Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation

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

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

Abstract

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

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

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

General Terms Design, Performance, Experimentation

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

1. Introduction

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

1.1 Dynamic Binary Analysis and Instrumentation

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

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

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

1.2 Shadow Value Tools and Heavyweight DBA

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.3 Contributions

This paper makes the following contributions.

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

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

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

2. Shadow Value Requirements

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3. How Valgrind Works

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

3.1 Basic Architecture

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

3.2 Execution Overview

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

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

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

3.3 Starting Up

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

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

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

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

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

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

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

3.4 Guest and Host Registers

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

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

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

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

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

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

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

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

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

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

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

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

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

3.6 Valgrind's IR

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

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

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

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

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

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

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

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

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

Figure 1. Disassembly: machine code → tree IR

3.7 Translating a Single Code Block

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.8 Storing Translations

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

3.9 Executing Translations

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

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

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

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

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

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

3.10 System Calls

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

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

3.11 Client Requests

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

3.12 The Events System

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

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

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

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

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

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

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

3.13 Function Replacement and Function Wrapping

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

3.14 Threads

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

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

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

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

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

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

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

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

3.15 Signals

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

3.16 Self-modifying Code

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

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

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

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

4. Valgrind's Shadow Value Support

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5. Evaluation

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

5.1 Tool-writing Ease

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

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

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

5.2 Tool Robustness

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

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

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

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

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

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

5.3 Tool Instrumentation Capabilities

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

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

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

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

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

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

5.4 Tool Performance

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.5 Summary

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

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

6. Related Work

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

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

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

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

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

7. Future Work and Conclusion

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

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

Acknowledgments

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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