# 1.&#65533;The Design and Implementation of Valgrind

***Detailed technical notes for hackers, maintainers and the overly-curious***

**Table of Contents**



## 1.1.&#65533;Introduction

This document contains a detailed, highly-technical description of the internals of Valgrind. This is not the user manual; if you are an end-user of Valgrind, you do not want to read this. Conversely, if you really are a hacker-type and want to know how it works, I assume that you have read the user manual thoroughly.

You may need to read this document several times, and carefully. Some important things, I only say once.

[Note: this document is now very old, and a lot of its contents are out of date, and misleading.]

### 1.1.1.&#65533;History

Valgrind came into public view in late Feb 2002. However, it has been under contemplation for a very long time, perhaps seriously for about five years. Somewhat over two years ago, I started working on the x86 code generator for the Glasgow Haskell Compiler (http://www.haskell.org/ghc), gaining familiarity with x86 internals on the way. I then did Cacheprof, gaining further x86 experience. Some time around Feb 2000 I started experimenting with a user-space x86 interpreter for x86-Linux. This worked, but it was clear that a JIT-based scheme would be necessary to give reasonable performance for Valgrind. Design work for the JITter started in earnest in Oct 2000, and by early 2001 I had an x86-to-x86 dynamic translator which could run quite large programs. This translator was in a sense pointless, since it did not do any instrumentation or checking.

Most of the rest of 2001 was taken up designing and implementing the instrumentation scheme. The main difficulty, which consumed a lot of effort, was to design a scheme which did not generate large numbers of false uninitialised-value warnings. By late 2001 a satisfactory scheme had been arrived at, and I started to test it on ever-larger programs, with an eventual eye to making it work well enough so that it was helpful to folks debugging the upcoming version 3 of KDE. I've used KDE since before version 1.0, and wanted to Valgrind to be an indirect contribution to the KDE 3 development effort. At the start of Feb 02 the kde-core-devel crew started using it, and gave a huge amount of helpful feedback and patches in the space of three weeks. Snapshot 20020306 is the result.

In the best Unix tradition, or perhaps in the spirit of Fred Brooks' depressing-but-completely-accurate epitaph "build one to throw away; you will anyway", much of Valgrind is a second or third rendition of the initial idea. The instrumentation machinery (`vg_translate.c`, `vg_memory.c`) and

core CPU simulation (vg_to_ucode.c, vg_from_ucode.c) have had three redesigns and rewrites; the register allocator, low-level memory manager (vg_malloc2.c) and symbol table reader (vg_symtab2.c) are on the second rewrite. In a sense, this document serves to record some of the knowledge gained as a result.

### 1.1.2.Design overview

Valgrind is compiled into a Linux shared object, valgrind.so, and also a dummy one, valgrinq.so, of which more later. The valgrind shell script adds valgrind.so to the LD_PRELOAD list of extra libraries to be loaded with any dynamically linked library. This is a standard trick, one which I assume the LD_PRELOAD mechanism was developed to support.

valgrind.so is linked with the -z initfirst flag, which requests that its initialisation code is run before that of any other object in the executable image. When this happens, valgrind gains control. The real CPU becomes "trapped" in valgrind.so and the translations it generates. The synthetic CPU provided by Valgrind does, however, return from this initialisation function. So the normal startup actions, orchestrated by the dynamic linker ld.so, continue as usual, except on the synthetic CPU, not the real one. Eventually main is run and returns, and then the finalisation code of the shared objects is run, presumably in inverse order to which they were initialised. Remember, this is still all happening on the simulated CPU. Eventually valgrind.so's own finalisation code is called. It spots this event, shuts down the simulated CPU, prints any error summaries and/or does leak detection, and returns from the initialisation code on the real CPU. At this point, in effect the real and synthetic CPUs have merged back into one, Valgrind has lost control of the program, and the program finally exit()s back to the kernel in the usual way.

The normal course of activity, once Valgrind has started up, is as follows. Valgrind never runs any part of your program (usually referred to as the "client"), not a single byte of it, directly. Instead it uses function VG_(translate) to translate basic blocks (BBs, straight-line sequences of code) into instrumented translations, and those are run instead. The translations are stored in the translation cache (TC), vg_tc, with the translation table (TT), vg_tt supplying the original-to-translation code address mapping. Auxiliary array VG_(tt_fast) is used as a direct-map cache for fast lookups in TT; it usually achieves a hit rate of around 98% and facilitates an orig-to-trans lookup in 4 x86 insns, which is not bad.

Function VG_(dispatch) in vg_dispatch.S is the heart of the JIT dispatcher. Once a translated code address has been found, it is executed simply by an x86 call to the translation. At the end of the translation, the next original code addr is loaded into %eax, and the translation then does a ret, taking it back to the dispatch loop, with, interestingly, zero branch mispredictions. The address requested in %eax is looked up first in VG_(tt_fast), and, if not found, by calling C helper VG_(search_transtab). If there is still no translation available, VG_(dispatch) exits back to the top-level C dispatcher VG_(toploop), which arranges for VG_(translate) to make a new translation. All fairly unsurprising, really. There are various complexities described below.

The translator, orchestrated by VG_(translate), is complicated but entirely self-contained. It is described in great detail in subsequent sections. Translations are stored in TC, with TT tracking administrative information. The translations are subject to an approximate LRU-based management scheme. With the current settings, the TC can hold at most about 15MB of translations, and LRU passes prune it to about 13.5MB. Given that the orig-to-translation expansion ratio is about 13:1 to 14:1, this means TC holds translations for more or less a megabyte of original code, which generally comes to about 70000 basic blocks for C++ compiled with optimisation on. Generating new translations is expensive, so it is worth having a large TC to minimise the (capacity) miss rate.

The dispatcher, VG_(dispatch), receives hints from the translations which allow it to cheaply spot all control transfers corresponding to x86 call and ret instructions. It has to do this in order to spot some special events:

- Calls to VG_(shutdown). This is Valgrind's cue to exit. NOTE: actually this is done a different way; it should be cleaned up.
- Returns of system call handlers, to the return address VG_(signalreturn_bogusRA). The signal simulator needs to know when a signal handler is returning, so we spot jumps (returns) to this address.
- Calls to vg_trap_here. All malloc, free, etc calls that the client program makes are eventually routed to a call to vg_trap_here, and Valgrind does its own special thing with these calls. In effect this provides a trapdoor, by which Valgrind can intercept certain calls on the simulated CPU, run the call as it sees fit itself (on the real CPU), and return the result to the simulated CPU, quite transparently to the client program.

Valgrind intercepts the client's malloc, free, etc, calls, so that it can store additional information. Each block malloc'd by the client gives rise to a shadow block in which Valgrind stores the call stack at the time of the malloc call. When the client calls free, Valgrind tries to find the shadow block corresponding to the address passed to free, and emits an error message if none can be found. If it is found, the block is placed on the freed blocks queue vg_freed_list, it is marked as inaccessible, and its shadow block now records the call stack at the time of the free call. Keeping free'd blocks in this queue allows Valgrind to spot all (presumably invalid) accesses to them. However, once the volume of blocks in the free queue exceeds VG_(clo_freelist_vol), blocks are finally removed from the queue.

Keeping track of A and V bits (note: if you don't know what these are, you haven't read the user guide carefully enough) for memory is done in vg_memory.c. This implements a sparse array structure which covers the entire 4G address space in a way which is reasonably fast and reasonably space efficient. The 4G address space is divided up into 64K sections, each covering 64Kb of address space. Given a 32-bit address, the top 16 bits are used to select one of the 65536 entries in VG_(primary_map). The resulting "secondary" (SecMap) holds A and V bits for the 64k of address space chunk corresponding to the lower 16 bits of the address.

### 1.1.3.Design decisions

Some design decisions were motivated by the need to make Valgrind debuggable. Imagine you are writing a CPU simulator. It works fairly well. However, you run some large program, like Netscape, and after tens of millions of instructions, it crashes. How can you figure out where in your simulator the bug is?

Valgrind's answer is: cheat. Valgrind is designed so that it is possible to switch back to running the client program on the real CPU at any point. Using the --stop-after= flag, you can ask Valgrind to run just some number of basic blocks, and then run the rest of the way on the real CPU.

If you are searching for a bug in the simulated CPU, you can use this to do a binary search, which quickly leads you to the specific basic block which is causing the problem.

This is all very handy. It does constrain the design in certain unimportant ways. Firstly, the layout of memory, when viewed from the client's point of view, must be identical regardless of whether it is running on the real or simulated CPU. This means that Valgrind can't do pointer swizzling -- well, no great loss -- and it can't run on the same stack as the client -- again, no great loss. Valgrind operates on its own stack, `VG_(stack)`, which it switches to at startup, temporarily switching back to the client's stack when doing system calls for the client.

Valgrind also receives signals on its own stack, `VG_(sigstack)`, but for different gruesome reasons discussed below.

