Electronic Notes in Theoretical Computer Science 89 No. 2 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume89.html 23 pages

# Valgrind: A Program Supervision Framework

Nicholas Nethercote [a,1] and Julian Seward [b,2]

[a] *Computer Laboratory, University of Cambridge*
*Cambridge, United Kingdom*

[b] *Cambridge, United Kingdom*

**Abstract**

Valgrind is a programmable framework for creating program supervision tools such as bug detectors and profilers. It executes supervised programs using dynamic binary translation, giving it total control over their every part without requiring source code, and without the need for recompilation or relinking prior to execution.

New supervision tools can be easily created by writing *skins* that plug into Valgrind's *core*. As an example, we describe one skin that performs Purify-style memory checks for C and C++ programs.

## 1 Introduction

Valgrind is a meta-tool: a tool for making tools. Programmers can use it to build tools that supervise almost any aspect of a program's execution, such as profilers, bug detectors, and tools that deduce program properties.

### *1.1 Overview*

Valgrind uses dynamic binary translation to provide complete control over a program. This approach has two major advantages.

(i) Coverage: It covers all the code of a program, and all the libraries it uses, even if the source code is not available. Only system calls are not directly under Valgrind's control, and even they can be indirectly observed.

(ii) Convenience: Neither a supervised program nor its libraries need to be recompiled, relinked, or altered in any other way before being run.

Tools are written in C. The simplest tools can be defined by writing only four functions, requiring only a few lines of code. Creating a new tool is simple

[1] Email: njn25@cam.ac.uk
[2] Email: jseward@acm.org

enough that it is feasible to create domain-specific tools that work with only one program or a small number of programs. However, Valgrind is also suited to writing more complex, very powerful supervision tools.

### *1.2 History and Current Status*

Valgrind was first released in early 2002 as a monolithic memory checker for C and C++ programs, *à la* Purify [16]. When one of this paper's authors needed a cache profiler, he added it to Valgrind because it provided an ideal infrastructure. Valgrind was then used to experiment with automatic invariant detection. When further extensions were suggested, we decided, rather than continually adding new capabilities in an ad hoc and confusing fashion, that Valgrind should be recast as a meta-tool for program supervision. Large chunks were re-written, separating tool-specific *skins* from the generic *core*.

Valgrind's core contains the low-level infrastructure to support program instrumentation. This includes the x86-to-x86 just-in-time (JIT) compiler, a basic C library replacement, support for signal handling, and a scheduler (for threaded programs). It also provides some services that are useful to some but not all skins, such as functions to read debug information, record and suppress errors, etc. It handles most of the fiddly tasks that a supervision tool author would rather not worry about. Each skin's main job is to instrument all code that passes through the core's JIT compiler.

Valgrind is free (GPL) software. It runs on most x86 machines running Linux, and is robust enough to run large programs, such as Mozilla and OpenOffice. The Valgrind distribution includes the core and several skins:

- Memcheck: The original Purify-style memory checker;
- Addrcheck: A more light-weight memory checker that only checks whether each memory access is to an addressable location;
- Cachegrind: A cache profiler that supports line-by-line source annotations of instruction and data cache misses;
- Helgrind: A data-race detector that uses the Eraser algorithm [25];
- Nulgrind: the "null" skin that performs no instrumentation.

### *1.3 Contributions of This Work*

We claim it is easier to write powerful supervision tools with Valgrind than with other, similar program instrumenters. This is because it was designed first and foremost as an instrumenter, not as a dynamic translator that later had instrumentation bolted on. In particular, unlike other tools, Valgrind uses a platform-independent intermediate format (called UCode) that is expressed using virtual registers. This level of abstraction allows a skin to add instrumentation without being constrained by the original code (e.g. do I have enough spare registers?) and without fear of changing the original code's effects (e.g. does my instrumentation alter the machine's condition codes?).

Skins can even extend UCode with new instructions, which allows very fine-grained instrumentation. Also, Valgrind provides commonly needed services such as debug information reading, error reporting, register shadowing, etc.

In addition, Valgrind supports *client requests*, which allow supervised programs to pass information to a supervision tool. This opens up the possibility of easily combining static and dynamic analysis.

### *1.4 Paper Organisation*

This paper is structured as follows. Section 2 describes Valgrind's core, Section 3 describes the core/skin interface, and Section 4 describes skins, using Memcheck as an example. Section 5 discusses performance, and Section 6 considers related work. Section 7 discusses future work and concludes.

## 2 The Core

Valgrind's core provides the base execution mechanism for supervised programs. This section describes its main features.

### *2.1 JIT Compiler*

The most important part of Valgrind's core is the x86-to-x86 JIT compiler. Valgrind [3] works with ordinary dynamically-linked ELF executables, and does not require them to be recompiled, relinked, or altered before they are run. The core is packaged as a shared object (`valgrind.so`) and loaded alongside the *client program* being supervised, using the `LD_PRELOAD` environment variable. Importantly, `valgrind.so` is linked with the `-z initfirst` flag, which ensures that its initialisation code is run before that of any other object in the loaded image.

When this initialisation code is run, Valgrind gains control. The real CPU becomes trapped in `valgrind.so` and the translations it generates. When the initialisation function returns, normal start-up actions, orchestrated by the dynamic linker `ld.so`, continue as usual, but on Valgrind's *simulated* CPU. Valgrind never runs any part of the client program directly.

#### *2.1.1 Basics*

To avoid the notorious complexity of x86 code, Valgrind uses *UCode*, a RISC-like two-address intermediate language. The translation performed by the JIT compiler is thus not x86-to-x86, but rather x86-to-UCode-to-x86. [4]

Each register for the simulated CPU is stored in memory, in a block called the `baseBlock`, which is permanently pointed to by register `%ebp` for easy

[3] We will use the terms "Valgrind", "Valgrind's core" and "the core" interchangeably.

