Building Workload Characterization Tools with Valgrind

Nicholas Nethercote - National ICT Australia
Robert Walsh - Qlogic Corporation
Jeremy Fitzhardinge - XenSource
This tutorial

1. Introduction to Valgrind
2. Example profiling tools
3. Building a new Valgrind tool
4. More advanced tools
(end of tutorial overview)
1. Introduction to Valgrind

Robert Walsh
This talk

• What is Valgrind?
• Who uses it?
• How it works
What is Valgrind?
Valgrind is…

• A framework
  – For building program analysis tools
  – E.g. profilers, visualizers, checkers

• A software package, containing:
  – Framework core
  – Several tools: memory checker, cache profiler, call graph profiler, heap profiler

• Memcheck, the most widely used tool, is often synonymous with “Valgrind”
What kind of analysis? (1/2)

• Categorization 1: when does analysis occur?
  – Before run-time: static analysis
    • Simple preliminaries: parsing
    • Complex analysis: e.g. abstract interpretation
    • Imprecise, but can be sound: sees all execution paths
  – At run-time: dynamic analysis
    • Complex preliminaries: instrumentation
    • Simpler analysis: “Perfect light of run-time”
    • Powerful, but unsound: sees one execution path

• Valgrind performs dynamic analysis
What kind of analysis? (2/2)

• Categorization 2: what code is analyzed?
  – Source code: source-level analysis
    • Language-specific
    • Requires source code
    • High-level information: e.g. variables, statements
  – Machine code: binary analysis
    • Language-independent (can be multi-language)
    • No source code (but debug info helps)
    • Lower-level information: e.g. registers, instructions

• Valgrind performs binary analysis
Dynamic binary analysis

<table>
<thead>
<tr>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Static source-level analysis</td>
</tr>
<tr>
<td>Binary</td>
<td>Static binary analysis</td>
</tr>
<tr>
<td></td>
<td>Dynamic binary analysis</td>
</tr>
</tbody>
</table>

- Valgrind: dynamic binary analysis (DBA)
  - Analysis of machine code at run-time
  - *Instrument* original code with *analysis code*
  - Track some extra information: *metadata*
  - Do some extra I/O, but don’t disturb execution otherwise
What kind of instrumentation?

- **Categorization:** When does binary instrumentation occur?
  - Before run-time: static binary instrumentation (SBI)
    - A.k.a. binary rewriting
  - At run-time: dynamic binary instrumentation (DBI)

- **Valgrind uses DBI. Compared to SBI:**
  - No preparation (e.g. recompilation) required
  - All user-mode code instrumented
    - Dynamically loaded libraries
    - Dynamically generated code
    - No code/data identification difficulties
  - Instrumentation cost incurred at run-time

- **A good DBI framework mitigates the run-time cost and makes tool-writing much easier**
An aside

• Similar things to DBA and DBI:
  – 1. Dynamic binary optimisation
    • Rewrite binary on-the-fly for speed-ups
    • E.g. Dynamo
  – 2. Dynamic binary translation
    • Run binary for platform X on platform Y
  – 3. Semantics-affecting tools
    • E.g. sandboxing, fault injection

• Not talking about these
  – Valgrind tools can do (3), but usually don’t
Similar systems

- DBI frameworks:
  - Pin, DynamoRIO, DIOTA, DynInst, etc.
  - Lots of overlap
  - Each system supports different platforms
- Purify, Chaperon (part of Insure++)
  - Memcheck (a memory-checking tool) is similar
- Valgrind:
  - GPL
  - Widely used, robust
  - Slower for simple tools
  - Designed for heavyweight tools, especially shadow value tools
    (more in talk 4)
Who uses Valgrind?
Valgrind users

• Developers
  – C (43%), C++ (49%), Fortran, Ada, Java
  – Firefox, OpenOffice, KDE, GNOME, libstdc++, PHP, Perl, Python, MySQL, Samba, RenderMan, NASA, CERN, Unreal Tournament, parts of the Linux kernel
  – Biggest program we know of: 25 MLOC
  – Memcheck: 80% of usage, other tools still widespread

• Researchers
  – Cambridge, MIT, CMU, UT, UNM, ANU, etc.
  – For building new kinds of analysis tools
  – For experimental evaluation of programs (CacheGrind)