This nice clean switch-back-to-the-real-CPU-whenever-you-like story is muddied by signals. Problem is that signals arrive at arbitrary times and tend to slightly perturb the basic block count, with the result that you can get close to the basic block causing a problem but can't home in on it exactly. My kludgey hack is to define `SIGNAL_SIMULATION` to 1 towards the bottom of `vg_syscall_mem.c`, so that signal handlers are run on the real CPU and don't change the BB counts.

A second hole in the switch-back-to-real-CPU story is that Valgrind's way of delivering signals to the client is different from that of the kernel. Specifically, the layout of the signal delivery frame, and the mechanism used to detect a sighandler returning, are different. So you can't expect to make the transition inside a sighandler and still have things working, but in practice that's not much of a restriction.

Valgrind's implementation of `malloc`, `free`, etc, (in `vg_clientmalloc.c`, not the low-level stuff in `vg_malloc2.c`) is somewhat complicated by the need to handle switching back at arbitrary points. It does work tho.

### 1.1.4.Correctness

There's only one of me, and I have a Real Life (tm) as well as hacking Valgrind [allegedly :-]. That means I don't have time to waste chasing endless bugs in Valgrind. My emphasis is therefore on doing everything as simply as possible, with correctness, stability and robustness being the number one priority, more important than performance or functionality. As a result:

- The code is absolutely loaded with assertions, and these are **permanently enabled.** I have no plan to remove or disable them later. Over the past couple of months, as valgrind has become more widely used, they have shown their worth, pulling up various bugs which would otherwise have appeared as hard-to-find segmentation faults.

  I am of the view that it's acceptable to spend 5% of the total running time of your valgrindified program doing assertion checks and other internal sanity checks.

- Aside from the assertions, valgrind contains various sets of internal sanity checks, which get run at varying frequencies during normal operation. `VG_(do_sanity_checks)` runs every 1000 basic blocks, which means 500 to 2000 times/second for typical machines at present. It checks that Valgrind hasn't overrun its private stack, and does some simple checks on the memory permissions maps. Once every 25 calls it does some more extensive checks on those maps. Etc, etc.

  The following components also have sanity check code, which can be enabled to aid debugging:

  The low-level memory-manager (`VG_(mallocSanityCheckArena)`). This does a complete check of all blocks and chains in an arena, which is very slow. Is not engaged by default.

  The symbol table reader(s): various checks to ensure uniqueness of mappings; see `VG_(read_symbols)` for a start. Is permanently engaged.

  The A and V bit tracking stuff in `vg_memory.c`. This can be compiled with cpp symbol `VG_DEBUG_MEMORY` defined, which removes all the fast, optimised cases, and uses simple-but-slow fallbacks instead. Not engaged by default.

  Ditto `VG_DEBUG_LEAKCHECK`.

  The JITter parses x86 basic blocks into sequences of UCode instructions. It then sanity checks each one with `VG_(saneUInstr)` and sanity checks the sequence as a whole with `VG_(saneUCodeBlock)`. This stuff is engaged by default, and has caught some way-obscure bugs in the simulated CPU machinery in its time.

  The system call wrapper does `VG_(first_and_last_secondaries_look_plausible)` after every syscall; this is known to pick up bugs in the syscall wrappers. Engaged by default.

  The main dispatch loop, in `VG_(dispatch)`, checks that translations do not set `%ebp` to any value different from `VG_EBP_DISPATCH_CHECKED` or `& VG_(baseBlock)`. In effect this test is free, and is permanently engaged.

  There are a couple of ifdefed-out consistency checks I inserted whilst debugging the new register allocater, `vg_do_register_allocation`.

- I try to avoid techniques, algorithms, mechanisms, etc, for which I can supply neither a convincing argument that they are correct, nor sanity-check code which might pick up bugs in my implementation. I don't always succeed in this, but I try. Basically the idea is: avoid techniques which are, in practice, unverifiable, in some sense. When doing anything, always have in mind: "how can I verify that this is correct?"

Some more specific things are:

- Valgrind runs in the same namespace as the client, at least from `ld.so`'s point of view, and it therefore absolutely had better not export any symbol with a name which could clash with that of the client or any of its libraries. Therefore, all globally visible symbols exported from `valgrind.so` are defined using the `VG_` CPP macro. As you'll see from `vg_constants.h`, this appends some arbitrary prefix to the symbol, in order that it be, we hope, globally unique. Currently the prefix is `vgPlain_`. For convenience there are also `VGM_`, `VGP_` and `VGOFF_`. All locally defined symbols are declared `static` and do not appear in the final shared object.

To check this, I periodically do `nm valgrind.so | grep " T "`, which shows you all the globally exported text symbols. They should all have an approved prefix, except for those like `malloc`, `free`, etc, which we deliberately want to shadow and take precedence over the same names exported from `glibc.so`, so that valgrind can intercept those calls easily. Similarly, `nm valgrind.so | grep " D "` allows you to find any rogue data-segment symbol names.

- Valgrind tries, and almost succeeds, in being completely independent of all other shared objects, in particular of `glibc.so`. For example, we have our own low-level memory manager in `vg_malloc2.c`, which is a fairly standard malloc/free scheme augmented with arenas, and `vg_mylibc.c` exports reimplementations of various bits and pieces you'd normally get from the C library.

  Why all the hassle? Because imagine the potential chaos of both the simulated and real CPUs executing in `glibc.so`. It just seems simpler and cleaner to be completely self-contained, so that only the simulated CPU visits `glibc.so`. In practice it's not much hassle anyway. Also, valgrind starts up before glibc has a chance to initialise itself, and who knows what difficulties that could lead to. Finally, glibc has definitions for some types, specifically `sigset_t`, which conflict (are different from) the Linux kernel's idea of same. When Valgrind wants to fiddle around with signal stuff, it wants to use the kernel's definitions, not glibc's definitions. So it's simplest just to keep glibc out of the picture entirely.

  To find out which glibc symbols are used by Valgrind, reinstate the link flags `-nostdlib -Wl,-no-undefined`. This causes linking to fail, but will tell you what you depend on. I have mostly, but not entirely, got rid of the glibc dependencies; what remains is, IMO, fairly harmless. AFAIK the current dependencies are: `memset`, `memcmp`, `stat`, `system`, `sbrk`, `setjmp` and `longjmp`.

- Similarly, valgrind should not really import any headers other than the Linux kernel headers, since it knows of no API other than the kernel interface to talk to. At the moment this is really not in a good state, and `vg_syscall_mem` imports, via `vg_unsafe.h`, a significant number of C-library headers so as to know the sizes of various structs passed across the kernel boundary. This is of course completely bogus, since there is no guarantee that the C library's definitions of these structs matches those of the kernel. I have started to sort this out using `vg_kerneliface.h`, into which I had intended to copy all kernel definitions which valgrind could need, but this has not gotten very far. At the moment it mostly contains definitions for `sigset_t` and `struct sigaction`, since the kernel's definition for these really does clash with glibc's. I plan to use a `vki_` prefix on all these types and constants, to denote the fact that they pertain to **V**algrind's **K**ernel **I**nterface.

  Another advantage of having a `vg_kerneliface.h` file is that it makes it simpler to interface to a different kernel. Once can, for example, easily imagine writing a new `vg_kerneliface.h` for FreeBSD, or x86 NetBSD.

### 1.1.5.Current limitations

Support for weird (non-POSIX) signal stuff is patchy. Does anybody care?

## 1.2.The instrumenting JITter

This really is the heart of the matter. We begin with various side issues.

### 1.2.1.Run-time storage, and the use of host registers

Valgrind translates client (original) basic blocks into instrumented basic blocks, which live in the translation cache TC, until either the client finishes or the translations are ejected from TC to make room for newer ones.

Since it generates x86 code in memory, Valgrind has complete control of the use of registers in the translations. Now pay attention. I shall say this only once, and it is important you understand this. In what follows I will refer to registers in the host (real) cpu using their standard names, `%eax`, `%edi`, etc. I refer to registers in the simulated CPU by capitalising them: `%EAX`, `%EDI`, etc. These two sets of registers usually bear no direct relationship to each other; there is no fixed mapping between them. This naming scheme is used fairly consistently in the comments in the sources.

Host registers, once things are up and running, are used as follows:

- `%esp`, the real stack pointer, points somewhere in Valgrind's private stack area, `VG_(stack)` or, transiently, into its signal delivery stack, `VG_(sigstack)`.

- `%edi` is used as a temporary in code generation; it is almost always dead, except when used for the `Left` value-tag operations.

- `%eax`, `%ebx`, `%ecx`, `%edx` and `%esi` are available to Valgrind's register allocator. They are dead (carry unimportant values) in between translations, and are live only in translations. The one exception to this is `%eax`, which, as mentioned far above, has a special significance to the dispatch loop `VG_(dispatch)`: when a translation returns to the dispatch loop, `%eax` is expected to contain the original-code-address of the next translation to run. The register allocator is so good at minimising spill code that using five regs and not having to save/restore `%edi` actually gives better code than allocating to `%edi` as well, but then having to push/pop it around special uses.