[4] One might also think that UCode makes Valgrind easy to port to other architectures. See Section 7.1 for a caveat about this.

```
movl $0xFFFF,%ebx     0: MOVL      $0xFFFF, t0
                      1: PUTL      t0, %EBX
                      2: INCEIPo   $5
andl %ebx,%eax        3: GETL      %EAX, t2
                      4: GETL      %EBX, t4
                      5: ANDL      t4, t2  (-wOSZACP)
                      6: PUTL      t2, %EAX
                      7: INCEIPo   $2
ret                   8: GETL      %ESP, t6
                      9: LDL       (t6), t8
                     10: ADDL      $0x4, t6
                     11: PUTL      t6, %ESP
                     12: JMPo-r    t8
```

Fig. 1. x86 code to UCode

access. Their contents are loaded from the `baseBlock` into memory when needed, and if the values change, the `baseBlock` values must be updated again by the end of the basic block. But they need not be updated in the middle of a basic block. [5]

The condition codes register is also stored in the `baseBlock`. If instruction $X$ affects the condition codes, and then a later instruction $Y$ clobbers those changes, the condition code update for $X$ can be skipped.

### *2.1.2 Translating Basic Blocks*

A basic block ends upon any control transfer instruction, e.g. a jump, call, return. If control transfers to the middle of an already translated basic block, its second half will be translated again and stored separately; in practice this is uncommon. One basic block is translated at a time, in the following steps.

(i) Disassembly: Each x86 instruction is independently disassembled into one or more UCode instructions, expressed in terms of virtual registers. The code produced fully updates the simulated register set in memory for every x86 instruction. The translation is straightforward but tedious due to the complexity of the x86 instruction set.

    Figure 1 gives an example, using AT&T assembler syntax. The simulated registers are called `%EAX`, `%EBX`, etc. Virtual registers are named `t0`, `t2`, etc. `PUT` and `GET` move values in and out of the simulated registers. `INCEIP` instructions mark where the UCode for each x86 instruction ends; the argument gives the length of the original x86 instruction.

(ii) Optimisation: Removes redundant UCode introduced by the simplistic disassembler. In particular, many of the simulated registers can be kept

[5] This means Valgrind cannot simulate precise exceptions. In our experience, this is not a serious limitation.

```
 0: MOVL      $0xFFFF, %eax             movl   $0xFFFF, %eax
 1: PUTL      %eax, %EBX                movl   %eax, 0xC(%ebp)
 2: INCEIPo   $5                        movb   $0x18, 0x24(%ebp)
 3: GETL      %EAX, %ebx                movl   0x0(%ebp), %ebx
 4: ANDL      %eax, %ebx (-wOSZACP)     andl   %eax, %ebx
 5: PUTL      %ebx, %EAX                movl   %ebx, 0x0(%ebp)
 6: INCEIPo   $2                        movb   $0x1A, 0x24(%ebp)
 7: GETL      %ESP, %ecx                movl   0x10(%ebp), %ecx
 8: LDL       (%ecx), %edx              movl   (%ecx), %edx
 9: ADDL      $0x4, %ecx                pushfl;  popl 32(%ebp)
                                        addl   $0x4, %ecx
10: PUTL      %ecx, %ESP                movl   %ecx, 0x10(%ebp)
11: JMPo-r    %edx                      movl   %edx, %eax
                                        ret
```

Fig. 2. UCode to x86 code

in a real register until their last use in the basic block. For the example in Figure 1, this phase deletes the `GET` at instruction #4, and renames `t4` as `t0` in instruction #5.

(iii) Instrumentation: Added by the skin. For clarity, our example adds no instrumentation.

(iv) Register allocation: Assigns each virtual register to the six real, freely usable general-purpose registers: `%eax`, `%ebx`, etc. Spill code is generated as needed. The linear-scan register allocator [29] does a fairly good job; importantly it passes over the basic block only twice, minimising compilation times. The left side of Figure 2 shows the results of allocation.

(v) Code generation: Each UCode instruction is translated independently. Most turn into a small number of x86 instructions, but some call assembly code or C helper functions. The right side of Figure 2 continues our example. `INCEIP` updates the simulated program counter, `%EIP`; if the update only modifies the least-significant byte of the program counter, we use a `movb` instead of an addition to update it. The generated code for instruction #9 stores the final condition codes in the `baseBlock`.

#### *2.1.3 Connecting Basic Blocks*

Translated basic blocks are stored in a table that can hold about 160,000 translations. Translations are evicted when necessary using a FIFO policy.

At the end of each basic block, one of three things can happen.

(i) Jumps to addresses known at compile-time are translated into direct jumps to the relevant translation (giving *chains* of directly connected basic blocks). Even in the worst case, around 70% of jumps are chained.

(ii) If a basic block does not end with a chained jump, the translation jumps back to an assembly code dispatch loop, carrying the original code address of the next block to run. The dispatcher searches for its translation in a small direct-mapped translation cache, which has a hit rate of around 98%. If that fails, a full search of the old-to-new code address table is conducted. If the translation is found, it is executed immediately.

(iii) If no translation is found, control falls out of the dispatcher back to Valgrind's main event scheduler, and a new translation is made.

Furthermore, Valgrind must periodically check whether there are any outstanding signals to be delivered, and whether a thread switch is due. To support this, all blocks begin with a preamble (not shown in Figures 1 and 2) which decrements a counter and falls out to the scheduler when the counter hits zero. If a translation does certain other tasks, such as make a system call or handle a client request, control also returns to the scheduler.

Although no part of the client program is executed directly on the real CPU, Valgrind does all its work (such as translation) on the real CPU.

#### *2.1.4 System Calls*

System calls are performed on the real CPU. For system calls to work correctly, Valgrind must make it look like the client is running normally, but not lose control of program execution. The steps the core takes are as follows.

(i) Save Valgrind's stack pointer;

(ii) Copy the simulated registers, except the program counter, into the real registers;

(iii) Do the system call;

(iv) Copy the simulated registers back out to memory, except the program counter;

(v) Restore Valgrind'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.