• Website receives >1000 unique visitors per day
Availability

• Free software (GPL)
• Standard Linux package
• Platforms:
  – Valgrind 3.2.1: x86/Linux, AMD64/Linux, PPC\{32,64\}/Linux
  – In repository: PPC\{32,64\}/AIX
  – Under development: PPC32/Darwin, x86/Darwin, x86/FreeBSD, others
• www.valgrind.org
How does Valgrind work?
Basic architecture

• Valgrind core + tool plug-in = Valgrind tool
• Core:
  – Executes the client program under its control
  – Provides services to aid tool-writing
    • E.g. error recording, debug info reading
• Tool plug-ins:
  – Main job: instrument code blocks passed by the core
• Lines of code (mostly C, a little asm in the core):
  – Core: 173,000
  – Call graph profiler: 11,800
  – Cache profiler: 2,400
  – Heap profiler: 1,700
Running a Valgrind tool (1/2)

[nevermore:~] date
Sat Oct 14 10:28:03 EST 2006
[nevermore:~] valgrind --tool=cachegrind date
==17789== Cachegrind, an I1/D1/L2 cache profiler.
==17789== Copyright (C) 2002-2006, and GNU GPL'd, by Nicholas Nethercote et al.
==17789== Using LibVEX rev 1601, a library for dynamic binary translation.
==17789== Copyright (C) 2004-2006, and GNU GPL'd, by OpenWorks LLP.
==17789== Using valgrind-3.2.1, a dynamic binary instrumentation framework.
==17789== Copyright (C) 2000-2006, and GNU GPL'd, by Julian Seward et al.
==17789== For more details, rerun with: -v
==17789==
Sat Oct 14 10:28:12 EST 2006
==17789==
==17789== I   refs:      395,633
==17789== I1  misses:      1,488
==17789== L2i misses:      1,404
==17789== I1  miss rate:    0.37%
==17789== L2i miss rate:    0.35%
==17789==
==17789== D   refs:      191,453  (139,922 rd + 51,531 wr)
==17789== D1  misses:      3,012  (  2,467 rd +    545 wr)
==17789== L2d misses:      1,980  (  1,517 rd +    463 wr)
==17789== D1  miss rate:     1.5% (    1.7%   +    1.0%  )
==17789== L2d miss rate:     1.0% (    1.0%   +    0.8%  )
==17789==
==17789== L2 refs:         4,500  (  3,955 rd +    545 wr)
==17789== L2 misses:       3,384  (  2,921 rd +    463 wr)
==17789== L2 miss rate:      0.5% (    0.5%   +    0.8%  )
Running a Valgrind tool (2/2)

• Tool output goes to stderr, file, fd or socket
• Program behaviour otherwise unchanged…
• …except much slower than normal
  – No instrumentation: 4-10x
  – Memcheck: 10-60x
  – Cachegrind: 20-100x
• For most tools, slow-down mostly due to analysis code
Starting up

• Valgrind loads the core, chosen tool and client program into a single process

• Lots of resource conflicts to handle, via:
  – Partitioning: address space, fds
  – Time-multiplexing: registers
  – Sharing: pid, current working directory, etc.

• Starting up is difficult to do robustly
  – Currently on our 3rd core/tool structuring and start-up mechanism!
Dynamic binary recompilation

- JIT translation of small code blocks
  - Often basic blocks, but can contain jumps
  - Typically 5-30 instructions

- Before a code block is executed for the first time:
  - Core: machine code $\rightarrow$ (architecture neutral) IR
  - Tool: IR $\rightarrow$ instrumented IR
  - Core: instrumented IR $\rightarrow$ instrumented machine code
  - Core: caches and links generated translations

- No original code is run

- Valgrind controls every instruction
  - Client is none the wiser
Complications

• System calls
  – Valgrind does not trace into the kernel
  – Some are checked to avoid core/tool conflicts
  – Blocking system calls require extra care

• Signals
  – Valgrind intercepts handler registration and delivery
  – Required to avoid losing control

• Threads
  – Valgrind serializes execution (one thread at a time)
  – Avoids subtle data races in tools
  – Requires reconsideration due to architecture trends
Function wrapping/replacement

• Function replacement
  – Can replace arbitrary functions
  – Replacement runs as if native (i.e. it is instrumented)

• Function wrapping
  – Replacement functions can call the function they replaced
  – This allows function wrapping
  – Wrappers can observe function arguments