- `%ebp` points permanently at `VG_(baseBlock)`. Valgrind's translations are position-independent, partly because this is convenient, but also because translations get moved around in TC as part of the LRUing activity. **All** static entities which need to be referred to from generated code, whether data or helper functions, are stored starting at `VG_(baseBlock)` and are therefore reached by indexing from `%ebp`. There is but one exception, which is that by placing the value `VG_EBP_DISPATCH_CHECKED` in `%ebp` just before a return to the dispatcher, the dispatcher is informed that the next address to run, in `%eax`, requires special treatment.

- The real machine's FPU state is pretty much unimportant, for reasons which will become obvious. Ditto its `%eflags` register.

The state of the simulated CPU is stored in memory, in `VG_(baseBlock)`, which is a block of 200 words IIRC. Recall that `%ebp` points permanently at the start of this block. Function `vg_init_baseBlock` decides what the offsets of various entities in `VG_(baseBlock)` are to be, and allocates word offsets for them. The code generator then emits `%ebp` relative addresses to get at those things. The sequence in which

entities are allocated has been carefully chosen so that the 32 most popular entities come first, because this means 8-bit offsets can be used in the generated code.

If I was clever, I could make `%ebp` point 32 words along `VG_(baseBlock)`, so that I'd have another 32 words of short-form offsets available, but that's just complicated, and it's not important -- the first 32 words take 99% (or whatever) of the traffic.

Currently, the sequence of stuff in `VG_(baseBlock)` is as follows:

- 9 words, holding the simulated integer registers, `%EAX` .. `%EDI`, and the simulated flags, `%EFLAGS`.
- Another 9 words, holding the V bit "shadows" for the above 9 regs.
- The **addresses** of various helper routines called from generated code: `VG_(helper_value_check4_fail)`, `VG_(helper_value_check0_fail)`, which register V-check failures, `VG_(helperc_STOREV4)`, `VG_(helperc_STOREV1)`, `VG_(helperc_LOADV4)`, `VG_(helperc_LOADV1)`, which do stores and loads of V bits to/from the sparse array which keeps track of V bits in memory, and `VGM_(handle_esp_assignment)`, which messes with memory addressibility resulting from changes in `%ESP`.
- The simulated `%EIP`.
- 24 spill words, for when the register allocator can't make it work with 5 measly registers.
- Addresses of helpers `VG_(helperc_STOREV2)`, `VG_(helperc_LOADV2)`. These are here because 2-byte loads and stores are relatively rare, so are placed above the magic 32-word offset boundary.
- For similar reasons, addresses of helper functions `VGM_(fpu_write_check)` and `VGM_(fpu_read_check)`, which handle the A/V maps testing and changes required by FPU writes/reads.
- Some other boring helper addresses: `VG_(helper_value_check2_fail)` and `VG_(helper_value_check1_fail)`. These are probably never emitted now, and should be removed.
- The entire state of the simulated FPU, which I believe to be 108 bytes long.
- Finally, the addresses of various other helper functions in `vg_helpers.S`, which deal with rare situations which are tedious or difficult to generate code in-line for.

As a general rule, the simulated machine's state lives permanently in memory at `VG_(baseBlock)`. However, the JITter does some optimisations which allow the simulated integer registers to be cached in real registers over multiple simulated instructions within the same basic block. These are always flushed back into memory at the end of every basic block, so that the in-memory state is up-to-date between basic blocks. (This flushing is implied by the statement above that the real machine's allocatable registers are dead in between simulated blocks).

### 1.2.2.&#65533;Startup, shutdown, and system calls

Getting into of Valgrind (`VG_(startup)`, called from `valgrind.so`'s initialisation section), really means copying the real CPU's state into `VG_(baseBlock)`, and then installing our own stack pointer, etc, into the real CPU, and then starting up the JITter. Exiting valgrind involves copying the simulated state back to the real state.

Unfortunately, there's a complication at startup time. Problem is that at the point where we need to take a snapshot of the real CPU's state, the offsets in `VG_(baseBlock)` are not set up yet, because to do so would involve disrupting the real machine's state significantly. The way round this is to dump the real machine's state into a temporary, static block of memory, `VG_(m_state_static)`. We can then set up the `VG_(baseBlock)` offsets at our leisure, and copy into it from `VG_(m_state_static)` at some convenient later time. This copying is done by `VG_(copy_m_state_static_to_baseBlock)`.

On exit, the inverse transformation is (rather unnecessarily) used: stuff in `VG_(baseBlock)` is copied to `VG_(m_state_static)`, and the assembly stub then copies from `VG_(m_state_static)` into the real machine registers.

Doing system calls on behalf of the client (`vg_syscall.S`) is something of a half-way house. We have to make the world look sufficiently like that which the client would normally have to make the syscall actually work properly, but we can't afford to lose control. So the trick is to copy all of the client's state, **except its program counter**, into the real CPU, do the system call, and copy the state back out. Note that the client's state includes its stack pointer register, so one effect of this partial restoration is to cause the system call to be run on the client's stack, as it should be.

As ever there are complications. We have to save some of our own state somewhere when restoring the client's state into the CPU, so that we can keep going sensibly afterwards. In fact the only thing which is important is our own stack pointer, but for paranoia reasons I save and restore our own FPU state as well, even though that's probably pointless.

The complication on the above complication is, that for horrible reasons to do with signals, we may have to handle a second client system call whilst the client is blocked inside some other system call (unbelievable!). That means there's two sets of places to dump Valgrind's stack pointer and FPU state across the syscall, and we decide which to use by consulting `VG_(syscall_depth)`, which is in turn maintained by `VG_(wrap_syscall)`.

### 1.2.3.&#65533;Introduction to UCode

UCode lies at the heart of the x86-to-x86 JITter. The basic premise is that dealing with the x86 instruction set head-on is just too darn complicated, so we do the traditional compiler-writer's trick and translate it into a simpler, easier-to-deal-with form.

In normal operation, translation proceeds through six stages, coordinated by `VG_(translate)`:

1. Parsing of an x86 basic block into a sequence of UCode instructions (`VG_(disBB)`).
2. UCode optimisation (`vg_improve`), with the aim of caching simulated registers in real registers over multiple simulated instructions, and removing redundant simulated `%EFLAGS` saving/restoring.
3. UCode instrumentation (`vg_instrument`), which adds value and address checking code.
4. Post-instrumentation cleanup (`vg_cleanup`), removing redundant value-check computations.
5. Register allocation (`vg_do_register_allocation`), which, note, is done on UCode.
6. Emission of final instrumented x86 code (`VG_(emit_code)`).

Notice how steps 2, 3, 4 and 5 are simple UCode-to-UCode transformation passes, all on straight-line blocks of UCode (type `UCodeBlock`). Steps 2 and 4 are optimisation passes and can be disabled for debugging purposes, with `--optimise=no` and `--cleanup=no` respectively.

Valgrind can also run in a no-instrumentation mode, given `--instrument=no`. This is useful for debugging the JITter quickly without having to deal with the complexity of the instrumentation mechanism too. In this mode, steps 3 and 4 are omitted.

These flags combine, so that `--instrument=no` together with `--optimise=no` means only steps 1, 5 and 6 are used. `--single-step=yes` causes each x86 instruction to be treated as a single basic block. The translations are terrible but this is sometimes instructive.

The `--stop-after=N` flag switches back to the real CPU after `N` basic blocks. It also re-JITs the final basic block executed and prints the debugging info resulting, so this gives you a way to get a quick snapshot of how a basic block looks as it passes through the six stages mentioned above. If you want to see full information for every block translated (probably not, but still ...) find, in `VG_(translate)`, the lines

```
dis = True;
dis = debugging_translation;
```

and comment out the second line. This will spew out debugging junk faster than you can possibly imagine.

### 1.2.4.UCode operand tags: type `Tag`

UCode is, more or less, a simple two-address RISC-like code. In keeping with the x86 AT&T assembly syntax, generally speaking the first operand is the source operand, and the second is the destination operand, which is modified when the uinstr is notionally executed.

UCode instructions have up to three operand fields, each of which has a corresponding `Tag` describing it. Possible values for the tag are:

- `NoValue`: indicates that the field is not in use.
- `Lit16`: the field contains a 16-bit literal.
- `Literal`: the field denotes a 32-bit literal, whose value is stored in the `lit32` field of the uinstr itself. Since there is only one `lit32` for the whole uinstr, only one operand field may contain this tag.
- `SpillNo`: the field contains a spill slot number, in the range 0 to 23 inclusive, denoting one of the spill slots contained inside `VG_(baseBlock)`. Such tags only exist after register allocation.
- `RealReg`: the field contains a number in the range 0 to 7 denoting an integer x86 ("real") register on the host. The number is the Intel encoding for integer registers. Such tags only exist after register allocation.
- `ArchReg`: the field contains a number in the range 0 to 7 denoting an integer x86 register on the simulated CPU. In reality this means a reference to one of the first 8 words of `VG_(baseBlock)`. Such tags can exist at any point in the translation process.
- Last, but not least, `TempReg`. The field contains the number of one of an infinite set of virtual (integer) registers. `TempReg`s are used everywhere throughout the translation process; you can have as many as you want. The register allocator maps as many as it can into `RealReg`s and turns the rest into `SpillNo`s, so `TempReg`s should not exist after the register allocation phase.