#### *2.1.5 Floating Point, MMX and SSE Instructions*

We take some liberties to simplify handling of x86 floating point (FP) instructions. 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. To run a simulated FP instruction on the real CPU, the simulated FPU state is loaded into the real CPU, the instruction is executed, and the real FPU state is then copied back to the simulated state (stored in `baseBlock`). Effort is made to avoid redundant 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. This arrangement sidesteps the need to simulate the FPU's internal state, whilst still making FPU load/store addresses and data widths available to skins. A similar approach is used for handling MMX, SSE and SSE2 instructions.

#### *2.1.6 Client Requests*

Valgrind's core has a trapdoor mechanism that allows a client program to pass signals and queries, or *client requests*, to a skin. This is done by inserting into the client program a short sequence of instructions that are effectively a no-op (six highly improbable, value-preserving rotations of register `%eax`). When Valgrind spots this sequence of instructions during x86 disassembly, the resulting translation causes control to drop into its code for handling client requests. Arguments can be passed to client requests, and they can return a value to the client.

Client requests can be embedded in any program written in any language in which assembly code or C code can be embedded; a macro makes this easy. The client just needs to be recompiled with the client requests inserted; 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.

#### *2.1.7 Self-modifying Code*

Valgrind does not directly support self-modifying code.[6] However, there is indirect support, via the `VALGRIND_DISCARD_TRANSLATIONS` client request, which tells it to discard any translations of x86 code in a certain address range. Thus, with some effort, Valgrind can work with programs that dynamically generate or modify code.

#### *2.1.8 Termination*

Eventually the client program calls the `exit()` system call, indicating that it wishes to quit. Valgrind halts the simulated CPU, performs its final actions (such as printing out final results from the active skin), and calls `exit()` itself, passing to the kernel the exit code that the client gave it.

#### *2.1.9 Ensuring Correctness*

The correctness of Valgrind's JIT compiler is paramount. The code is littered with assertions, and the core periodically sanity checks various critical pieces of state. These measures have found many bugs during Valgrind's development, and contributed immensely to its stability.

One important design decision was to ensure Valgrind could stop simulated CPU and revert back to the real CPU part way through a program's execution. Bugs in the JIT compiler can be pin-pointed using a binary search on the reversion point. This decision did constrain Valgrind's design in two minor

[6] It did once, but it was too complicated and not useful enough to be worthwhile.

ways. Firstly, the layout of memory seen by the client must be identical regardless of whether it is running on the real or simulated CPU; this precluded data-address swizzling. Secondly, Valgrind cannot run on the same stack as the client; instead it uses its own stack which it switches to at start-up.

### *2.2 C Library*

Valgrind does not rely on the GNU standard C library (`glibc`) at all, or any other shared objects or libraries. This decision was made to avoid the possibility of subtle bugs caused by running C library code both on the simulated CPU (for the client program) and the real CPU (for the core and skins).[7] Also, `glibc` has different definitions for some types (e.g. `sigset_t`) to the Linux kernel, and Valgrind uses the kernel's definitions. The only header files that Valgrind includes are kernel header files.

Because of this, Valgrind's core provides a reimplementation of some standard C library functions, such as `printf()`, `mmap()`, `open()`, `sigaction()`, etc. Each one is prepended with a unique (we hope!) prefix, to avoid name clashes, since Valgrind's text and data segments share the same namespace as the client program.

### *2.3 Signals*

Unix signal handling presents special problems for all user-process emulators: 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. We cannot allow this to happen, since the handler will run on the real CPU and not the simulated one. Even worse, if the handler does not return but instead does a `longjmp`, Valgrind will permanently lose control.

Instead, Valgrind intercepts the `sigaction()` and `sigprocmask()` system calls, which are used to register signal handlers. Valgrind notes the address of the signal handler specified, and instead requests the kernel delivers that signal to its own handler. When a signal arrives, Valgrind notes that the signal is now pending for the client, but does not deliver the signal immediately, unless it indicates an exception which is non-resumable according to POSIX semantics (segmentation fault, bus error, or floating point exception).

Every few thousand basic blocks, any pending signals are delivered to the client. Signal delivery frames are built on the client's stack, and the handler code is then run on the simulated CPU. If a signal frame is observed to return, Valgrind removes the frame from the client's stack and resumes executing the client wherever it was before the frame was pushed.

Interactions with signal masks, with signals interrupting system calls, and with threads, make the signal simulation machinery complex and fragile. Nevertheless it works well enough to run almost all applications, and in particular

[7] At least one astonishingly obscure bug was caused by an accidental `glibc` dependency.

POSIX signal semantics work well.

### *2.4 Pthreads Implementation*

How should threads be handled in this framework? Valgrind could run instrumented code in separate threads, one per child thread in the client program. This sounds simple, but it would be complex and slow. All Valgrind's internal data structures would need to be suitably threaded and locked. This might be viable. However, skins would also have to lock skin-specific data structures. For Memcheck (and several other skins), this would mean locking shadow memory (see Section 4.1) for every load/store done by the client, which would be too slow. The reason is that originally-atomic loads and stores can become non-atomic in instrumented code. Under Memcheck, each load or store translates into a load or store of shadow memory, followed a few instructions later by the original load/store. It was unclear how to guarantee, efficiently, that when multiple threads accessed 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 (p)threads model, providing its own binary-compatible replacement for the standard `libpthread` pthread library. This, combined with Valgrind's core, provides a user-space threads package. All threads are run on a single kernel thread, and all thread switching and scheduling is entirely under Valgrind's control. The standard abstractions are provided—mutexes, condition variables, etc.

This scheme works well enough to run most threaded programs, including large applications such as Mozilla, OpenOffice and MySQL. 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 skins.[8]

Unfortunately, the reimplementation of `libpthread` greatly complicates Valgrind's core, particularly when dealing with signals and ensuring that system calls such as `read()` block only the calling thread and not the whole of Valgrind. Furthermore, the system `libpthread` interacts closely with `glibc` and renders our version very susceptible to changes in the C library. We are considering a compromise scheme, in which the system `libpthread` is used, but Valgrind still schedules 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 Valgrind can easily intercept the `clone()` system call with which the standard library starts a new thread. However, it is unclear whether this scheme will work, and whether it will simplify matters.

