembering approximations to the set of values or behaviours seen so far when executing a program. Or, less formally, the analysis of programs at run-time. Involves the use of **analysis code** and **metadata**. Contrast with **static analysis**.

†**Dynamic binary analysis** The intersection of **dynamic analysis** and **binary analysis**.

Dynamic binary compilation and caching An implementation technique used in **dynamic binary instrumentation**, **dynamic binary optimisation**, and **dynamic binary translation**, whereby a program's original code is **instrumented** and stored in a cache, and only code in the code cache is run. Used by Valgrind.

Dynamic binary instrumentation The **instrumentation** of a program at the level of machine code, at run-time, with **analysis code**, for the purpose of doing **dynamic binary analysis**. Used by Valgrind. Contrast with **static binary instrumentation**.

Dynamic binary optimisation The use of **dynamic binary analysis** and **dynamic binary instrumentation** specifically to improve a program's speed by utilising optimisations that can not be performed statically by a compiler.

Dynamic binary translation The running of a program compiled for one platform (the guest) on another (the host), typically through a translation tool.

†**Dynamic data flow graph** A directed acyclic graph that represents all value-producing operations executed by a program at run-time, as produced by the **Valgrind tool** Redux.

†**Dynamic source analysis** The intersection of **dynamic analysis** and **source analysis**.

†**Essence** The parts of a **dynamic data flow graph** that are reachable from the inputs of any system calls executed by a program, and thus have a data flow connection to the program's observable behaviour.

†**Global metadata** **Metadata** that pertains to an entire program or the whole machine.

†**Heavyweight** Used to describe **dynamic binary analysis** tools that involve **rich metadata** (particularly **location metadata**) and **analysis code** that is highly pervasive and interconnected. Particularly used for **shadow computation**. Contrast with **lightweight**.

†**I-hook** A piece of code, such as an instruction, which a **dynamic analysis** tool can **instrument** with **analysis code**.

Instrumentation The act of adding **analysis code** to a program. Note that in this dissertation, this term is not used for the added code itself.

Intermediate representation A representation of code used internally by a program such as a compiler. Valgrind uses an intermediate representation called **UCode**.

IR Short for **intermediate representation**.

†**Lightweight** Used to describe **dynamic binary analysis** tools that do not involve rich metadata or complex analysis code. Contrast with **heavyweight**.

†**Location metadata** **Metadata** describing registers or memory locations.

†**Metadata** Approximations to the set of values or behaviours seen so far when executing a program. Metadata is attached to **M-hook**s. The key component of **dynamic analysis**.

†**Metavalue** A single piece of **metadata**.

†**M-hook** A program or machine entity, such as a register or value, to which **metadata** can be attached.

Passive Used to describe **analysis code** that does not affect the semantics of the **instrumented** program. This includes any **dynamic binary optimisation** performed by a tool. Contrast with **active**.

Red-zone A small area at the edge of a region of memory, such as a heap block, that should not be accessed. Any accesses to the red-zone indicate a bug in a program.

†**Rich metadata** Particularly complex forms of **metadata**, such as **location metadata**.

RISC Short for reduced instruction set computing. Used to describe an instruction set that is particularly simple, and in which memory accesses are typically only done through dedicated load and store instructions.

†**Shadow computation** A particular kind of **dynamic binary analysis** in which every value in registers and memory is shadowed with a **metavalue** (also known as a **shadow value**) that describes the value, and every value-writing operation is shadowed with a shadow operation that computes and writes the corresponding **shadow value**.

♮**Shadow memory** Valgrind's support for the use of a **shadow value** for every value in memory.

♮**Shadow registers** Valgrind's support for the use of a **shadow value** for every value in a register.

†**Shadow value** Synonym for a **metavalue**, typically used in the context of **shadow computation**.

†**Source analysis** The analysis of programs at the level of source code. Contrast with **binary analysis**.

Static analysis Compile-time techniques for predicting computable approximations to the set of values or behaviours arising at run-time when executing a program.[1] Or, less formally, the analysis of programs prior to run-time. Contrast with **dynamic analysis**.