  `TempReg`s are always 32 bits long, even if the data they hold is logically shorter. In that case the upper unused bits are required, and, I think, generally assumed, to be zero. `TempReg`s holding V bits for quantities shorter than 32 bits are expected to have ones in the unused places, since a one denotes "undefined".

### 1.2.5.UCode instructions: type `UInstr`

UCode was carefully designed to make it possible to do register allocation on UCode and then translate the result into x86 code without needing any extra registers ... well, that was the original plan, anyway. Things have gotten a little more complicated since then. In what follows, UCode instructions are referred to as uinstrs, to distinguish them from x86 instructions. Uinstrs of course have uopcodes which are (naturally) different from x86 opcodes.

A uinstr (type `UInstr`) contains various fields, not all of which are used by any one uopcode:

- Three 16-bit operand fields, `val1`, `val2` and `val3`.

- Three tag fields, `tag1`, `tag2` and `tag3`. Each of these has a value of type `Tag`, and they describe what the `val1`, `val2` and `val3` fields contain.
- A 32-bit literal field.
- Two `FlagSet`s, specifying which x86 condition codes are read and written by the uinstr.
- An opcode byte, containing a value of type `Opcode`.
- A size field, indicating the data transfer size (1/2/4/8/10) in cases where this makes sense, or zero otherwise.
- A condition-code field, which, for jumps, holds a value of type `Condcode`, indicating the condition which applies. The encoding is as it is in the x86 insn stream, except we add a 17th value `CondAlways` to indicate an unconditional transfer.
- Various 1-bit flags, indicating whether this insn pertains to an x86 CALL or RET instruction, whether a widening is signed or not, etc.

UOpcodes (type `Opcode`) are divided into two groups: those necessary merely to express the functionality of the x86 code, and extra uopcodes needed to express the instrumentation. The former group contains:

- `GET` and `PUT`, which move values from the simulated CPU's integer registers (`ArchReg`s) into `TempReg`s, and back. `GETF` and `PUTF` do the corresponding thing for the simulated `%EFLAGS`. There are no corresponding insns for the FPU register stack, since we don't explicitly simulate its registers.
- `LOAD` and `STORE`, which, in RISC-like fashion, are the only uinstrs able to interact with memory.
- `MOV` and `CMOV` allow unconditional and conditional moves of values between `TempReg`s.
- ALU operations. Again in RISC-like fashion, these only operate on `TempReg`s (before reg-alloc) or `RealReg`s (after reg-alloc). These are: `ADD`, `ADC`, `AND`, `OR`, `XOR`, `SUB`, `SBB`, `SHL`, `SHR`, `SAR`, `ROL`, `ROR`, `RCL`, `RCR`, `NOT`, `NEG`, `INC`, `DEC`, `BSWAP`, `CC2VAL` and `WIDEN`. `WIDEN` does signed or unsigned value widening. `CC2VAL` is used to convert condition codes into a value, zero or one. The rest are obvious.

  To allow for more efficient code generation, we bend slightly the restriction at the start of the previous para: for `ADD`, `ADC`, `XOR`, `SUB` and `SBB`, we allow the first (source) operand to also be an `ArchReg`, that is, one of the simulated machine's registers. Also, many of these ALU ops allow the source operand to be a literal. See `VG_(saneUInstr)` for the final word on the allowable forms of uinstrs.
- `LEA1` and `LEA2` are not strictly necessary, but allow faciliate better translations. They record the fancy x86 addressing modes in a direct way, which allows those amodes to be emitted back into the final instruction stream more or less verbatim.
- `CALLM` calls a machine-code helper, one of the methods whose address is stored at some `VG_(baseBlock)` offset. `PUSH` and `POP` move values to/from `TempReg` to the real (Valgrind's) stack, and `CLEAR` removes values from the stack. `CALLM_S` and `CALLM_E` delimit the boundaries of call setups and clearings, for the benefit of the instrumentation passes. Getting this right is critical, and so `VG_(saneUCodeBlock)` makes various checks on the use of these uopcodes.

  It is important to understand that these uopcodes have nothing to do with the x86 `call`, `return`, `push` or `pop` instructions, and are not used to implement them. Those guys turn into combinations of `GET`, `PUT`, `LOAD`, `STORE`, `ADD`, `SUB`, and `JMP`. What these uopcodes support is calling of helper functions such as `VG_(helper_imul_32_64)`, which do stuff which is too difficult or tedious to emit inline.
- `FPU`, `FPU_R` and `FPU_W`. Valgrind doesn't attempt to simulate the internal state of the FPU at all. Consequently it only needs to be able to distinguish FPU ops which read and write memory from those that don't, and for those which do, it needs to know the effective address and data transfer size. This is made easier because the x86 FP instruction encoding is very regular, basically consisting of 16 bits for a non-memory FPU insn and 11 (IIRC) bits + an address mode for a memory FPU insn. So our `FPU` uinstr carries the 16 bits in its `val1` field. And `FPU_R` and `FPU_W` carry 11 bits in that field, together with the identity of a `TempReg` or (later) `RealReg` which contains the address.
- `JIFZ` is unique, in that it allows a control-flow transfer which is not deemed to end a basic block. It causes a jump to a literal (original) address if the specified argument is zero.
- Finally, `INCEIP` advances the simulated `%EIP` by the specified literal amount. This supports lazy `%EIP` updating, as described below.

Stages 1 and 2 of the 6-stage translation process mentioned above deal purely with these uopcodes, and no others. They are sufficient to express pretty much all the x86 32-bit protected-mode instruction set, at least everything understood by a pre-MMX original Pentium (P54C).

Stages 3, 4, 5 and 6 also deal with the following extra "instrumentation" uopcodes. They are used to express all the definedness-tracking and -checking machinery which valgrind does. In later sections we show how to create checking code for each of the uopcodes above. Note that these instrumentation uopcodes, although some appearing complicated, have been carefully chosen so that efficient x86 code can be generated for them. GNU superopt v2.5 did a great job helping out here. Anyways, the uopcodes are as follows:

- `GETV` and `PUTV` are analogues to `GET` and `PUT` above. They are identical except that they move the V bits for the specified values back and forth to `TempRegs`, rather than moving the values themselves.
- Similarly, `LOADV` and `STOREV` read and write V bits from the synthesised shadow memory that Valgrind maintains. In fact they do more than that, since they also do address-validity checks, and emit complaints if the read/written addresses are unaddressible.
- `TESTV`, whose parameters are a `TempReg` and a size, tests the V bits in the `TempReg`, at the specified operation size (0/1/2/4 byte) and emits an error if any of them indicate undefinedness. This is the only uopcode capable of doing such tests.
- `SETV`, whose parameters are also `TempReg` and a size, makes the V bits in the `TempReg` indicated definedness, at the specified operation size. This is usually used to generate the correct V bits for a literal value, which is of course fully defined.

- GETVF and PUTVF are analogues to GETF and PUTF. They move the single V bit used to model definedness of %EFLAGS between its home in VG_(baseBlock) and the specified TempReg.

- TAG1 denotes one of a family of unary operations on TempRegs containing V bits. Similarly, TAG2 denotes one in a family of binary operations on V bits.

These 10 uopcodes are sufficient to express Valgrind's entire definedness-checking semantics. In fact most of the interesting magic is done by the TAG1 and TAG2 suboperations.

First, however, I need to explain about V-vector operation sizes. There are 4 sizes: 1, 2 and 4, which operate on groups of 8, 16 and 32 V bits at a time, supporting the usual 1, 2 and 4 byte x86 operations. However there is also the mysterious size 0, which really means a single V bit. Single V bits are used in various circumstances; in particular, the definedness of %EFLAGS is modelled with a single V bit. Now might be a good time to also point out that for V bits, 1 means "undefined" and 0 means "defined". Similarly, for A bits, 1 means "invalid address" and 0 means "valid address". This seems counterintuitive (and so it is), but testing against zero on x86s saves instructions compared to testing against all 1s, because many ALU operations set the Z flag for free, so to speak.

With that in mind, the tag ops are:

- **(UNARY) Pessimising casts:**VgT_PCast40, VgT_PCast20, VgT_PCast10, VgT_PCast01, VgT_PCast02 and VgT_PCast04. A "pessimising cast" takes a V-bit vector at one size, and creates a new one at another size, pessimised in the sense that if any of the bits in the source vector indicate undefinedness, then all the bits in the result indicate undefinedness. In this case the casts are all to or from a single V bit, so for example VgT_PCast40 is a pessimising cast from 32 bits to 1, whereas VgT_PCast04 simply copies the single source V bit into all 32 bit positions in the result. Surprisingly, these ops can all be implemented very efficiently.