### *2.5 Limitations*

Valgrind does not yet support the new Native POSIX Thread Library (NPTL) for Linux, although we are working on this. A workaround is to use the

[8] These are critical for the data-race detection skin, Helgrind.

`LD_ASSUME_KERNEL` variable, so that the old LinuxThreads library is used instead. Also, statically linked programs cannot be run under Valgrind, due to its (ab)use of the dynamic linker to gain control at program start-up.

## 3 Core/Skin Interface

Valgrind's core leaves certain critical functions undefined, which a skin must supply. Most notably, skins define how program code should be instrumented. Each skin defines a separate program supervision tool. Writing a new tool just requires writing a new skin. The core takes care of the rest.

### *3.1 Execution Spaces*

To understand skins, one must understand the three *spaces* in which a client program's code executes (from the skin's point of view), and the different levels of control that a skin has over these spaces.

(i) User space: Covers all code that is JIT compiled. The skin 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.

(ii) Core space: Covers the small proportion of the program's execution that takes place entirely within Valgrind's core. It includes signal handling, and pthread operations and scheduling. A skin cannot instrument these operations, as it never sees their code. However, the core provides hooks so a skin can be notified when certain interesting events happen, such as a signal being delivered memory, a pthread mutex being locked, etc. These hooks are described in more detail in Section 3.3, with examples in Section 4.4.

(iii) Kernel space: Covers execution in the operating system kernel. System call internals cannot be directly observed by either the skin or the core, but the core built-in knowledge about what each system call does with its arguments, and it provides a hook allowing skins to wrap system calls, so they can be aware of their execution. All other kernel activity (e.g. process scheduling) is opaque to Valgrind and irrelevant to its execution.

Skins only see code executed in user space. This is the vast majority of code, but note that any profiling information recorded will not be exhaustive.

### *3.2 How Skins Work*

Skins must define four functions that are called by Valgrind's core, and it is important that programmers can write and distribute skins that can be plugged into Valgrind's core in a modular and binary-compatible way.

To achieve this, each skin is packaged into a separate shared object which is loaded ahead of `valgrind.so`, again using the `LD_PRELOAD` variable. The functions that must be defined by the skin are declared as weak symbols in the core; definitions of them in the skin override the core's definition. The core versions of these function abort immediately if ever called.

One copy of the core shared object `valgrind.so` is shared by all the skins on a system. Valgrind is installed so that selecting a skin simply requires using the `--skin` option of the Valgrind start-up script.

### *3.3 Required Functions*

A skin must define at least four functions, for initialisation, instrumentation, and finalization, described in the following sections.

#### *3.3.1 Initialisation*

The functions `vgSkin_pre_clo_init()` and `vgSkin_post_clo_init()` are called before and after command-line processing occurs.[9] Most importantly, there are three structures in which a skin can set fields.

(i) *Details*: This structure contains the skin's name, copyright notice, version number, etc. These details are used when constructing the skin's start-up message, and must be filled in by a skin, or the core aborts execution.

(ii) *Needs*: This structure contains several boolean fields. If a *need* is set (they default to *false*), it indicates that the skin wishes to use a service provided by the core. Most services require the skin to define some extra functions. For example, to record and report errors, a skin can set the `skin_errors` need, and then must define several functions for comparing errors, printing errors, etc. This is much easier than doing error handling from scratch. See Section 4.2 for examples of needs.

(iii) *Trackable Events*: This structure can be used by a skin to indicate which core space events it is interested in. To declare its interest, the skin sets the relevant pointer in the structure to point to a function, which will be called when that event happens. For example, if the skin sets the `post_mutex_lock` function pointer, the assigned function will be called each time a mutex is locked.

The main difficulty with trackable events is predicting which are interesting to skins. We have added all the hooks that are necessary for the skins distributed with Valgrind (quite a number), plus a few more obviously useful ones, and we are prepared to add new hooks for skin-writers as they need them. New events can be added to the core in a binary-compatible way that does not break old skins.

As well as initialising these structures, a skin must register any C functions to be called from instrumented code, and do any other initialisation it needs.

[9] Both are needed; `vgSkin_pre_clo_init()` so a skin can declare that it wants to process command line options, and `vgSkin_post_clo_init()` so it can do any initialisation that relies on the command line options given.

#### 3.3.2 Instrumentation

The function `vgSkin_instrument()` is called to instrument UCode. The easiest way to do this is to insert calls to C functions, using the UCode instruction `CCALL`, at interesting program points. C calls are efficient—the code generator preserves only live caller-save registers across calls, and we allow called functions to utilise gcc's `regparms` attribute so that their arguments are passed in registers instead of on the stack. If we are lucky with liveness, and arguments are in the right registers already, a C call requires just a single `call` instruction. A more complicated way to add instrumentation, albeit one that might result in better performance if a skin does fine-grained instrumentation, is to extend UCode with new instructions. Memcheck does this (see Section 4.2.5).

Because UCode is instrumented one basic block at a time, basic block-level instrumentation is easy. Instrumentation at the x86-instruction level is also possible, thanks to the `INCEIP` instruction which groups together all UCode instructions from a single x86 instruction. However, function-level instrumentation is surprisingly difficult. How do we know if a basic block is the first in a function? There is no tell-tale instruction sequence at a function's start, and we cannot rely on spotting a `call` instruction, because functions in dynamically linked shared objects are called using a `jmp`. Instead we 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 it is the best we can do.

#### 3.3.3 Finalization

The function `vgSkin_fini()` lets the skin do any final processing, such as presenting the final results, writing a log file, etc.

### 3.4 Skin Output

By default, skin output goes to `stderr`. Core command line options can be used to redirect it to a given file descriptor, file, or network socket. A skin can also write directly to file using Valgrind's C library file functions.

