Valgrind: A Program Supervision Framework

Nicholas Nethercote[a] and Julian Seward[b]

[a]Computer Laboratory, Cambridge University

[b]Cambridge, UK

`http://www.cl.cam.ac.uk/~njn25/`

`http://developer.kde.org/~sewardj`

13 July 2003

Overview

Valgrind is a:

- Framework for building tools
- Tools that *supervise* programs at runtime

What I will discuss:

- Why such a framework is good
- Example tools built using it
- Performance
- Strengths and weaknesses

What I won't discuss:

- Gory technical details – read the paper

Supervision Tools

Lots of them:

- Profilers
- Bug detectors (e.g. memory debuggers, data race detectors)
- Program visualisation, comprehension

All rely on instrumenting code:

- Retain original behaviour
- Do some extra stuff, too

Problems

UNIVERSITY OF CAMBRIDGE

Building supervision tools is hard:

- Instrumentation itself often straightforward. . .
- . . . but adding the instrumentation is difficult

Want a common infrastructure.

Using supervision tools can be hard:

- Recompile and/or relink
- Incomplete coverage
- Libraries are a pain

Want to avoid these problems.

Building tools is easy(ish)

Valgrind core + skin = supervision tool

- Write a *skin* (plug-in) that defines an instrumentation pass
- Core inserts instrumentation into running program (hard part)
- Core provides services (make skin-writing easier)

Core	55,000 lines
C/C++ memory debugger	7,400 lines
Dataflow tracer	5,800 lines
Data race detector	3,500 lines
Cache profiler	2,400 lines
Instruction counter	60 lines

The hard part is done for you; don't reinvent the wheel.

Using tools is easy

Normal use: `./myapp <args>`

Under Valgrind: `valgrind --skin=<name> ./myapp <args>`

Thanks to *dynamic binary translation*:

- No recompilation needed
- No relinking needed
- No source code needed
- *All* code outside kernel (including libraries) instrumented
- Skin selected at startup

Isn't that easy?

Status

Distribution:

- First released: early 2002
- Current version: 1.9.6
- x86/Linux, GPL
- Not a toy: runs large programs (e.g. Mozilla, OpenOffice)
- Four skins

Widely used:

- At least hundreds of users
- Used extensively for KDE 3.0

Execution Basics

At runtime:

- Valgrind hijacks execution of client at startup
- Client runs on simulated CPU
- Client doesn't realise
- Valgrind x86-to-x86 JIT compiles code
- Skin adds instrumentation
- No original client code runs, only compiled (instrumented) code

Execution Details

Actually an x86-*UCode*-x86 compiler.

UCode is intermediate format:

- RISC-like
- Expressed in virtual registers

Compilation of a basic block:

1. Disassembly (core):	x86	→ UCode
2. Optimisation (core):	UCode	→ UCode
3. Instrumentation (skin):	UCode	→ Instr. UCode
4. Register alloc. (core):	Instr. UCode	→ Instr. UCode
5. Code generation (core):	Instr. UCode	→ x86

Generated code is cached and linked (core).

Skins

Must implement:

- `init()`
- `instrument()`
- `fini()`

Instrumentation:

- Calls to C functions easiest (via `CCALL` UCode instruction)
- Can extend UCode for fine-grained instrumentation

Some callbacks must be provided if using certain core services.

More details in paper.

Memcheck: A C/C++ memory debugger

Similar to Purify. Detects:

- Use of uninitialised memory;
- Accessing memory before start/past end of heap blocks;
- Mismatched `malloc()`/`new`/`new[]` vs. `free()`/`delete`/`delete[]`;
- Attempts to free non-heap blocks;
- Accessing heap blocks after they have been freed;
- Memory leaks;
- Accessing inappropriate areas on the stack;
- Passing uninitialised/unaddressable memory to system calls;
- Overlapping source/destination areas for `memcpy()`, etc.

Errors pin-pointed to a single line of code.

Memcheck

Memory: each byte *shadowed* by:

- 8 validity (V) bits
- 1 addressability (A) bit

Shadow maps created in 64KB chunks as needed.

Registers: each 32-bit register shadowed by:

- 32 validity (V) bits

A bits:

- Checked for every memory access
- Updated upon `malloc()`, `free()`, stack grows/shrinks, etc.

Memcheck

V bits checked on:

- Conditional branch tests
- Syscall inputs
- Address computations

Not checked on:

- Copies (copying uninitialised bits is ok, common)
- Arithmetic ops (doesn't affect external behaviour, yet)

Updated on copies, value-producing computations

Performance

SPEC2000 slowdown factors:

	Nulgrind	Memcheck	Addrcheck	Cachegrind
min	1.8	11.6	5.9	18.5
max	8.5	47.9	32.7	107.4
median	5.2	28.7	18.5	49.9

Paper has more figures, including space performance.

Correctness is more important than performance!

Strengths and Weaknesses

UNIVERSITY OF CAMBRIDGE

Similar frameworks (e.g. DynamoRIO, DELI, Strata):

- Instrument machine code directly
- Good for dynamic optimisation
- Good for simple instrumentation

Valgrind:

- Too slow for optimisation
- Better for complex instrumentation

UCode advantages:

- Simple (RISC-like; compare to x86)
- Unconstrained by original code (e.g. enough registers?)
- No fear of changing original code's effect (e.g. condition codes?)

Also services (error recording, debug info, etc.) are very useful.

Future Work

UNIVERSITY OF CAMBRIDGE

Skins:

- Not just profiling
- *Deep* bug detection (metavalues)
- More formal verification tools?

Core:

- Different skins (done, for x86-only case)
- Different architectures (doable)
- Different OS/environments (difficult, esp. signals, threads)

Avoid M×N×P code blow-up.

Take-home message

UNIVERSITY OF CAMBRIDGE

With Valgrind:

- Building supervision tools is easy(ish)
- Running supervision tools is easy

Available at `http://developer.kde.org/~sewardj`