  There are also the pessimising casts VgT_PCast14, from 8 bits to 32, VgT_PCast12, from 8 bits to 16, and VgT_PCast11, from 8 bits to 8. This last one seems nonsensical, but in fact it isn't a no-op because, as mentioned above, any undefined (1) bits in the source infect the entire result.

- **(UNARY) Propagating undefinedness upwards in a word:**VgT_Left4, VgT_Left2 and VgT_Left1. These are used to simulate the worst-case effects of carry propagation in adds and subtracts. They return a V vector identical to the original, except that if the original contained any undefined bits, then it and all bits above it are marked as undefined too. Hence the Left bit in the names.

- **(UNARY) Signed and unsigned value widening:**VgT_SWiden14, VgT_SWiden24, VgT_SWiden12, VgT_ZWiden14, VgT_ZWiden24 and VgT_ZWiden12. These mimic the definedness effects of standard signed and unsigned integer widening. Unsigned widening creates zero bits in the new positions, so VgT_ZWiden* accordingly park mark those parts of their argument as defined. Signed widening copies the sign bit into the new positions, so VgT_SWiden* copies the definedness of the sign bit into the new positions. Because 1 means undefined and 0 means defined, these operations can (fascinatingly) be done by the same operations which they mimic. Go figure.

- **(BINARY) Undefined-if-either-Undefined, Defined-if-either-Defined:**VgT_UifU4, VgT_UifU2, VgT_UifU1, VgT_UifU0, VgT_DifD4, VgT_DifD2, VgT_DifD1. These do simple bitwise operations on pairs of V-bit vectors, with UifU giving undefined if either arg bit is undefined, and DifD giving defined if either arg bit is defined. Abstract interpretation junkies, if any make it this far, may like to think of them as meets and joins (or is it joins and meets) in the definedness lattices.

- **(BINARY; one value, one V bits) Generate argument improvement terms for AND and OR.**VgT_ImproveAND4_TQ, VgT_ImproveAND2_TQ, VgT_ImproveAND1_TQ, VgT_ImproveOR4_TQ, VgT_ImproveOR2_TQ, VgT_ImproveOR1_TQ. These help out with AND and OR operations. AND and OR have the inconvenient property that the definedness of the result depends on the actual values of the arguments as well as their definedness. At the bit level:

  ```
  1 AND undefined = undefined, but
  0 AND undefined = 0, and
  similarly
  0 OR undefined = undefined, but
  1 OR undefined = 1.
  ```

  It turns out that gcc (quite legitimately) generates code which relies on this fact, so we have to model it properly in order to avoid flooding users with spurious value errors. The ultimate definedness result of AND and OR is calculated using UifU on the definedness of the arguments, but we also DifD in some "improvement" terms which take into account the above phenomena.

  ImproveAND takes as its first argument the actual value of an argument to AND (the T) and the definedness of that argument (the Q), and returns a V-bit vector which is defined (0) for bits which have value 0 and are defined; this, when DifD into the final result causes those bits to be defined even if the corresponding bit in the other argument is undefined.

  The ImproveOR ops do the dual thing for OR arguments. Note that XOR does not have this property that one argument can make the other irrelevant, so there is no need for such complexity for XOR.

That's all the tag ops. If you stare at this long enough, and then run Valgrind and stare at the pre- and post-instrumented ucode, it should be fairly obvious how the instrumentation machinery hangs together.

One point, if you do this: in order to make it easy to differentiate TempRegs carrying values from TempRegs carrying V bit vectors, Valgrind prints the former as (for example) t28 and the latter as q28; the fact that they carry the same number serves to indicate their relationship. This is purely for the convenience of the human reader; the register allocator and code generator don't regard them as different.

### 1.2.6.Translation into UCode

VG_(disBB) allocates a new UCodeBlock and then uses disInstr to translate x86 instructions one at a time into UCode, dumping the result in the UCodeBlock. This goes on until a control-flow transfer instruction is encountered.

Despite the large size of vg_to_ucode.c, this translation is really very simple. Each x86 instruction is translated entirely independently of its neighbours, merrily allocating new TempRegs as it goes. The idea is to have a simple translator -- in reality, no more than a macro-expander -- and the -- resulting bad UCode translation is cleaned up by the UCode optimisation phase which follows. To give you an idea of some x86 instructions and their translations (this is a complete basic block, as Valgrind sees it):

```
0x40435A50:  incl %edx
     0: GETL      %EDX, t0
     1: INCL      t0  (-wOSZAP)
     2: PUTL      t0, %EDX

0x40435A51:  movsbl (%edx),%eax
     3: GETL      %EDX, t2
     4: LDB       (t2), t2
     5: WIDENL_Bs t2
     6: PUTL      t2, %EAX

0x40435A54:  testb $0x20, 1(%ecx,%eax,2)
     7: GETL      %EAX, t6
     8: GETL      %ECX, t8
     9: LEA2L     1(t8,t6,2), t4
    10: LDB       (t4), t10
    11: MOVB      $0x20, t12
    12: ANDB      t12, t10  (-wOSZACP)
    13: INCEIPo   $9

0x40435A59:  jnz-8 0x40435A50
    14: Jnzo      $0x40435A50  (-rOSZACP)
    15: JMPo      $0x40435A5B
```

Notice how the block always ends with an unconditional jump to the next block. This is a bit unnecessary, but makes many things simpler.

Most x86 instructions turn into sequences of GET, PUT, LEA1, LEA2, LOAD and STORE. Some complicated ones however rely on calling helper bits of code in vg_helpers.S. The ucode instructions PUSH, POP, CALL, CALLM_S and CALLM_E support this. The calling convention is somewhat ad-hoc and is not the C calling convention. The helper routines must save all integer registers, and the flags, that they use. Args are passed on the stack underneath the return address, as usual, and if result(s) are to be returned, it (they) are either placed in dummy arg slots created by the ucode PUSH sequence, or just overwrite the incoming args.

In order that the instrumentation mechanism can handle calls to these helpers, VG_(saneUCodeBlock) enforces the following restrictions on calls to helpers:

- Each CALL uinstr must be bracketed by a preceding CALLM_S marker (dummy uinstr) and a trailing CALLM_E marker. These markers are used by the instrumentation mechanism later to establish the boundaries of the PUSH, POP and CLEAR sequences for the call.
- PUSH, POP and CLEAR may only appear inside sections bracketed by CALLM_S and CALLM_E, and nowhere else.
- In any such bracketed section, no two PUSH insns may push the same TempReg. Dually, no two two POPs may pop the same TempReg.
- Finally, although this is not checked, args should be removed from the stack with CLEAR, rather than POPs into a TempReg which is not subsequently used. This is because the instrumentation mechanism assumes that all values POPped from the stack are actually used.

Some of the translations may appear to have redundant TempReg-to-TempReg moves. This helps the next phase, UCode optimisation, to generate better code.

### 1.2.7.UCode optimisation

UCode is then subjected to an improvement pass (vg_improve()), which blurs the boundaries between the translations of the original x86 instructions. It's pretty straightforward. Three transformations are done:

- Redundant GET elimination. Actually, more general than that -- eliminates redundant fetches of ArchRegs. In our running example, uinstr 3 GETs %EDX into t2 despite the fact that, by looking at the previous uinstr, it is already in t0. The GET is therefore removed, and t2 renamed to t0. Assuming t0 is allocated to a host register, it means the simulated %EDX will exist in a host CPU register for more than one simulated x86 instruction, which seems to me to be a highly desirable property.

  There is some mucking around to do with subregisters; %AL vs %AH %AX vs %EAX etc. I can't remember how it works, but in general we are very conservative, and these tend to invalidate the caching.

- Redundant PUT elimination. This annuls PUTs of values back to simulated CPU registers if a later PUT would overwrite the earlier PUT value, and there is no intervening reads of the simulated register (ArchReg).

  As before, we are paranoid when faced with subregister references. Also, PUTs of %ESP are never annulled, because it is vital the instrumenter always has an up-to-date %ESP value available, %ESP changes affect addressibility of the memory around the simulated stack pointer.

The implication of the above paragraph is that the simulated machine's registers are only lazily updated once the above two optimisation phases have run, with the exception of `%ESP`. `TempReg`s go dead at the end of every basic block, from which is is inferrable that any `TempReg` caching a simulated CPU reg is flushed (back into the relevant `VG_(baseBlock)` slot) at the end of every basic block. The further implication is that the simulated registers are only up-to-date at in between basic blocks, and not at arbitrary points inside basic blocks. And the consequence of that is that we can only deliver signals to the client in between basic blocks. None of this seems any problem in practice.

- Finally there is a simple def-use thing for condition codes. If an earlier uinstr writes the condition codes, and the next uinsn along which actually cares about the condition codes writes the same or larger set of them, but does not read any, the earlier uinsn is marked as not writing any condition codes. This saves a lot of redundant cond-code saving and restoring.