### 3.5 Replacing Library Functions

Just as a skin can overwrite functions defined in the core, it can overwrite functions defined in libraries. Section 4.3 gives examples. In particular, the core provides some support to make it easier to replace `malloc()`, `free()` and friends, since knowing about heap operations is important for many skins.

## 4 Memcheck: An Example Skin

This section describes Memcheck, the memory checker skin that is distributed with Valgrind. Although it describes some of the skin's mechanics, this section is intended to show how skins work, and what they can achieve, rather than be a detailed description of Memcheck itself.

### *4.1 Overview*

Memcheck is a Purify-style memory checker designed primarily for C and C++ programs. When a program is run under Memcheck, all used memory is tracked, and all memory accesses are checked. It can detect these errors:

- Use of uninitialised memory;
- Accessing memory after it has been freed;
- Accessing memory past the end of heap blocks;
- Accessing inappropriate areas on the stack;
- Memory leaks, where pointers to heap blocks are lost;
- Passing of uninitialised and/or unaddressable memory to system calls;
- Mismatched use of `malloc()`/`new`/`new[]` vs. `free()`/`delete`/`delete[]`;
- Overlapping source and destination areas for `memcpy()`, `strcpy()`, etc.

The skin shadows each byte of memory used in the original program with nine status bits. Shadows are added in 64KB segments as new memory is used. One of the shadow bits, the *A* (addressability) bit, indicates whether the byte is currently addressable by the program. The relevant A bit(s) are checked for every memory access, so as to detect invalid references to memory on a per-byte granularity.

The other eight *V* (validity) bits indicate which bits in the byte have defined values, according to some loose understanding of C's semantics. The V bits are used to detect if any of the following operations depend on uninitialised values: conditional tests and moves, system calls, and memory address computations. The V bits are not checked simply when a value is read, because partially defined words are often copied around, due to the common practice 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. A faster alternative is to use Addrcheck, which is based on Memcheck, but only uses A bits, and thus reports fewer errors.

### *4.2 Services Used*

Memcheck is a good example skin because it uses most of the services provided by the core. The more interesting ones follow.

#### *4.2.1 Error Recording*

As mentioned above, to use the error recording service a skin must provide definitions of several functions for comparing and printing errors, reading files containing suppressions (descriptions of errors which the user does not want to see), etc. The core reports each error only once, to prevent the user from being flooded with duplicate error reports.

#### *4.2.2 Debug Information*

Valgrind's core provides functions that take an instruction address and search the program's debug information to find the name of executable or shared object it came from, and also its origin source file name, function name and line number. The debug information (if present) is used to identify the exact line at which memory errors occur.

#### *4.2.3 Shadow Registers*

Just as each memory location is shadowed with eight V bits and one A bit, each (32-bit) register is shadowed with 32 V bits—the A bit is not necessary as registers are always addressable. The skin must define one function that defines the initial value of the shadow registers. During execution, Memcheck handles register V bits similarly to memory V bits.

#### *4.2.4 Client Requests*

Skins can define their own client requests. If a request is not recognised by the core, it can be passed to the skin which can interpret it as it likes. Each client request has a unique number; a two-letter code identifies which skin it belongs to. Unrecognised client requests are ignored, so that client requests for more than one skin can be embedded in a client program.

#### *4.2.5 Extended UCode*

Extending UCode's instruction set does not make a skin any more powerful—the same effect can be achieved by using calls to C functions—but it can help performance, by effectively inlining instrumentation. Memcheck adds a lot of fine-grained instrumentation, typically 1–3 x86 instructions per UCode instruction. Calling a C function for each one would slow it down greatly.

To extend UCode, a skin must define a function that, for each new instruction, indicates its register use (for register allocation), and another that generates its code. The core provides functions to assist code generation. Memcheck defines ten new UCode instructions, each of which mirrors a normal UCode instruction, but works with the V bits of the relevant value instead of the value itself.

### *4.3 Replacement Library Functions*

Memcheck replaces the standard C and C++ memory management functions (`malloc()`, `free()`, etc.) with its own definitions. These replacements run on the simulated CPU, but use a client request to transfer control to the real CPU, so that the core's low-level memory management routines can be used.

There are four reasons to use replacements. First, to track when heap memory is allocated, moved, and deallocated, in order to maintain the correct A and V bits. Second, to flank allocated blocks with *redzones*—unused areas at their edges—which can help with detection of block overruns and underruns. Third, to record additional information for each heap block, allowing detection of errors such as bad deallocations (e.g. trying to deallocate a non-heap block), and mismatched use of `malloc()`/`new`/`new[]` vs. `free()`/`delete`/`delete[]`. Also included is an *execution context*, a snapshot of the program counter and the top few return addresses on the stack from the time of allocation. This is vital for writing informative error messages about heap blocks. Finally, Memcheck postpones heap block deallocation by storing freed blocks in a list for a short time, which allows it to catch erroneous accesses to freed blocks.

Memcheck also replaces some C string functions: `strlen()`, `strcpy()`, `memcpy()`, etc. This is because their x86 `glibc` implementation uses highly optimised assembly code which is not handled well by Memcheck, causing it to emit many spurious errors. Simpler versions do not cause these problems. Also, these replacement versions can check that the source and destination memory blocks do not overlap in `memcpy()`, `strcpy()`, etc. These functions are run on the simulated CPU.

### *4.4 Events Tracked*

Memcheck registers functions so that it is notified when the system calls that affect memory permissions occur: `mmap()`, `brk()`, `mprotect()`, `mremap()`, `munmap()`. Also, 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()`). Such notification can be done because each system call's behaviour is clearly defined, and known by the core.

The core also provides hooks for notifying skins when the stack grows and shrinks. Skins could do this, since stack pointer (`%esp`) manipulations are entirely visible from the UCode. But it is fiddly and must be well optimised—`%esp` is changed *very* frequently—so the core does it. Common cases (when `%esp` is changed by 4, 8, 12, 16 or 32 bytes) are optimised with unrolled loops.