†**Static binary analysis** The intersection of **static analysis** and **binary analysis**.

Static binary instrumentation The **instrumentation** of a program at the level of machine code, prior to run-time, with **analysis code**, for the purpose of doing **dynamic binary analysis**. Contrast with **dynamic binary instrumentation**.

†**Static source analysis** The intersection of **static analysis** and **source analysis**.

♮**Tool plug-in** A component of a **Valgrind tool** which defines an **instrumentation** pass over **UCode**, and plugs into Valgrind's **core**.

♮**Translation** What a **Valgrind tool** produces when it has finished transforming and **instrumenting** a **basic block** of code.

♮**UCode** The **intermediate representation** used by Valgrind. **Tool plug-ins** perform **instrumentation** of UCode. Valgrind's **core** performs several passes over UCode, including optimisation, register allocation and code generation.

♮**Valgrind tool** A tool built by combining a **tool plug-in** with Valgrind's **core**.

†**Value metadata** **Metadata** describing values. When using a system like Valgrind that makes explicit every computed intermediate value, it is no different to **location metadata** from an implementation point of view, but conceptually it is substantially different.

[1] This is a paraphrasing of Nielson, Nielson and Hankin's definition [83].

Bibliography

[1] AbsInt Angewandte Informatik. aiSee – graph visualisation. `http://www.absint.com/aisee/`.

[2] Advanced Micro Devices. 3DNow! technology manual, 2000. `http://www.amd.com`.

[3] Advanced Micro Devices. AMD Athlon processor x86 code optimization guide, July 2001. `http://www.amd.com`.

[4] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the ACM SIGSOFT'91 Symposium on Software Testing, Analysis, and Verification (TAV4)*, pages 60–73, Victoria, Canada, October 1991.

[5] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software—Practice and Experience*, 23(6):589–616, June 1993.

[6] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '90)*, pages 246–256, White Plains, New York, USA, June 1990.

[7] AT&T Labs-Research. Graphviz. `http://www.research.att.com/sw/tools/graphviz/`.

[8] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '94)*, pages 290–301, Orlando, Florida, USA, June 1994.

[9] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2000)*, pages 1–12, Vancouver, Canada, June 2000.

[10] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the 13th International Conference on Compiler Construction (CC 2004)*, pages 5–23, Barcelona, Spain, March 2004.

[11] Thomas Ball. The concept of dynamic analysis. In *Proceedings of the 7th European Software Engineering Conference and the 7th ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE'99)*, pages 216–234, Toulouse, France, September 1999.

[12] Thomas Ball. The essence of dynamic analysis. Talk presented at The University of Washington/Microsoft Research Summer Institute on Technologies to Improve Software Development, 1999. `http://research.microsoft.com/tisd/Slides/TomBall.ppt`.

[13] Elena Gabriel Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanović, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*, pages 281–289, Washington, DC, USA, October 2003.

[14] Lujo Bauer, Jarred Ligatti, and David Walker. Types and effects for non-interfering program monitors. In *Proceedings of the International Symposium on Software Security (ISSS 2002)*, pages 154–171, Tokyo, Japan, November 2002.

[15] Fabrice Bellard. QEMU CPU emulator. `http://fabrice.bellard.free.fr/qemu/`.

[16] Frederick P. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975.

[17] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. In *Proceedings of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, Austin, Texas, USA, December 2001.

[18] Derek Bruening et al. DynamoRIO. `http://www.cag.lcs.mit.edu/dynamorio/`.

[19] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*, pages 265–276, San Francisco, California, USA, March 2003.

[20] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[21] Michael Burrows, Stephen N. Freund, and Janet L. Wiener. Run-time type checking for binary programs. In *Proceedings of the 12th International Conference on Compiler Construction (CC 2003)*, pages 90–105, Warsaw, Poland, April 2003.

[22] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience*, 30(7):775–802, 2000.

[23] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 15–28, Boston, Massachusetts, USA, June 2004.

[24] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gillies. Mojo: A dynamic optimization system. In *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, Monterey, California, USA, December 2000.