• System call wrapping
  – Similar functionality to function wrapping
  – But separate mechanism
Client requests

• Trap-door mechanism
  – An unusual no-op instruction sequence
  – Under Valgrind, it transfers control to core/tool
  – Client can pass queries and messages to the core/tool
  – Allow arguments and a return value
  – Augments tool’s standard instrumentation

• Easy to put in source code via macros
  – Tools only need to include a header file to use them
  – They do nothing when running natively
  – Tool-specific client requests ignored by other Valgrind tools

• Example:
  – Memcheck instruments malloc and free
  – Custom allocators can be marked with client requests that say “a heap block was just allocated/freed”
  – A little extra user effort helps Memcheck give better results
Self-modifying code

- Without care, self-modifying code won’t run correctly
  - Dynamically generated code is fine if it doesn’t change
  - But if changed, the old translations will be executed
- An automatic mechanism:
  - Hash of original code checked before each translation is executed
  - Expensive, by default on only for code on the stack
  - E.g. handles GCC trampolines for nested functions (esp. for Ada)
- A manual mechanism:
  - A built-in client request: “discard existing translations for address range A..B”
  - Useful for dynamic code generators, e.g. JIT compilers
Forests and trees

- Valgrind is a framework for building DBA tools
- Interesting in and of itself
  - But it is a means to an end
- The tools themselves are the interesting part
  - Actually, it is what the tools can tell you about programs that is really the interesting part
- Next three talks cover:
  - Existing profiling tools
  - How to write new tools
  - Some ideas for interesting new tools
(end of talk 1)
2. Example profiling tools

Jeremy Fitzhardinge
This talk

• Three existing profiling tools
  – Cache profiler
  – Call graph profiler
  – Heap profiler
Cacheegrind: a cache profiler
Cache behaviour is crucial
  – L1 misses: ~10 cycles
  – L2 misses: ~200 cycles

But difficult to predict

Cachegrind gives three outputs:
  – Total hit/miss counts and ratios (I1, D1, L2)
  – Per-function hit/miss counts (sorted from most to least)
  – Per-line hit/miss counts (source code annotations)

Source code annotations are the most useful
  – Most fine-grained data
  – Data that programmers can act on to speed up their programs
Sample output

<table>
<thead>
<tr>
<th>Ir</th>
<th>I1mr</th>
<th>I2mr</th>
<th>Dr</th>
<th>D1mr</th>
<th>D2mr</th>
<th>Dw</th>
<th>D1mw</th>
<th>D2mw</th>
</tr>
</thead>
<tbody>
<tr>
<td>14,789,396</td>
<td>547</td>
<td>544</td>
<td>6,329,792</td>
<td>751</td>
<td>689</td>
<td>2,111,757</td>
<td>1,113,292</td>
<td>1,094,855</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ir</th>
<th>I1mr</th>
<th>I2mr</th>
<th>Dr</th>
<th>D1mr</th>
<th>D2mr</th>
<th>Dw</th>
<th>D1mw</th>
<th>D2mw</th>
<th>file:</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>14,688,273</td>
<td>1</td>
<td>1</td>
<td>6,294,531</td>
<td>0</td>
<td>0</td>
<td>2,098,178</td>
<td>1,113,088</td>
<td>1,094,656</td>
<td>example.c:</td>
<td>main</td>
</tr>
</tbody>
</table>

-- Auto-annotated source: example.c

```c
int main(void)
{
    int i, j, a[1024][1024];
    for (i = 0; i < 1024; i++) {
        for (j = 0; j < 1024; j++) {
            a[i][j] = 0; // fast
            a[j][i] = 0; // slow
        }
    }
    return 0;
}
```
How Cachegrind works

- Each instruction is instrumented
  - Call to a C cache simulation function
  - Different functions for loads, stores, modifies
  - Some combining of C calls for efficiency
- Each source code line gets a *cost centre*
  - Holds counters: accesses, hits and misses
  - Uses debug info to map each instruction to a cost centre
- Online simulation (i.e. no trace gathering)
- Cost centres dumped to file at end
  - Simple but compact text format
  - Post-processing script produces previous slide’s output
Cache simulation

- Approximates an AMD Athlon hierarchy
  - I1, D1, inclusive L2
  - Write-allocate
  - LRU replacement