### *4.5 Other Initialisation*

In addition to initialising the details, needs and tracked events structures, Memcheck also registers twelve C functions that it calls from instrumented code (e.g. when an error is detected), and initialises the A and V shadow bits for all bytes of memory accessible at program start-up.

### *4.6 Instrumentation*

Each UCode instruction is instrumented individually, with instrumentation added immediately before it. UCode instructions that move values around

| Program | Time (s) | Nulgrind | Memcheck | Addrcheck | Cachegrind |
|---|---|---|---|---|---|
| bzip2 | 10.7 | 2.4 | 13.6 | 9.1 | 31.0 |
| crafty | 3.5 | 7.2 | 44.6 | 26.5 | 107.4 |
| gap | 0.9 | 5.4 | 28.7 | 14.4 | 46.6 |
| gcc | 1.5 | 8.5 | 36.2 | 23.6 | 73.2 |
| gzip | 1.8 | 4.4 | 20.8 | 14.5 | 50.3 |
| mcf | 0.3 | 2.1 | 11.6 | 5.9 | 18.5 |
| parser | 3.3 | 3.7 | 17.4 | 12.5 | 34.8 |
| twolf | 0.2 | 5.2 | 29.2 | 18.5 | 53.3 |
| vortex | 6.5 | 7.5 | 47.9 | 32.7 | 88.4 |
| ammp | 18.9 | 1.8 | 24.8 | 21.1 | 47.1 |
| art | 26.1 | 5.9 | 14.1 | 11.5 | 19.4 |
| equake | 2.1 | 5.5 | 32.7 | 28.0 | 49.9 |
| mesa | 2.7 | 4.7 | 41.9 | 31.6 | 64.5 |
| median | | 5.2 | 28.7 | 18.5 | 49.9 |

Table 1
Slowdown factor of four skins

(e.g. `LOAD`) are instrumented with the corresponding extended UCode instruction that does the same for the shadow V bits (e.g. `LOADV`). In other places A and V bit tests are inserted, and for arithmetic operations instrumentation is inserted to compute the V bits of the result from the V bits of the operands.

This first pass can be quite naïve about adding instrumentation, as any excess instrumentation is removed by a subsequent optimisation pass that performs constant-propagation and constant-folding of operations on V bits.

### *4.7 Finalization*

Memcheck finishes by printing some basic memory statistics (number of bytes allocated, freed, etc.), summarising any found errors, and running its leak checker if the `--leak-check=yes` option was specified.

## 5 Performance

This section discusses Valgrind's performance. All experiments were performed on an 1400 MHz AMD Athlon with 1GB of RAM, running Red Hat Linux 7.1, kernel version 2.4.19. The test programs are a subset of the SPEC2000 suite. All were tested with the "test" (smallest) inputs.

Table 1 shows the time performance of four skins. Column 1 gives the benchmark name, column 2 gives its normal running time in seconds, and columns 3–6 give the slowdown factor for each skin. Programs above the line are integer programs, those below are floating point programs.

Table 2 shows the post-instrumentation code expansion for the four skins.

| Program | Size (KB) | Nulgrind | Memcheck | Addrcheck | Cachegrind |
|---|---|---|---|---|---|
| bzip2 | 34 | 5.2 | 12.1 | 6.8 | 9.1 |
| crafty | 156 | 4.5 | 10.9 | 5.9 | 8.2 |
| gap | 140 | 5.6 | 12.7 | 7.3 | 9.7 |
| gcc | 564 | 5.9 | 13.1 | 7.6 | 9.9 |
| gzip | 30 | 5.5 | 12.6 | 7.2 | 9.4 |
| mcf | 30 | 5.7 | 13.5 | 7.7 | 9.9 |
| parser | 97 | 6.0 | 13.6 | 7.8 | 10.1 |
| twolf | 114 | 5.2 | 12.2 | 7.0 | 9.3 |
| vortex | 234 | 5.8 | 13.2 | 8.1 | 10.1 |
| ammp | 68 | 4.7 | 11.7 | 7.1 | 9.5 |
| art | 24 | 5.5 | 13.0 | 7.5 | 9.8 |
| equake | 44 | 5.0 | 12.2 | 7.1 | 9.2 |
| mesa | 69 | 4.8 | 11.2 | 6.7 | 8.9 |
| median | | 5.5 | 12.6 | 7.2 | 9.5 |

Table 2
Code expansion of four skins

Column 1 gives the benchmark name, column 2 gives its normal code size in kilobytes, and columns 3–6 give the code expansion factor for each skin. In addition to the space used by instrumented code, the core uses some extra memory, and each skin also introduces its own space overhead: Memcheck uses an extra 9 bits per byte of addressable memory, Addrcheck uses an extra 1 bit per byte, and Cachegrind uses 32–80 bytes per x86 memory-accessing instruction translated. Note that the slowdown and code expansion factors for each skin in do not correlate, because instrumentation speed varies greatly. In particular, Cachegrind's instrumentation includes many calls to C functions that update the simulated cache state, so its code expansion factor is relatively low, but its slowdown factor is high.

The time and space figures are quite high. In several ways—the use of UCode being the most obvious—Valgrind sacrifices performance in favour of making instrumentation easier, more flexible and more powerful. In practice, performance is quite acceptable. As an extreme example, we have used Addrcheck to check all processes running in a KDE-3.0.3 desktop session. Using a 1.7 GHz P4 with 512 MB of memory, performance was hardly stellar, but still quite usable. Judging from extensive user feedback, performance is a minor issue; stability and correctness are much more important.

## 6 Related Work

We first consider program checkers. Static checkers work directly on source code, not executing a program. Their main advantage is that their analyses are typically conservative and thus sound, so conclusions found are guaranteed to always hold. By contrast, dynamic checkers instrument a program (before or at run-time) and then observe execution. They only consider executed paths, and so cannot be sound. However, they work with real values, and do not have to predict all possible outcomes, so their conclusions can be much stronger. The two approaches are complementary. We then consider tools for instrumenting programs.