[25] Olaf Chitil, Colin Runciman, and Malcolm Wallace. Transforming Haskell for tracing. In *Proceedings of the 2002 International Workshop on the Implementation of Functional Languages*, pages 165–181, Madrid, Spain, September 2002.

[26] Jong-Deok Choi, Barton P. Miller, and Robert H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, October 1991.

[27] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Faculty of Information Technology, Queensland University of Technology, Australia, July 1994.

[28] Cristina Cifuentes, Brian Lewis, et al. Walkabout. `http://research.sun.com/walkabout/`.

[29] Cristina Cifuentes, Brian T. Lewis, and David Ung. Walkabout – A retargetable dynamic binary translation framework. Technical Report TR-2002-106, Sun Microsystems Laboratories, Palo Alto, California, USA, January 2002.

[30] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 128–137, Nashville, Tennessee, USA, May 1994.

[31] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report UWCSE 93-06-06, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, USA, 1993.

[32] Robert Cohn. Instrumentation of Intel Itanium Linux programs with Pin. Tutorial at International Symposium on Code Generation and Optimization (CGO'04), San Jose, California, USA, March 2004.

[33] Robert S. Cohn, Daniel A. Connors, and Dirk Grunwald. Pin. `http://systems.cs.colorado.edu/Pin/`.

[34] Compuware Corporation. Boundschecker. `http://www.compuware.com/products/devpartner/bounds.htm`.

[35] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252, Los Angeles, California, USA, January 1977.

[36] Nelly M. Delgado. A taxonomy of dynamic software-fault monitoring tools. Master's thesis, Computer Science Department, University of Texas at El Paso, El Paso, Texas, USA, 2001.

[37] Luiz DeRose, Ted Hoover Jr., and Jeffrey K. Hollingsworth. The Dynamic Probe Class Library – an infrastructure for developing instrumentation for performance tools. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, San Francisco, California, USA, April 2001.

[38] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, and Joseph A. Fisher. Deli: A new run-time control point. In *Proceedings of the 35th Annual Symposium on Microarchitecture (MICRO35)*, pages 257–270, Istanbul, Turkey, November 2002.

[39] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, and Josh Fisher. A new facility for dynamic control of program execution: DELI. Talk presented at the Second International Workshop on Embedded Software (EMSOFT'02), October 2002.

[40] DynInst. `http://www.dyninst.org/`.

[41] Michael D. Ernst. Static and dynamic analysis: synergy and duality. In *Proceedings of the ICSE Workshop on Dynamic Analysis (WODA 2003)*, pages 6–9, Portland, Oregon, May 2003.

[42] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[43] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2):163–189, 1999.

[44] Manel Fernández and Roger Espasa. Dixie: A retargetable binary instrumentation tool. In *Proceedings of the Workshop on Binary Translation*, Newport Beach, California, USA, October 1999.

[45] Free Software Foundation. GNU General Public License. `http://www.gnu.org/licenses/gpl.txt`.

[46] Andy Gill. Debugging Haskell by observing intermediate data structures. In *Proceedings of the 2000 Haskell Workshop*, Montreal, Canada, September 2000.

[47] The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`.

[48] Seth Hallem, Benjamin Chen, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2002)*, Berlin, Germany, June 2002.

[49] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering (ICSE 2002)*, pages 291–301, Orlando, Florida, USA, May 2002.

[50] Timothy L. Harris. Dependable software needs pervasive debugging. In *Proceedings of the ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[51] Pieter H. Hartel et al. Benchmarking implementations of functional languages with "Pseudo-knot" a float-intensive benchmark. *Journal of Functional Programming*, 6(4):621–655, 1996.

[52] Reed Hastings and Bob 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.

[53] Don Heller. Rabbit: A performance counters library for Intel/AMD processors and Linux. `http://www.scl.ameslab.gov/Projects/Rabbit/`.

[54] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, pages 841–850, Knoxville, Tennessee, USA, May 1994.

[55] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[56] Imperial College Department of Computing. Free on-line dictionary of computing (FOLDOC). `http://wombat.doc.ic.ac.uk/foldoc/`.

[57] Intel. IA-32 Intel architecture software developer's manual. `http://www.intel.com/`.