- Each cache is command-line configurable:
  - Cache size
  - Line size
  - Associativity

- On x86/AMD64 can use CPUID to auto-detect these parameters

- Simulation can be replaced easily
Inaccuracies

- **Imperfect address trace**
  - No kernel code
  - Other processes ignored (arguably good)
  - Conversion to Valgrind’s IR changes a very small number of loads/stores

- **Incorrect addresses**
  - Virtual addresses
  - Memory layout and thread scheduling is different under CacheGrind compared to native

- **Prefetches and cache-bypassing are ignored**
  - Difficult to handle well without detailed microarchitectural simulation

- **Still useful for general insights**
How is it used?

• Characterization:
  – Program A vs. program B
  – Cache hierarchy A vs. cache hierarchy B

• Optimisation:
  – Identifies cache-unfriendly code
  – Fixing such code requires non-trivial insight
    • But easier (i.e. not impossible!) than fixing without this data

• Evaluation of optimisations:
  – Program A vs. optimised program A
Cachegrind summary

- Cachegrind is a cache simulator
- Gives total, per-function and per-line hit/miss counts
- Simulation is imperfect, but still useful
- Used for characterization, optimisation and evaluation
Callgrind: a call graph profiler
Callgrind

• Extension of Cachegrind
• By Josef Weidendorfer
• Also provides:
  – Call graph information
  – Graphical results viewer (KCachegrind)
    • Allows interactive browsing of results
    • Accepts Cachegrind results also
  – Greater selectivity of what code is profiled
KCachegrind’s tree-map view

- Box sizes represent relative counts
- Nesting of boxes represents call chains
- Interactive: can drill down through boxes
KCacheGrind’s call graph view

- Shows whole call graph
- Boxes show count proportions
- Interactive
Selective profiling

• Can dump counts at particular times
  – At termination (same as Cachegrind)
  – Periodically (every N code blocks)
  – At entry/exit of named functions
  – At particular program points (using client requests)
  – At any time (by invoking a separate script)

• Counters are zeroed after each dump

• Can choose which events to count
  – Instructions
  – Memory events (for cache simulation)
  – Function entries/exits
An interesting difficulty

• Callgrind maintains a call stack
  – For tracking function entries/exits

• Several difficulties:
  – `setjmp/longjmp`
  – Tail recursion
  – Dynamic linking
    • Calls through jump tables
    • Jump table patched on first call after loading
  – Stack switching

• Missed entries/exits can throw everything out
Interesting lessons

• Good tools go beyond the basics
  – Results presentation
  – Analysis selectivity

• Some tool tasks are more difficult than you would expect
Massif: a heap profiler
Massif heap graph
Massif

• Measures heap and stack
  – Each heap allocation site is a band
  – Stack is a band

• Also produces HTML output
  – Represents the call graph underlying allocations
  – Users can drill down through calling chains from allocation sites

• Simple interaction with Valgrind’s core
  – Only uses function wrapping
  – No instrumentation of code blocks
  – Complexity in the tool, not at the core/tool boundary
Summary

- Cachegrind, Callgrind, Massif
- Three different profilers
  - Not necessarily what you need
  - Demonstrate the kinds of things you can do
- Next: details of how to write a tool
(end of talk 2)
3. Building a new Valgrind tool

Nicholas Nethercote
This talk

• How to write a new tool from scratch
  – Simple but useful example: memory tracer
  – Start with simplest version
  – Improve its accuracy and performance
A new tool from scratch
Memtrace

• Example tool

• Trace memory (data) accesses
  – Loads, stores, modifies

• Print entry for each memory access
  – Data address
  – Data size
Tool basics

• Tools must provide functions for 3 tasks:
  – Initialization
  – Instrumentation
  – Finalization

• Analysis code can be added
  – Inline
  – Calls to C functions

• Tools provide functions that help the core provide certain services
  – E.g. error reporting, options processing
Build environment

• In what follows, all filenames are relative to top-level Valgrind directory

• Valgrind uses automake/autoconf
  – Use an SVN version of Valgrind; this simplifies Makefile handling
  – www.valgrind.org explains how to get the SVN version
Preliminaries

• Create empty directories:
  - memtrace/
  - memtrace/docs/
  - memtrace/tests/

• Create empty files:
  - memtrace/docs/Makefile.am
  - memtrace/tests/Makefile.am