### *6.1 Static Checkers*

The most basic static analysis tools are very familiar: type checkers, gcc's `-Wall` option, Lint [18] and its successors (e.g. [10]), and so on.

Engler *et al*'s tool xgcc [8,12] uses an intriguing approach. It performs syntax-based analysis of C programs, working from analysis specifications written in a state-based pattern language called Metal. Each specification is small (e.g. 100 lines of code) and finds a specific kind of bug, such as null pointer dereferences. Although not released publicly, the system has been used to find hundreds of bugs, of more than ten different kinds, in the Linux and BSD kernels. Programmability is a great strength of this approach; for example, the authors improved analyses by adding domain-specific knowledge about the behaviour of certain functions in the Linux kernel. Valgrind shares a similar levels of programmability, and xgcc/Metal's success with domain-specific checks shows that domain-specific Valgrind skins might be useful too.

### *6.2 Dynamic Checkers*

Many C and C++ memory checkers exist. Purify [16] is a well known commercial system. It performs a similar level of memory checking as Memcheck; its checking is slightly coarser but also faster.

Other dynamic memory checkers typically provide custom checked wrappers for `malloc()` and `free()` that use a combination of pattern fills and boundary blocks to detect uninitialised memory references and array/block overflows. Some wrap library and system calls to check their parameters. Some also provide some kind of garbage collector to detect memory leaks at program termination. An extensive but not exhaustive list is maintained by Benjamin Zorn [32]. These tools typically do not intercept every load and store and so are relatively limited in the errors they can detect.

Similar tools find potential data races in threaded programs. Eraser [25] uses a *lockset refinement* algorithm that looks for shared memory that is not consistently protected by one or more locks when accessed. Improved algorithms, used by Helgrind, have been proposed and implemented [15].

RTC (runtime type checker) [19] performs run-time type checking of C and C++ programs, instrumenting via source annotation, and finds a similar set of run-time storage errors to Purify. Hobbes [4] is similar, but uses binary interpretation and thus can work on any program.

### 6.3 Program Instrumenters

Several tools, such as ATOM [28], statically instrument program binaries; others instrument parse trees. However, we will concentrate on instrumenters that use dynamic binary translation, which are more similar to Valgrind.

Shade [6] was an early dynamic translator. It supported insertion of basic trace instrumentation, and could run programs written for some architectures on some others, e.g. SPARC V8 programs on MIPS or SPARC V9. DynamoRIO [2], provides an API for adding instrumentation and editing programs not dissimilar to Valgrind's. The main difference is that the instrumentation occurs directly on x86 code, which restricts some of the tasks that can be done, since register allocation, condition codes, etc., must be respected. It is aimed primarily at code optimisation, but has been used to write a tool that protects against some security attacks, by checking that all jumps in a program look safe. DELI [7] is related to DynamoRIO (they are both descendants of Dynamo [1]). As well as code translation, caching and linking services, it provides hardware virtualization, and integrates emulated and native code execution. DynInst [3] allows a separate "mutator" process to insert and remove snippets of code from a running program [17]. It forms the core of the Paradyn parallel profiling tool [21]. Strata [26,27] runs on SPARC, MIPS, and x86. It has been used to implement a guard against stack-smashing attacks, and a system call monitor. Sind [23] is an instrumenter for SPARC/Solaris. The paper describing it suggests its use for security tools preventing stack-smash and buffer overflow attacks, although it is unclear whether these have been implemented. Walkabout [5] is a framework for experimenting with dynamic binary translation, designed from the ground up to be highly retargetable and machine-independent. It supports basic instrumentation such as basic block counting. DIOTA [20] also supports simple instrumentation, at the level of instructions and functions. It uses an unusual execution technique to support self-modifying code and programs that heavily mix code and data.

### 6.4 Tools Built With Valgrind

Calltree, by Josef Weidendorfer, extends Cachegrind to collect call tree information, and comes with a graphical viewer called KCachegrind [30]. VGprof [11], by Jeremy Fitzhardinge, is a gprof-style profiler with some extra features: exact (rather than sample-based) profiling information, histograms over multiple address spaces, and client requests to dump profiling information and zero counters, useful for timing specific sections of code.

Timothy Harris used Valgrind for a prototype implementation of a "virtualized debugger" [14] designed for debugging threaded and distributed programs that are difficult to debug using traditional techniques. His debugger sat "beneath" the debugged process, rather than alongside it, giving greater control over aspects such as scheduling.

Redux [22] creates *dynamic dataflow graphs* of the entire history of a program's computation; the sub-trace for each value shows how it was computed.

# 7 Future Work and Conclusion

This paper has described how program supervision tools can be created by plugging skins into Valgrind's core, and as an example described Memcheck, a powerful Purify-like C and C++ memory checker. It also presented some performance results for Valgrind used with four different skins.

## *7.1 Improving Valgrind*

Thinking long-term, there are three main challenges in making a supervision tool like Valgrind truly generic.

(i) Valgrind should support architectures other than x86. Although UCode is mostly platform-independent, it has several x86-specific characteristics (particularly the treatment of registers, condition codes, and floating point instructions) that make it unsuitable in its current form for other architectures. We do not plan to support cross-architecture translation; we think the advantages gained would not be worth the significant extra engineering effort that would be required.

(ii) Valgrind should operate in multiple environments (operating systems and libraries). Currently, the most fragile, intrusive and generally unsatisfactory parts of Valgrind are environment-specific, dealing with signals, system calls and threads. Ideally, these would be cleanly separated from the core via a well-specified interface, in order to port Valgrind to other operating systems, such as BSD, or even Windows. This is the greatest of the three challenges, we have only vague ideas on how to do this best.

(iii) The base execution mechanism should be cleanly separated from the code for the individual tools. Valgrind's core/skin split has largely achieved this.