[58] Intel. Intel Pentium 4 and Intel Xeon processor optimization: Reference manual. `http://www.intel.com/`.

[59] Christopher January. Logrind 2: A program trace framework. MEng Computing final year individual project report, Department of Computing, Imperial College, London, United Kingdom, 2004.

[60] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the Third International Workshop on Automated Debugging (AADEBUG'97)*, pages 13–26, Linköping, Sweden, May 1997.

[61] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *LNCS*, pages 220–242, Jyväskylä, June 1997. Springer-Verlag.

[62] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, San Francisco, California, USA, August 2002.

[63] Amir Kishon, Paul Hudak, and Charles Consel. Monitoring semantics: A formal framework for specifying, implementing, and reasoning about execution monitors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '91)*, pages 338–352, Toronto, Canada, June 1991.

[64] Alexander Klaiber. The technology behind Crusoe processors. Transmeta Corporation White Paper, January 2000. `http://www.transmeta.com`.

[65] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '95)*, pages 291–300, La Jolla, California, USA, June 1995.

[66] Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Debugging via run-time type checking. In *Proceedings of Fundamental Approaches to Software Engineering (FASE 2001)*, Genoa, Italy, April 2001.

[67] Thomas Ludwig and Roland Wismüller. OMIS 2.0 — a universal interface for monitoring systems. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: Proceedings of the 4th European PVM/MPI Users' Group Meeting*, volume 1332 of *LNCS*, pages 267–276, Kraków, Poland, November 1997. Springer-Verlag.

[68] 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 International Symposium on Code Generation and Optimization (CGO 2004)*, pages 15–26, Palo Alto, California, USA, March 2004.

[69] Jonas Maebe and Koen De Bosschere. Instrumenting self-modifying code. In *Proceedings of the Fifth International Workshop on Automated and Algorithmic Debugging (AADEBUG2003)*, Ghent, Belgium, September 2003.

[70] Jonas Maebe, Michiel Ronsse, and Koen De Bosschere. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Compendium of Workshops and Tutorials held in conjunction with PACT'02: International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, Virginia, USA, September 2002.

[71] Jonas Maebe, Michiel Ronsse, and Koen De Bosschere. Precise detection of memory leaks. In *Proceedings of the Second International Workshop on Dynamic Analysis (WODA 2004)*, Edinburgh, Scotland, May 2004.

[72] Stefan Manegold and Peter Boncz. Cache-memory and TLB calibration tool. `http://www.cwi.nl/~manegold/Calibrator/`.

[73] Stephen McCamant. Kvasir: a Valgrind tool for program tracing. Message to the valgrind-developers mailing list, June 2004.

[74] Microsoft Research. Binary technologies group. `http://research.microsoft.com/bit/`.

[75] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 28(11):37–46, November 1995.

[76] Alan Mycroft. Type-based decompilation. In *Proceedings of the 1999 European Symposium on Programming*, volume 1576 of *LNCS*, pages 208–223, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[77] George Necula et al. *CCured Documentation*, September 2003. `http://manju.cs.berkeley.edu/ccured/`.

[78] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL 2002)*, pages 128–139, London, United Kingdom, January 2002.

[79] Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-checking entire programs without recompiling. In *Informal Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2004)*, Venice, Italy, January 2004.

[80] Nicholas Nethercote and Alan Mycroft. The cache behaviour of large lazy functional programs on stock hardware. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance (MSP 2002)*, pages 44–55, Berlin, Germany, July 2002.

[81] Nicholas Nethercote and Alan Mycroft. Redux: A dynamic dataflow tracer. In *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA, July 2003.

[82] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA, July 2003.

[83] Flemming Nielson, Hanne Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[84] OC Systems. Aprobe. `http://www.ocsystems.com/`.

[85] Will Partain. The `nofib` benchmark suite of Haskell programs. In *Proceedings of the Glasgow Workshop on Functional Programming*, pages 195–202, Ayr, Scotland, July 1992.

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

[87] Bruce Perens. Electric Fence. `ftp://ftp.perens.com/pub/ElectricFence/`.