• Copy none/Makefile.am to memtrace/

• Edit files:
  – Add three entries to AC_OUTPUT in configure.in:
    • memtrace/Makefile
    • memtrace/docs/Makefile
    • memtrace/tests/Makefile
  – Add memtrace to TOOLS in Makefile.am
  – Change names within memtrace/Makefile.am appropriately:
    • s/none/memtrace/
    • s/nl_/mt_/
First `mt_main.c` (1/3)

- **Create** `memtrace(mt_main).c`
  - Two-letter prefix is just a convention

```c
#include "pub_tool Basics.h"     // Needed by every tool
#include "pub_tool Tooliface.h"  // Needed by every tool
#include "pub_tool Libcprint.h"  // For printing functions
#include "pub_tool Machine.h"    // For VG_(fnptr_to_fnentry)
```

- **Most tool-visible headers in** `include/pub_tool*_h`
- **Next: four functions must be defined**
  - Pre-option-processing initialization
  - Post-option-processing initialization
  - Instrumentation
  - Finalization
static void mt_pre_clo_init(void)
{
    // Required details for start-up message
    VG_(details_name) ("Memtrace");
    VG_(details_version) ("0.1");
    VG_(details_description) ("a memory tracer");
    VG_(details_copyright_author) ("Copyright (C) 2006, J. Random Hacker.");
    // Required detail for crash message
    VG_(details_bug_reports_to) ("/dev/null");

    // Name the required functions #2, #3 and #4.
    VG_(basic_tool_funcs) (mt_post_clo_init,
                            mt_instrument,
                            mt_fini);
}

    // This prevents core/tool interface problems, and names the required
    // function #1, giving the core an entry point into the tool.
    VG_DETERMINE_INTERFACE_VERSION(mt_pre_clo_init)
First mt_main.c (3/3)

// Post-option-processing initialization function
static void mt_post_clo_init(void) { }

// Instrumentation function. "bbIn" is the code block.
// Others arguments are more obscure and often not needed -- see
// include/pub_tool_tooliface.h.
static IRBB* mt_instrument ( VgCallbackClosure* closure,
                           IRBB* bbIn,
                           VexGuestLayout* layout,
                           VexGuestExtents* vge,
                           IRTyp gWordTy, IRTyp hWordTy )
{
    return bbIn;
}

// Finalization function
static void mt_fini(Int exitcode) { }

• (These functions must precede mt_pre_clo_init)
Build and test

• Build:

  ./autogen.sh
  ./configure --prefix=`pwd`/inst
  make install

• Test:

  inst/bin/valgrind --tool=memtrace date
  – Should run ok, but produce no output

• So far, almost identical to none/nl_main.c
  – Now ready for proper tool-writing
Vex IR

- Intermediate representation (Vex IR)
  - *Vex* is the name of the JIT compiler sub-system
  - Short code blocks (**IRBB**)  
    - Represent roughly 3-50 instructions each
  - Arbitrary number of temporaries (intermediate values)  
    - A block’s *type environment* holds size of each temporary
  - Sequences of statements (with side-effects) (**IRStmt**)  
    - E.g. stores, register writes
  - Statements contain expression trees (no side-effects) (**IRExpr**)  
    - E.g. loads, arithmetic operations
    - E.g. a store’s address and value are both expressions
  - Each block ends in a jump

- All IR-related details are in *VEX/pub/libvex_ir.h*  
  - Included by *pub_tool_tooliface.h*, via *libvex.h*
mt_instrument (outer)

// include/pub_tool_basics.h provides types such as "Int".
Int i;

// Setup bbOut: allocate, initialize non-statement parts: type
// environment, block-ending jump's destination and kind.
IRBB* bbOut = emptyIRBB();
bbOut->tyenv = dopyIRTypeEnv(bbIn->tyenv);
bbOut->next = dopyIRExpr(bbIn->next);
bbOut->jumpkind = bbIn->jumpkind;