The effect of these transformations on our short block is rather unexciting, and shown below. On longer basic blocks they can dramatically improve code quality.

```
at 3: delete GET, rename t2 to t0 in (4 .. 6)
at 7: delete GET, rename t6 to t0 in (8 .. 9)
at 1: annul flag write OSZAP due to later OSZACP

Improved code:
     0: GETL      %EDX, t0
     1: INCL      t0
     2: PUTL      t0, %EDX
     4: LDB       (t0), t0
     5: WIDENL_Bs t0
     6: PUTL      t0, %EAX
     8: GETL      %ECX, t8
     9: LEA2L     1(t8,t0,2), t4
    10: LDB       (t4), t10
    11: MOVB      $0x20, t12
    12: ANDB      t12, t10  (-wOSZACP)
    13: INCEIPo   $9
    14: Jnzo      $0x40435A50  (-rOSZACP)
    15: JMPo      $0x40435A5B
```

### 1.2.8.UCode instrumentation

Once you understand the meaning of the instrumentation uinstrs, discussed in detail above, the instrumentation scheme is fairly straightforward. Each uinstr is instrumented in isolation, and the instrumentation uinstrs are placed before the original uinstr. Our running example continues below. I have placed a blank line after every original ucode, to make it easier to see which instrumentation uinstrs correspond to which originals.

As mentioned somewhere above, `TempReg`s carrying values have names like `t28`, and each one has a shadow carrying its V bits, with names like `q28`. This pairing aids in reading instrumented ucode.

One decision about all this is where to have "observation points", that is, where to check that V bits are valid. I use a minimalistic scheme, only checking where a failure of validity could cause the original program to (seg)fault. So the use of values as memory addresses causes a check, as do conditional jumps (these cause a check on the definedness of the condition codes). And arguments `PUSH`ed for helper calls are checked, hence the weird restrictions on help call preambles described above.

Another decision is that once a value is tested, it is thereafter regarded as defined, so that we do not emit multiple undefined-value errors for the same undefined value. That means that `TESTV` uinstrs are always followed by `SETV` on the same (shadow) `TempReg`s. Most of these `SETV`s are redundant and are removed by the post-instrumentation cleanup phase.

The instrumentation for calling helper functions deserves further comment. The definedness of results from a helper is modelled using just one V bit. So, in short, we do pessimising casts of the definedness of all the args, down to a single bit, and then `UifU` these bits together. So this single V bit will say "undefined" if any part of any arg is undefined. This V bit is then pessimally cast back up to the result(s) sizes, as needed. If, by seeing that all the args are got rid of with `CLEAR` and none with `POP`, Valgrind sees that the result of the call is not actually used, it immediately examines the result V bit with a `TESTV` -- `SETV` pair. If it did not do this, there would be no observation point to detect that the some of the args to the helper were undefined. Of course, if the helper's results are indeed used, we don't do this, since the result usage will presumably cause the result definedness to be checked at some suitable future point.

In general Valgrind tries to track definedness on a bit-for-bit basis, but as the above para shows, for calls to helpers we throw in the towel and approximate down to a single bit. This is because it's too complex and difficult to track bit-level definedness through complex ops such as integer multiply and divide, and in any case there is no reasonable code fragments which attempt to (eg) multiply two partially-defined values and end up with something meaningful, so there seems little point in modelling multiplies, divides, etc, in that level of detail.

Integer loads and stores are instrumented with firstly a test of the definedness of the address, followed by a `LOADV` or `STOREV` respectively. These turn into calls to (for example) `VG_(helperc_LOADV4)`. These helpers do two things: they perform an address-valid check, and they load or store V bits from/to the relevant address in the (simulated V-bit) memory.

FPU loads and stores are different. As above the definedness of the address is first tested. However, the helper routine for FPU loads (`VGM_(fpu_read_check)`) emits an error if either the address is invalid or the referenced area contains undefined values. It has to do this because we do not simulate the FPU at all, and so cannot track definedness of values loaded into it from memory, so we have to check them as soon as they are loaded into the FPU, ie, at this point. We notionally assume that everything in the FPU is defined.

It follows therefore that FPU writes first check the definedness of the address, then the validity of the address, and finally mark the written bytes as well-defined.

If anyone is inspired to extend Valgrind to MMX/SSE insns, I suggest you use the same trick. It works provided that the FPU/MMX unit is not used to merely as a conduit to copy partially undefined data from one place in memory to another. Unfortunately the integer CPU is used like that (when copying C structs with holes, for example) and this is the cause of much of the elaborateness of the instrumentation here described.

`vg_instrument()` in `vg_translate.c` actually does the instrumentation. There are comments explaining how each uinstr is handled, so we do not repeat that here. As explained already, it is bit-accurate, except for calls to helper functions. Unfortunately the x86 insns `bt/bts/btc/btr` are done by helper fns, so bit-level accuracy is lost there. This should be fixed by doing them inline; it will probably require adding a couple new uinstrs. Also, left and right rotates through the carry flag (x86 `rcl` and `rcr`) are approximated via a single V bit; so far this has not caused anyone to complain. The non-carry rotates, `rol` and `ror`, are much more common and are done exactly. Re-visiting the instrumentation for AND and OR, they seem rather verbose, and I wonder if it could be done more concisely now.

The lowercase `o` on many of the uopcodes in the running example indicates that the size field is zero, usually meaning a single-bit operation.

Anyroads, the post-instrumented version of our running example looks like this:

```
Instrumented code:
     0: GETVL     %EDX, q0
     1: GETL      %EDX, t0

     2: TAG1o     q0 = Left4 ( q0 )
     3: INCL      t0

     4: PUTVL     q0, %EDX
     5: PUTL      t0, %EDX

     6: TESTVL    q0
     7: SETVL     q0
     8: LOADVB    (t0), q0
     9: LDB       (t0), t0

    10: TAG1o     q0 = SWiden14 ( q0 )
    11: WIDENL_Bs t0

    12: PUTVL     q0, %EAX
    13: PUTL      t0, %EAX

    14: GETVL     %ECX, q8
    15: GETL      %ECX, t8

    16: MOVL      q0, q4
    17: SHLL      $0x1, q4
    18: TAG2o     q4 = UifU4 ( q8, q4 )
    19: TAG1o     q4 = Left4 ( q4 )
    20: LEA2L     1(t8,t0,2), t4

    21: TESTVL    q4
    22: SETVL     q4
    23: LOADVB    (t4), q10
    24: LDB       (t4), t10

    25: SETVB     q12
    26: MOVB      $0x20, t12

    27: MOVL      q10, q14
    28: TAG2o     q14 = ImproveAND1_TQ ( t10, q14 )
    29: TAG2o     q10 = UifU1 ( q12, q10 )
    30: TAG2o     q10 = DifD1 ( q14, q10 )
    31: MOVL      q12, q14
    32: TAG2o     q14 = ImproveAND1_TQ ( t12, q14 )
    33: TAG2o     q10 = DifD1 ( q14, q10 )
    34: MOVL      q10, q16
    35: TAG1o     q16 = PCast10 ( q16 )
    36: PUTVFo    q16
    37: ANDB      t12, t10  (-wOSZACP)

    38: INCEIPo   $9

    39: GETVFo    q18
    40: TESTVo    q18
    41: SETVo     q18
    42: Jnzo      $0x40435A50  (-rOSZACP)
```

```
    43: JMPo      $0x40435A5B
```

### 1.2.9.UCode post-instrumentation cleanup

This pass, coordinated by `vg_cleanup()`, removes redundant definedness computation created by the simplistic instrumentation pass. It consists of two passes, `vg_propagate_definedness()` followed by `vg_delete_redundant_SETVs`.

`vg_propagate_definedness()` is a simple constant-propagation and constant-folding pass. It tries to determine which `TempReg`s containing V bits will always indicate "fully defined", and it propagates this information as far as it can, and folds out as many operations as possible. For example, the instrumentation for an ADD of a literal to a variable quantity will be reduced down so that the definedness of the result is simply the definedness of the variable quantity, since the literal is by definition fully defined.

`vg_delete_redundant_SETVs` removes `SETV`s on shadow `TempReg`s for which the next action is a write. I don't think there's anything else worth saying about this; it is simple. Read the sources for details.