[88] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[89] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[90] Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, September 1998.

[91] Mark Probst. Dynamic binary translation. In *Proceedings of the UKUUG Linux Developers' Conference*, Bristol, United Kingdom, July 2002.

[92] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[93] Eric S. Raymond. *The Cathedral and the Bazaar*. O'Reilly, 1999.

[94] Steven P. Reiss and Manos Renieris. Languages for dynamic instrumentation. In *Proceedings of the ICSE Workshop on Dynamic Analysis (WODA 2003)*, pages 6–9, Portland, Oregon, May 2003.

[95] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, pages 1–7, Seattle, Washington, USA, August 1997.

[96] Michiel Ronsse, Jonas Maebe, Bastiaan Stougie, et al. DIOTA. `http://www.elis.ugent.be/diota/`.

[97] Graeme S. Roy. *mpatrol: A Library for controlling and tracing dynamic memory allocations*, January 2002. `http://www.cbmamiga.demon.co.uk/mpatrol/`.

[98] Olatunji Ruwase and Monica Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS04)*, San Diego, California, USA, February 2004.

[99] K. Scott, N. Kumar, S. Velusamy, B. Childers, J.W. Davidson, and M.L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the First Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2003)*, pages 36–47, San Francisco, California, USA, March 2003.

[100] Kevin Scott and Jack Davidson. Strata: A software dynamic translation infrastructure. In *Proceedings of the Workshop on Binary Translation (WBT '01)*, Barcelona, Spain, September 2001.

[101] Kevin Scott, Jack W. Davidson, and Kevin Skadron. Low-overhead software dynamic translation. Technical Report CS-2001-18, Department of Computer Science, University of Virginia, Charlottesville, Virginia, USA, 2001.

[102] Julian Seward, Nicholas Nethercote, Jeremy Fitzhardinge, et al. Valgrind. `http://valgrind.kde.org/`.

[103] Anthony M. Sloane. Generating dynamic program analysis tools. In *Proceedings of the Australian Software Engineering Conference (ASWEC'97)*, pages 166–173, Sydney, Australia, September 1997.

[104] Amitabh Srivastava, Andre Edwards, and Hoi Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, Redmond, Washington, USA, April 2001.

[105] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '94)*, pages 196–205, Orlando, Florida, USA, June 1994.

[106] Standard Performance Evaluation Corporation. SPEC CPU2000 benchmarks. `http://www.spec.org/`.

[107] Sun Microsystems. Shade 1.7.3 beta. `http://wwws.sun.com/software/download/products/3ff9c026.html`.

[108] Herman ten Brugge. Boundschecking project. `http://sourceforge.net/projects/boundschecking/`.

[109] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[110] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)*, pages 142–151, Montreal, Canada, June 1998.

[111] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29(2):128–170, September 1997.

[112] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles (SOSP-14)*, pages 203–216, Asheville, North Carolina, USA, December 1993.

[113] David W. Wall. Limits of instruction-level parallelism. Research Report 93/6, Digital Western Research Laboratory, Palo Alto, California, USA, November 1993.

[114] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, 3rd edition, 2000.

[115] Josef Weidendorfer. KCachegrind. `http://kcachegrind.sourceforge.net/`.

[116] Josef Weidendorfer. Performance analysis of GUI applications on Linux. In *Proceedings of the KDE Developers' Conference (Kastle 2003)*, Nové Hrady, Czech Republic, August 2003.

[117] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1975.

[118] Roland Wismüller, Jörg Trinitis, and Thomas Ludwig. OCM — a monitoring system for interoperable tools. In *Proceedings of the Second SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pages 1–9, Welches, Oregon, USA, August 1998.

[119] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 68–79, Philadelphia, Pennsylvania, USA, May 1996.

[120] Yichen Xie, Andy Chou, and Dawson Engler. Archer: Using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2003)*, pages 327–336, Helsinki, Finland, September 2003.

[121] Andreas Zeller. Program analysis: A hierarchy. In *Proceedings of the ICSE Workshop on Dynamic Analysis (WODA 2003)*, pages 6–9, Portland, Oregon, May 2003.