// Iterate through statements, copy to bbOut, instrumenting
// loads and stores along the way.
for (i = 0; i < bbIn->stmts_used; i++) {
  IRStmt* st = bbIn->stmts[i];
  if (!st) continue; // Ignore null statements
  // <Instrument loads and stores here (next 2 slides)>
  addStmtToIRBB(bbOut, st);
}
return bbOut;
mt_instrument (inner, 1/2)

switch (st->tag) {
    case Ist_Store: {
        // Pass to handle_store: bbOut, store address and store size.
        handle_store(bbOut, st->Ist.Store.addr,
            sizeofIRType(typeOfIRExpr(bbIn->tyenv, st->Ist.Store.data)));
        break;
    }
    case Ist_Tmp: { // A "Tmp" is an assignment to a temporary.
        // Expression trees are flattened here, so "Tmp" is the only
        // kind of statement a load may appear within.
        IRExpr* data = st->Ist.Tmp.data; // Expr on RHS of assignment
        if (data->tag == Iex_Load) { // Is it a load expression?
            // Pass handle_load bbOut plus the load address and size.
            handle_load(bbOut, data->Iex.Load.addr,
                sizeofIRType(data->Iex.Load.ty)); // Get load size from
            // type environment
        } 
        break;
    }
}

// <One more case (see next slide)>

mt_instrument (inner, 2/2)

• “Dirty” statements represent unusual instructions, e.g. cpuid, fxsave
  – Avoids encoding highly architecture-specific details in the IR
  – Tools can still see the register and memory accesses done by the instruction, and so do basic instrumentation

```c
  case Ist_Dirty: {
    IRDirty* d = st->Ist.Dirty.details;
    if (d->mFx == Ifx_Read || d->mFx == Ifx_Modify)
      handle_load(bbOut, d->mAddr, d->mSize);
    if (d->mFx == Ifx_Write || d->mFx == Ifx_Modify)
      handle_store(bbOut, d->mAddr, d->mSize);
    break;
  }
```
Adding calls to tracing functions

```c
static void add_call(IRBB* bb, IRExpr* dAddr, Int dSize,
                      Char* helperName, void* helperAddr)
{
    // Create argument vector with two IRExpr* arguments.
    IRExpr** argv = mkIRExprVec_2(dAddr, mkIRExpr_HWord(dSize));
    // Create call statement to function at "helperAddr".
    IRDirty* di = unsafeIRDirty_0_N(/*regparms*/2, helperName,
                                     VG_(fnptr_to_fnentry)(helperAddr), argv);
    addStmtToIRBB(bb, IRStmt_Dirty(di));
}
static void handle_load(IRBB* bb, IRExpr* dAddr, Int dSize) {
    add_call(bb, dAddr, dSize, "trace_load", trace_load);
}
static void handle_store(IRBB* bb, IRExpr* dAddr, Int dSize) {
    add_call(bb, dAddr, dSize, "trace_store", trace_store);
}
```

• (These functions must precede mt_instrument)
Run-time tracing functions

// VG_REGPARM(N): pass N (up to 3) arguments in registers on x86 -- // more efficient than via stack. Ignored on other architectures.
static VG_REGPARM(2) void trace_load(Addr addr, SizeT size)
{
    VG_(printf)("load : %08p, %d\n", addr, size);
}

static VG_REGPARM(2) void trace_store(Addr addr, SizeT size)
{
    VG_(printf)("store : %08p, %d\n", addr, size);
}

• (These functions must precede handle_load and handle_store)
• These functions called for every load and store at run-time
• VG_(printf) is Valgrind’s printf function
  – Valgrind does not use libc
  – VG_() is a macro that prefixes a longer string to the name
Improving accuracy and speed
Improving Memtrace’s accuracy

- Previous code treats “modify” instructions as a load + store
  - `addl %eax, (%ebx)` modifies `%ebx`

- Some instructions load/store multiple separate locations
  - `cmpsb` loads `%esi`, loads `%edi`
  - `pushl (%edx)` loads `%edx`, stores –4(%esp)
  - `movsw` loads `%esi`, stores `%edi`

- Collect load and store accesses for each instruction to identify memory access type, then instrument
  - `IMark` statements mark instruction boundaries in statement list
  - Modifies have a load and store to same address
  - Allows instruction reads to be traced as well
  - See `lackey/lk_main.c` for exactly this

- Could track loads/stores at system call boundaries
Improving Memtrace’s speed

• C calls are expensive
  – Save/restore caller-save registers around call
  – Setup arguments
  – Jump to function and back