So the cleaned-up running example looks like this. As above, I have inserted line breaks after every original (non-instrumentation) uinstr to aid readability. As with straightforward ucode optimisation, the results in this block are undramatic because it is so short; longer blocks benefit more because they have more redundancy which gets eliminated.

```
at 29: delete UifU1 due to defd arg1
at 32: change ImproveAND1_TQ to MOV due to defd arg2
at 41: delete SETV
at 31: delete MOV
at 25: delete SETV
at 22: delete SETV
at 7: delete SETV

     0: GETVL     %EDX, q0
     1: GETL      %EDX, t0

     2: TAG1o     q0 = Left4 ( q0 )
     3: INCL      t0

     4: PUTVL     q0, %EDX
     5: PUTL      t0, %EDX

     6: TESTVL    q0
     8: LOADVB    (t0), q0
     9: LDB       (t0), t0

    10: TAG1o     q0 = SWiden14 ( q0 )
    11: WIDENL_Bs t0

    12: PUTVL     q0, %EAX
    13: PUTL      t0, %EAX

    14: GETVL     %ECX, q8
    15: GETL      %ECX, t8

    16: MOVL      q0, q4
    17: SHLL      $0x1, q4
    18: TAG2o     q4 = UifU4 ( q8, q4 )
    19: TAG1o     q4 = Left4 ( q4 )
    20: LEA2L     1(t8,t0,2), t4

    21: TESTVL    q4
    23: LOADVB    (t4), q10
    24: LDB       (t4), t10

    26: MOVB      $0x20, t12

    27: MOVL      q10, q14
    28: TAG2o     q14 = ImproveAND1_TQ ( t10, q14 )
    30: TAG2o     q10 = DifD1 ( q14, q10 )
    32: MOVL      t12, q14
    33: TAG2o     q10 = DifD1 ( q14, q10 )
    34: MOVL      q10, q16
    35: TAG1o     q16 = PCast10 ( q16 )
    36: PUTVFo    q16
    37: ANDB      t12, t10  (-wOSZACP)

    38: INCEIPo   $9
```

```
   39: GETVFo    q18
   40: TESTVo    q18
   42: Jnzo      $0x40435A50  (-rOSZACP)

   43: JMPo      $0x40435A5B
```

### 1.2.10.Translation from UCode

This is all very simple, even though `vg_from_ucode.c` is a big file. Position-independent x86 code is generated into a dynamically allocated array `emitted_code`; this is doubled in size when it overflows. Eventually the array is handed back to the caller of `VG_(translate)`, who must copy the result into TC and TT, and free the array.

This file is structured into four layers of abstraction, which, thankfully, are glued back together with extensive `__inline__` directives. From the bottom upwards:

- Address-mode emitters, `emit_amode_regmem_reg` et al.
- Emitters for specific x86 instructions. There are quite a lot of these, with names such as `emit_movv_offregmem_reg`. The `v` suffix is Intel parlance for a 16/32 bit insn; there are also `b` suffixes for 8 bit insns.
- The next level up are the `synth_*` functions, which synthesise possibly a sequence of raw x86 instructions to do some simple task. Some of these are quite complex because they have to work around Intel's silly restrictions on subregister naming.
  See `synth_nonshiftop_reg_reg` for example.
- Finally, at the top of the heap, we have `emitUInstr()`, which emits code for a single uinstr.

Some comments:

- The hack for FPU instructions becomes apparent here. To do a `FPU` ucode instruction, we load the simulated FPU's state into from its `VG_(baseBlock)` into the real FPU using an x86 `frstor` insn, do the ucode `FPU` insn on the real CPU, and write the updated FPU state back into `VG_(baseBlock)` using an `fnsave` instruction. This is pretty brutal, but is simple and it works, and even seems tolerably efficient. There is no attempt to cache the simulated FPU state in the real FPU over multiple back-to-back ucode FPU instructions.

  `FPU_R` and `FPU_W` are also done this way, with the minor complication that we need to patch in some addressing mode bits so the resulting insn knows the effective address to use. This is easy because of the regularity of the x86 FPU instruction encodings.
- An analogous trick is done with ucode insns which claim, in their `flags_r` and `flags_w` fields, that they read or write the simulated `%EFLAGS`. For such cases we first copy the simulated `%EFLAGS` into the real `%eflags`, then do the insn, then, if the insn says it writes the flags, copy back to `%EFLAGS`. This is a bit expensive, which is why the ucode optimisation pass goes to some effort to remove redundant flag-update annotations.

And so ... that's the end of the documentation for the instrumentating translator! It's really not that complex, because it's composed as a sequence of simple(ish) self-contained transformations on straight-line blocks of code.

### 1.2.11.Top-level dispatch loop

Urk. In `VG_(toploop)`. This is basically boring and unsurprising, not to mention fiddly and fragile. It needs to be cleaned up.

The only perhaps surprise is that the whole thing is run on top of a `setjmp`-installed exception handler, because, supposing a translation got a segfault, we have to bail out of the Valgrind-supplied exception handler `VG_(oursignalhandler)` and immediately start running the client's segfault handler, if it has one. In particular we can't finish the current basic block and then deliver the signal at some convenient future point, because signals like SIGILL, SIGSEGV and SIGBUS mean that the faulting insn should not simply be re-tried. (I'm sure there is a clearer way to explain this).

### 1.2.12.Lazy updates of the simulated program counter

Simulated `%EIP` is not updated after every simulated x86 insn as this was regarded as too expensive. Instead ucode `INCEIP` insns move it along as and when necessary. Currently we don't allow it to fall more than 4 bytes behind reality (see `VG_(disBB)` for the way this works).

Note that `%EIP` is always brought up to date by the inner dispatch loop in `VG_(dispatch)`, so that if the client takes a fault we know at least which basic block this happened in.

### 1.2.13.Signals

Horrible, horrible. `vg_signals.c`. Basically, since we have to intercept all system calls anyway, we can see when the client tries to install a signal handler. If it does so, we make a note of what the client asked to happen, and ask the kernel to route the signal to our own signal handler, `VG_(oursignalhandler)`. This simply notes the delivery of signals, and returns.

Every 1000 basic blocks, we see if more signals have arrived. If so, `VG_(deliver_signals)` builds signal delivery frames on the client's stack, and allows their handlers to be run. Valgrind places in these signal delivery frames a bogus return address, `VG_(signalreturn_bogusRA)`, and checks all jumps to see if any jump to it. If so, this is a sign that a signal handler is returning, and if so Valgrind removes the relevant signal frame from the client's stack, restores the from the signal frame the simulated state before the signal was delivered, and allows the client to run onwards. We have to do it this way because some signal handlers never return, they just `longjmp()`, which nukes the signal delivery frame.

The Linux kernel has a different but equally horrible hack for detecting signal handler returns. Discovering it is left as an exercise for the reader.

### 1.2.14.&#65533;To be written

The following is a list of as-yet-not-written stuff. Apologies.

1. The translation cache and translation table
2. Exceptions, creating new translations
3. Self-modifying code
4. Errors, error contexts, error reporting, suppressions
5. Client malloc/free
6. Low-level memory management
7. A and V bitmaps
8. Symbol table management
9. Dealing with system calls
10. Namespace management
11. GDB attaching
12. Non-dependence on glibc or anything else
13. The leak detector
14. Performance problems
15. Continuous sanity checking
16. Tracing, or not tracing, child processes
17. Assembly glue for syscalls

## 1.3.&#65533;Extensions

Some comments about Stuff To Do.

### 1.3.1.&#65533;Bugs

Stephan Kulow and Marc Mutz report problems with kmail in KDE 3 CVS (RC2 ish) when run on Valgrind. Stephan has it deadlocking; Marc has it looping at startup. I can't repro either behaviour. Needs repro-ing and fixing.

### 1.3.2.&#65533;Threads

Doing a good job of thread support strikes me as almost a research-level problem. The central issues are how to do fast cheap locking of the `VG_(primary_map)` structure, whether or not accesses to the individual secondary maps need locking, what race-condition issues result, and whether the already-nasty mess that is the signal simulator needs further hackery.

I realise that threads are the most-frequently-requested feature, and I am thinking about it all. If you have guru-level understanding of fast mutual exclusion mechanisms and race conditions, I would be interested in hearing from you.

### 1.3.3.&#65533;Verification suite

Directory `tests/` contains various ad-hoc tests for Valgrind. However, there is no systematic verification or regression suite, that, for example, exercises all the stuff in `vg_memory.c`, to ensure that illegal memory accesses and undefined value uses are detected as they should be. It would be good to have such a suite.

### 1.3.4.&#65533;Porting to other platforms

It would be great if Valgrind was ported to FreeBSD and x86 NetBSD, and to x86 OpenBSD, if it's possible (doesn't OpenBSD use a.out-style executables, not ELF ?)

The main difficulties, for an x86-ELF platform, seem to be:

- You'd need to rewrite the `/proc/self/maps` parser (`vg_procselfmaps.c`). Easy.
- You'd need to rewrite `vg_syscall_mem.c`, or, more specifically, provide one for your OS. This is tedious, but you can implement syscalls on demand, and the Linux kernel interface is, for the most part, going to look very similar to the *BSD interfaces, so it's really a copy-paste-and-modify-on-demand job. As part of this, you'd need to supply a new `vg_kerneliface.h` file.

- You'd also need to change the syscall wrappers for Valgrind's internal use, in `vg_mylibc.c`.

All in all, I think a port to x86-ELF *BSDs is not really very difficult, and in some ways I would like to see it happen, because that would force a more clear factoring of Valgrind into platform dependent and independent pieces. Not to mention, *BSD folks also deserve to use Valgrind just as much as the Linux crew do.

