Valgrind, Dynamic Binary Instrumentation and what you can do with it

March 28, EDF, Paris

Josef Weidendorfer

Chair for Computer Architecture (LRR)
TUM, Munich, Germany

Fakultät für Informatik
der Technischen Universität München
Informatik X: Rechnertechnik und Rechnerorganisation / Parallelrechnerarchitektur
Prof. Dr. Arndt Bode , Prof. Dr. Hans Michael Gerndt

My Background

- Chair for computer architecture at CS faculty, TUM
  - how to exploit current & future (HPC) systems (multicore, accelerators)
  - programming models, performance analysis tools, application tuning
- PhD on load balancing of commercial car crash code (MPI) 2003
- Interested especially in cache analysis and optimization
  - cache simulation: Callgrind (using Valgrind)
  - applied to 2D/3D stencil codes
  - recently extended to multicore (additional bottlenecks, new benefits)

What is Valgrind?

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 (most widely used tool), often call just “Valgrind”

Analysis Tools

Categorization 1: when does analysis occur?

- before run-time: static analysis
  - needs parsing (addition to compiler)
  - imprecise, but can be sound: sees all execution paths
- at run-time: dynamic analysis
  - needs instrumentation = inject analysis code into code
  - powerful, but unsound: sees one execution path

Valgrind performs dynamic analysis

Analysis Tools

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 to show source code)
  - lower-level information: e.g. registers, instructions

Valgrind performs binary analysis

What is Valgrind?

Valgrind: Dynamic binary analysis (DBA)

- analysis of machine code at run-time
- instrument original code with analysis code
  (= Dynamic Binary Instrumentation, DBI)
- track some extra information: metadata
- do some extra I/O, but don’t disturb execution

DBI frameworks:

- Pin, DynamoRIO, DIOTA, DynInst, etc.
- lots of overlap
- each system supports different platforms

Valgrind Basics

- dynamic binary instrumentation infrastructure
  - allows to build tools able to observe / modify the execution of programs
  - works at the binary level at runtime (no recompilation needed)

- supports Linux/AIX/OS-X on x86, x86-64, PPC32/64, ARM, MIPS

- correctness checking & profiling tools on top
  - “memcheck”: accessibility/validity of memory accesses
  - “helgrind” / ”drd”: race detection on multithreaded code
  - “cachegrind”/”callgrind”: cache & branch prediction simulation
  - “massif”: memory profiling

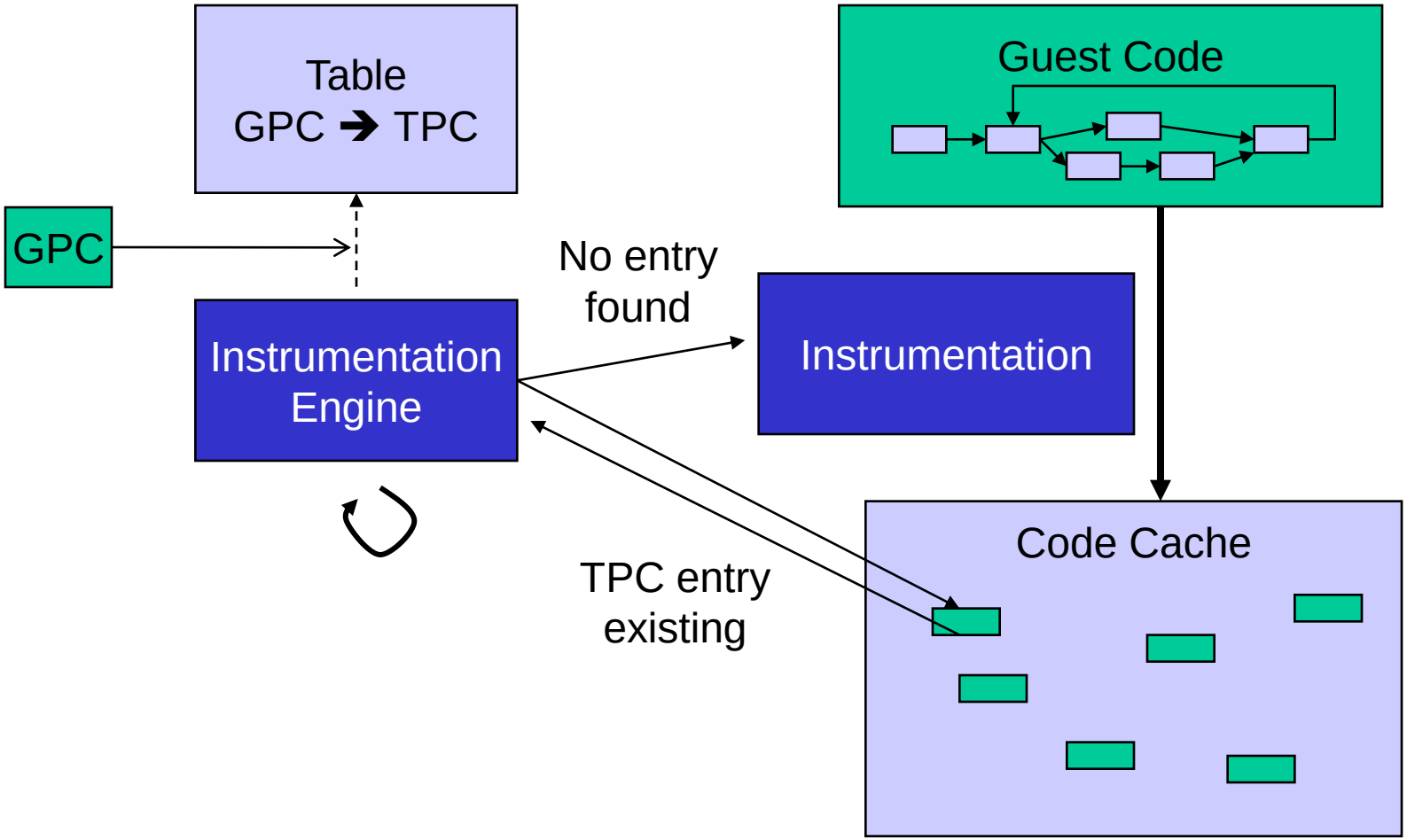
- Open source (GPL), www.valgrind.org, current version 3.9.0