• Can group C calls together
  – E.g. common pairs like load/load, load/store, store/store
  – ~1/2 as many C calls to trace functions
  – ~1/2 as many calls to VG_(printf)
Improving speed in general

- C calls are expensive
  - Combine when possible
  - Use inline code where possible
    - Especially for simple things like incrementing a counter
- Do work at instrumentation-time, not run-time
  - CacheGrind stores unchanging info about each instruction (instr. size, instr. addr, data size if a load/store) in a struct, passes struct pointer to simulation functions
    - Fewer arguments passed, shorter, faster code
- Do work in batches
  - Eg. Instruction counter: increment by N at start of block, rather than by 1 at every instruction
- Compress repetitive analysis data
More about tool-writing

- Vex IR is powerful but complex
  - We have only scratched the surface
  - All IR details are in VEX/pub/libvex_ir.h

- Tool-visible headers, one per module:
  - include/pub_tool_*\*.h
  - VEX/pub/libvex{,basictypes,ir}.h

- About 30 tool-visible modules:
  - Header files provide best documentation
  - coregrind/pub_core_<M>.h also helps explain things about module <M>

- Existing tools (especially Lackey) are best guides
Summary

• Have seen how to build a very simple tool
• Next: ideas for more ambitious tools
(end of talk 3)
4. More advanced tools

Nicholas Nethercote
This talk

• Some interesting kinds of advanced tools
  – Shadow location tools
  – Shadow value tools
    • Example: Redux, a dynamic dataflow graph tracer
    • Idea: Bandsaw, a memory bandwidth profiler
• What can you do with a Valgrind tool
Shadow location & value tools
Shadow location tools

- Tools that shadow every register and/or memory location with a metavalue that says something about it

- Examples:
  - Memcheck: addressability of memory bytes
  - Eraser: lock-sets held when memory bytes accessed
  - Or, simpler: count how many times the location has been accessed

- Each shadow location holds an approximation of the history of its corresponding location
Shadow value tools

• Tools that shadow every register and/or memory value with a metavalue that says something about it

• Examples:
  – Memcheck: definedness of values
  – TaintCheck: taintedness of values
  – Annelid: bounds of pointer values
  – Hobbes: run-time types of values

• Each shadow value is an approximation of the history of its corresponding value
A powerful facility?

• Shadowing every location or value is expensive and difficult, but doable
  – Valgrind provides unique built-in support for it
  – Memcheck’s slowdown factor is 10--60x

• What can you achieve by recording something about every location or value in a program?
  – Let us consider an illuminating example
  – Redux, a dynamic dataflow graph tracer
Two programs

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

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

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

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

Diagram showing two Directed Data-Flow Graphs (DDFGs) with nodes labeled with operations and values, illustrating the flow of data and control in a computational process.
DDFG Features

• Each node represents a constant, or value-producing operation:
  – Arithmetic/logic instructions (add, sub, and, or, ...)
  – Address computation instructions (lea)
  – System calls

• Doesn’t show other operations:
  – Copies (register/register, register/memory)
  – Function calls, returns
  – Branches

• Only shows:
  – System call nodes (external behaviour)
  – Parts of graph reachable from system call nodes (data flow)
  – Interesting computations only! *No book-keeping*
Hello world

- `fstat64` checks stdout
- `mmap` allocates an output buffer
- String length is counted
- `write` prints the string
- `munmap` frees the output buffer
- `_exit` terminates program
- 29,000 nodes built, 17 shown!
  - Most in dynamic linker
Essences

int faca(int n, int acc)
{
    if (n <= 1)
        return acc;
    else
        return faca(n-1, acc*n);
}

int main(void)
{
    return faca(5, 1);
}

• Accumulator recursion
• Algorithmically equivalent to iterative version
• Identical DDFGs
Stack machine version

- **Folded** top node reads program, converts ASCII characters to integers (1, 1, and 5)
- Same as C version, except -(X,1) vs. dec(X)
- Very different computation model
Haskell version

main =
putStrLn
  (show
   (facr 5 +
    faca 5 1))

• fac computations top right
Scaling difficulties

• bzip2’ing a two-byte file
  – dot: 8 seconds
  – ghostview: 5 seconds

• Scales terribly
  – CPU/memory use
  – Too big to view
Possible uses?

- Hmm, maybe:
  - Program visualisation
  - Debugging by sub-graph inspection
  - Dynamic slicing
  - Program comparison

- Really, grasping at straws
  - Too impractical as-is
So why talk about Redux?