## 1.4.&#65533;Easy stuff which ought to be done

### 1.4.1.&#65533;MMX Instructions

MMX insns should be supported, using the same trick as for FPU insns. If the MMX registers are not used to copy uninitialised junk from one place to another in memory, this means we don't have to actually simulate the internal MMX unit state, so the FPU hack applies. This should be fairly easy.

### 1.4.2.&#65533;Fix stabs-info reader

The machinery in `vg_symtab2.c` which reads "stabs" style debugging info is pretty weak. It usually correctly translates simulated program counter values into line numbers and procedure names, but the file name is often completely wrong. I think the logic used to parse "stabs" entries is weak. It should be fixed. The simplest solution, IMO, is to copy either the logic or simply the code out of GNU binutils which does this; since GDB can clearly get it right, binutils (or GDB?) must have code to do this somewhere.

### 1.4.3.&#65533;BT/BTC/BTS/BTR

These are x86 instructions which test, complement, set, or reset, a single bit in a word. At the moment they are both incorrectly implemented and incorrectly instrumented.

The incorrect instrumentation is due to use of helper functions. This means we lose bit-level definedness tracking, which could wind up giving spurious uninitialised-value use errors. The Right Thing to do is to invent a couple of new UOpcodes, I think `GET_BIT` and `SET_BIT`, which can be used to implement all 4 x86 insns, get rid of the helpers, and give bit-accurate instrumentation rules for the two new UOpcodes.

I realised the other day that they are mis-implemented too. The x86 insns take a bit-index and a register or memory location to access. For registers the bit index clearly can only be in the range zero to register-width minus 1, and I assumed the same applied to memory locations too. But evidently not; for memory locations the index can be arbitrary, and the processor will index arbitrarily into memory as a result. This too should be fixed. Sigh. Presumably indexing outside the immediate word is not actually used by any programs yet tested on Valgrind, for otherwise they (presumably) would simply not work at all. If you plan to hack on this, first check the Intel docs to make sure my understanding is really correct.

### 1.4.4.&#65533;Using PREFETCH Instructions

Here's a small but potentially interesting project for performance junkies. Experiments with valgrind's code generator and optimiser(s) suggest that reducing the number of instructions executed in the translations and mem-check helpers gives disappointingly small performance improvements. Perhaps this is because performance of Valgrindified code is limited by cache misses. After all, each read in the original program now gives rise to at least three reads, one for the `VG_(primary_map)`, one of the resulting secondary, and the original. Not to mention, the instrumented translations are 13 to 14 times larger than the originals. All in all one would expect the memory system to be hammered to hell and then some.

So here's an idea. An x86 insn involving a read from memory, after instrumentation, will turn into ucode of the following form:

```
... calculate effective addr, into ta and qa ...
  TESTVL qa             -- is the addr defined?
  LOADV (ta), qloaded   -- fetch V bits for the addr
  LOAD  (ta), tloaded   -- do the original load
```

At the point where the `LOADV` is done, we know the actual address (`ta`) from which the real `LOAD` will be done. We also know that the `LOADV` will take around 20 x86 insns to do. So it seems plausible that doing a prefetch of `ta` just before the `LOADV` might just avoid a miss at the `LOAD` point, and that might be a significant performance win.

Prefetch insns are notoriously tempermental, more often than not making things worse rather than better, so this would require considerable fiddling around. It's complicated because Intels and AMDs have different prefetch insns with different semantics, so that too needs to be taken into account. As a general rule, even placing the prefetches before the `LOADV` insn is too near the `LOAD`; the ideal distance is apparently circa 200 CPU cycles. So it might be worth having another analysis/transformation pass which pushes prefetches as far back as possible, hopefully immediately after the effective address becomes available.

Doing too many prefetches is also bad because they soak up bus bandwidth / cpu resources, so some cleverness in deciding which loads to prefetch and which to not might be helpful. One can imagine not prefetching client-stack-relative (`%EBP` or `%ESP`) accesses, since the stack in general tends to show good locality anyway.

There's quite a lot of experimentation to do here, but I think it might make an interesting week's work for someone.

As of 15-ish March 2002, I've started to experiment with this, using the AMD `prefetch/prefetchw` insns.

### 1.4.5.&#65533;User-defined Permission Ranges

This is quite a large project -- perhaps a month's hacking for a capable hacker to do a good job -- but it's potentially very interesting. The outcome would be that Valgrind could detect a whole class of bugs which it currently cannot.

The presentation falls into two pieces.

#### 1.4.5.1.Part 1: User-defined Address-range Permission Setting

Valgrind intercepts the client's `malloc`, `free`, etc calls, watches system calls, and watches the stack pointer move. This is currently the only way it knows about which addresses are valid and which not. Sometimes the client program knows extra information about its memory areas. For example, the client could at some point know that all elements of an array are out-of-date. We would like to be able to convey to Valgrind this information that the array is now addressable-but-uninitialised, so that Valgrind can then warn if elements are used before they get new values.

What I would like are some macros like this:

```
  VALGRIND_MAKE_NOACCESS(addr, len)
  VALGRIND_MAKE_WRITABLE(addr, len)
  VALGRIND_MAKE_READABLE(addr, len)
```

and also, to check that memory is addressible/initialised,

```
  VALGRIND_CHECK_ADDRESSIBLE(addr, len)
  VALGRIND_CHECK_INITIALISED(addr, len)
```

I then include in my sources a header defining these macros, rebuild my app, run under Valgrind, and get user-defined checks.

Now here's a neat trick. It's a nuisance to have to re-link the app with some new library which implements the above macros. So the idea is to define the macros so that the resulting executable is still completely stand-alone, and can be run without Valgrind, in which case the macros do nothing, but when run on Valgrind, the Right Thing happens. How to do this? The idea is for these macros to turn into a piece of inline assembly code, which (1) has no effect when run on the real CPU, (2) is easily spotted by Valgrind's JITter, and (3) no sane person would ever write, which is important for avoiding false matches in (2). So here's a suggestion:

```
  VALGRIND_MAKE_NOACCESS(addr, len)
```

becomes (roughly speaking)

```
  movl addr, %eax
  movl len,  %ebx
  movl $1,   %ecx   -- 1 describes the action; MAKE_WRITABLE might be
                    -- 2, etc
  rorl $13, %ecx
  rorl $19, %ecx
  rorl $11, %eax
  rorl $21, %eax
```

The rotate sequences have no effect, and it's unlikely they would appear for any other reason, but they define a unique byte-sequence which the JITter can easily spot. Using the operand constraints section at the end of a gcc inline-assembly statement, we can tell gcc that the assembly fragment kills `%eax`, `%ebx`, `%ecx` and the condition codes, so this fragment is made harmless when not running on Valgrind, runs quickly when not on Valgrind, and does not require any other library support.

#### 1.4.5.2.Part 2: Using it to detect Interference between Stack Variables

Currently Valgrind cannot detect errors of the following form:

```
void fooble ( void )
{
  int a[10];
  int b[10];
  a[10] = 99;
}
```

Now imagine rewriting this as

```
void fooble ( void )
{
  int spacer0;
  int a[10];
  int spacer1;
  int b[10];
  int spacer2;
  VALGRIND_MAKE_NOACCESS(&spacer0, sizeof(int));
  VALGRIND_MAKE_NOACCESS(&spacer1, sizeof(int));
  VALGRIND_MAKE_NOACCESS(&spacer2, sizeof(int));
```

```
  a[10] = 99;
}
```

Now the invalid write is certain to hit `spacer0` or `spacer1`, so Valgrind will spot the error.

There are two complications.

1. The first is that we don't want to annotate sources by hand, so the Right Thing to do is to write a C/C++ parser, annotator, prettyprinter which does this automatically, and run it on post-CPP'd C/C++ source. The parser/prettyprinter is probably not as hard as it sounds; I would write it in Haskell, a powerful functional language well suited to doing symbolic computation, with which I am intimately familar. There is already a C parser written in Haskell by someone in the Haskell community, and that would probably be a good starting point.

2. The second complication is how to get rid of these `NOACCESS` records inside Valgrind when the instrumented function exits; after all, these refer to stack addresses and will make no sense whatever when some other function happens to re-use the same stack address range, probably shortly afterwards. I think I would be inclined to define a special stack-specific macro:

   ```
   VALGRIND_MAKE_NOACCESS_STACK(addr, len)
   ```

   which causes Valgrind to record the client's `%ESP` at the time it is executed. Valgrind will then watch for changes in `%ESP` and discard such records as soon as the protected area is uncovered by an increase in `%ESP`. I hesitate with this scheme only because it is potentially expensive, if there are hundreds of such records, and considering that changes in `%ESP` already require expensive messing with stack access permissions.

This is probably easier and more robust than for the instrumenter program to try and spot all exit points for the procedure and place suitable deallocation annotations there. Plus C++ procedures can bomb out at any point if they get an exception, so spotting return points at the source level just won't work at all.

Although some work, it's all eminently doable, and it would make Valgrind into an even-more-useful tool.