Finally, we wish Valgrind did not rely on the underhand `LD_PRELOAD` technique to gain control of programs, if only so it could run (and check) itself.

## *7.2 Future Tools*

Many tools could be built using Valgrind, the most obvious being profilers. For example, a branch prediction simulator/profiler similar to Cachegrind could be useful. Although we do not wish to criticise profilers—they are very useful tools—we hope that Valgrind can be used for creating more innovative and powerful tools for understanding programs, and improving their correctness, in "deep" ways that programmers cannot achieve without automated assistance.

As one example, we believe there is great potential for tools that track additional state for each value used by the program. Existing examples include Memcheck's A and V bits, RTC's and Hobbes' dynamic types, Helgrind and Eraser's states [25], the invariants tracked by Daikon [9] and DIDUCE [13], and Redux's computation histories. Each of these tools fit our notion of being "deep", and those not already implemented as Valgrind skins could be easily. Such tools are not *code-centric* like profilers, which consider program fragments such as instructions and basic blocks. Instead they are *instance-centric*, where the instances in this case are words of memory. Other interesting instances might include memory blocks, stack frames, threads, mutexes, sockets, and so on.

We also hope that the judicious use of client requests will lead to novel tools, where static analysis information embedded in programs by compilers enable combined static/dynamic analysis. At the very least, this could reduce the amount of instrumentation for some skins (as in [31]).

### *7.3 Conclusion*

Valgrind, particularly the Memcheck skin, is in wide use. KDE 3.0 was extensively tested with Valgrind prior to its release, and we have received bug reports from hundreds of Valgrind users. As for the future, we hope Valgrind will be used in creating new tools that will benefit many people. Valgrind is available at `http://developer.kde.org/~sewardj`.

## Acknowledgement

Many thanks to: Donna Robinson and Rob Noble for encouragement; Reuben Thomas, Dirk Mueller, Stephan Kulow, Michael Matz, Simon Hausmann, David Faure, Ellis Whitehead and Frédéric Gobry for help and feedback; Josef Weidendorfer and Jeremy Fitzhardinge for writing new skins; Alan Mycroft and Jeremy Singer for helpful comments about this paper; and all those who have contributed patches, bug reports and suggestions for Valgrind. The first-listed author gratefully acknowledges the financial support of Trinity College, Cambridge.

## 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, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of CGO'03*, pages 265–276, San Francisco, California, USA, Mar. 2003.

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

[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, Apr. 2003.

[5] C. Cifuentes, B. T. Lewis, and D. Ung. Walkabout – A retargetable dynamic binary translation framework. Technical Report TR-2002-106, Sun Microsystems Laboratories, Palo Alto, California, USA, Jan. 2002.

[6] R. F. Cmelik and D. 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.

[7] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. Deli: A new run-time control point. In *Proceedings of MICRO35*, Istanbul, Turkey, Nov. 2002.

[8] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of OSDI 2000*, San Diego, California, USA, Oct. 2000.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb. 2001.

[10] D. Evans. Annotation-assisted lightweight static checking. In *Proceedings of ICSE 2000*, pages 40–42, Limerick, Ireland, Feb. 2000.

[11] J. Fitzhardinge. VGprof. `http://www.goop.org/~jeremy/valgrind/vgprof.html`.

[12] S. Hallem, B. Chen, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of PLDI 2002*, Berlin, Germany, June 2002.

[13] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of ISCE 2002*, pages 291–301, Orlando, Florida, USA, May 2002.

[14] T. L. Harris. Dependable software needs pervasive debugging. In *Proceedings of the ACM SIGOPS European Workshop*, Sintra, Portugal, Sept. 2002.

[15] J. J. Harrow, Jr. Runtime checking of multithreaded applications with Visual Threads. In *Proceedings of SPIN 2000*, volume 1885 of *LNCS*, pages 331–342, Stanford, California, USA, Aug. 2000.

[16] 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, Jan. 1992.

[17] J. K. Hollingsworth, B. P. Miller, and J. 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.

[18] S. C. Johnson. Lint, a C program checker. Computer Science Technical Report CSTR-65, updated version TM 78-1273-3, AT&T Bell Laboratories, Murray Hill, New Jersey, USA, Dec. 1977.

[19] A. Loginov, S. H. Yong, S. Horwitz, and T. Reps. Debugging via run-time type checking. In *Proceedings of FASE 2001*, Genoa, Italy, Apr. 2001.

[20] J. Maebe, M. Ronsse, and K. D. Bosschere. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Compendium of Workshops and Tutorials held in conjunction with PACT'02*, Charlottesville, Virginia, USA, Sept. 2002.

[21] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 28(11):37–46, Nov. 1995.

[22] N. Nethercote and A. Mycroft. Redux: A dynamic dataflow tracer. In *Proceedings of RV'03*, Boulder, Colorado, USA, July 2003. To appear.

[23] T. Palmer, D. D. Zovi, and D. Stefanovic. Sind: A framework for binary translation. Technical Report TR-CS-2001-38, Department of Computer Science, University of New Mexico, Alberquerque, New Mexico, USA, Dec. 2001.

[24] M. Probst. Dynamic binary translation. In *Proceedings of the UKUUG Linux Developers' Conference*, Bristol, United Kingdom, July 2002.

[25] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.

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

[27] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of CGO 2003*, pages 36–47, San Francisco, California, USA, Mar. 2003.

[28] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of PLDI '94*, pages 196–205, Orlando, Florida, USA, June 1994.

[29] 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.

[30] J. Weidendorfer. KCachegrind. `http://kcachegrind.sourceforge.net/`.

[31] S. H. Yong and S. Horwitz. Reducing the overhead of dynamic analysis. In *Proceedings of the 2nd International Workshop on Run-time Verification (RV'02)*, Copenhagen, Denmark, July 2002.

[32] B. Zorn. Debugging tools for dynamic storage allocation and memory management. `http://www.cs.colorado.edu/~zorn/MallocDebug.html`.