• It is a good pedagogical tool
  – Explains dynamic binary analysis
  – Explains shadow value tools
  – Gets people thinking, generates ideas
    • “You can do anything” is too abstract
    • Makes the possibilities more concrete

• Shadow values are approximations of a value’s history
  – Redux shadow values show most of that history
Shadow value/location profilers

• All existing shadow value/location tools are error checkers
  – Except Redux

• Profiling shadow location tools?
  – Count how many times registers or memory locations accessed?

• Profiling shadow value tools?
  – Count how many times value has been copied?

• Something more interesting?
An idea: Bandsaw

- Show how data flows from place to place through memory
- Measure the amount of memory bandwidth used by each producer/consumer instruction pair

```c
line A: for (i = 0; i < 10*1000*1000; i++)
    a[i] = <...whatever...>
line B: for (i = 0; i < 10*1000*1000; i++)
    sum += a[i];
```

- 40 MB transferred from line A to line B
- Shadow locations
  - Each memory location shadowed with instr. addr of its producer
  - Upon a read, increment the producer/consumer pair count
- Useful? Don’t know… but shows what you can do
What can you do with a Valgrind tool?
Valgrind tools can…

- Delete, replace or augment every user-mode instruction
- Add analysis code inline, or as calls to C functions
- Wrap any system call
- Wrap any function
- Replace any function with a different one
- Observe or change any register or memory value
Instrumentation limitations

• Tools see Valgrind’s IR, not original instruction stream
  – Allows platform-independent instrumentation
  – Some information is lost
  – But instruction boundaries are preserved
• Virtual addresses
• Microarchitecture not directly visible (e.g. pipelines, μ-ops)
  – Can simulate to a point (e.g. caches, branch predictors)
Some underlying concepts

- **Profilers:**
  - Concepts: X happened N times, X happened near Y
    - CacheGrind, CallGrind, Massif

- **Checkers:**
  - Concept: X happened so Y should/should not happen
    - MemCheck, Helgrind, TaintCheck, Annelid, Daikon
  - Concept: X and Y were true at the same time, so…
    - Data race detectors (Eraser, DRD)

- **Visualizers:**
  - Concept: X fed into Y
    - Redux

- These concepts are common, but not the only ones
Brainstorming for new tools

- Power consumption profiling (Valgrind too high-level?)
- Floating point analysis/tracking
  - Loss of precision, underflows, NaN propagation
- Global domain-specific constraints
  - Pre/post-conditions, e.g. pthreads
  - Resource allocation/deallocation tracking
- Fault/event injection
- Data flow profiling to guide hardware compilation
- De-compilation/de-obfuscation tools
- Test suite generation
- Analyse crypto code as it runs to extract keys?
Tool design is difficult

- Need output that programmers can directly act on
- Efficiency of analysis code is crucial
- In checkers: getting the false positive rate down is hard
- Compilers generate really strange code
  - So do humans
- Inferring high-level info from low-level code is hard
  - E.g. is that a stack switch or large local array?
- Simple tools are boring!
  - The good tools are 1000s of lines of code, not 10s or 100s
  - Instrumentation (basic data extraction) is often only a small part
  - Good tools do clever things with the extracted data
  - Ability to write an instruction counter in only 5 lines is overrated
Take-home message
What do you want to know?

• What do you want to know about program execution that existing tools cannot tell you?
• Valgrind lets you build powerful program analysis tools
  – Can you learn what you want about programs using shadow locations or shadow values?
  – Or any other Valgrind-supported feature?
• The best tools do not arise in a vacuum
  – Good: “I wish I knew X about my program…”
  – Bad: “I want to write a tool. What would be a good one?”
• You are the people with the “I wish I knew X” ideas
  – Let your imaginations loose
  – Talk to the tool-makers
  – Maybe your idea is possible
Acknowledgments

• Valgrind developers: Julian Seward, Nicholas Nethercote, Tom Hughes, Jeremy Fitzhardinge, Robert Walsh, Josef Weidendorfer, Dirk Mueller, Paul Mackerras, Cerion Armour-Brown, and many others

• Other contributors: Donna Robinson, Alan Mycroft

• Tim Sherwood and IISWC organizers for the invitation
(end of talk 4)