How does Valgrind work?

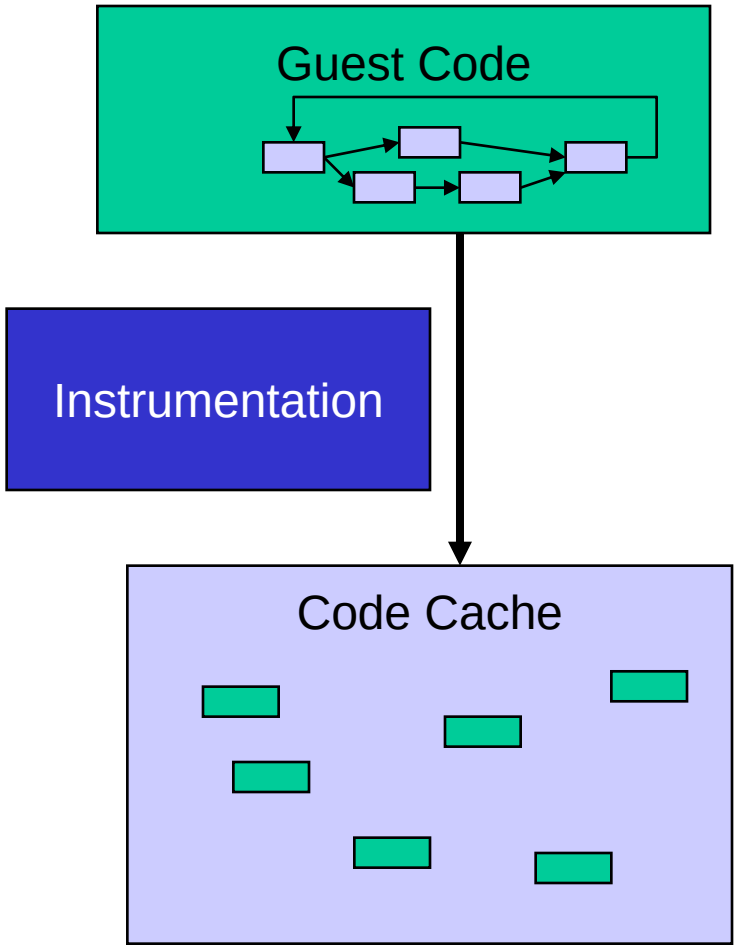
Dynamic Runtime Instrumentation

- full control of the execution of a program (“guest code”)
- loads guest into same process, but never run original code
- piece-wise controlled execution of instrumented original code
- granularity of instrumentation:
  dynamic basic block (BB): instruction sequence ending at branch
- loop in instrumentation engine:
  - for next guest PC (GPC) to execute: BB at GPC already translated?
  - if not: generate translated/instrumented version, store into code cache (may throw away old translations if cache is full)
  - run translated BB corresponding to GPC from code cache
  - translated BB returns next GPC to execute

Dynamic Runtime Instrumentation



Dynamic Runtime Instrumentation



ISA: x86, ARM, MIPS, …

↓

VG VEX Bytecode

↓

Tool Instrumentation Function

↓

VG VEX Bytecode

↓

ISA: x86, ARM, MIPS, …

How does it Valgrind work? (cont.)

Custom tools need to provide 3 functions:

- instrumentation function
  - input: list of VEX instructions from BB to instrument
  - output: list of VEX instructions of instrumented version
  - instructions are abstracted from ISAs to allow platform independence (this V-ISA/bytecode is called “VEX”)
- initialization:
  called at the very beginning to allocate own memory
- finalization:
  called at program termination

How does Valgrind work? (cont.)

Typical instrumentation: loop over instructions

- add additional instructions depending on instruction type
- also needs to add the original instruction into output sequence (this allows modification)
- added instructions may be CALLs into own code

```
// 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];
// Add your own instrumentation
addStmtToIRBB(bbOut, st);
}
return bbOut;
```

Example Instrumentation Function

```
switch (st->tag) {
 case Ist_Store: {
   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
   }
   break;
   ...
}
```

Example Instrumentation Function

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

```
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;
}
```

How does Valgrind work? (cont.)

With dynamic instrumentations, 3 types of times can be distinguished

- compilation time: no runtime information known
  - original code, generated by comiler
- instrumentation time: usually happening once, directly before exec.
  - partial runtime information known, may be used for optimizations
  - code shown on previous slides
- execution time
  - code run when instrumented code is executing itself

Example: Still Instrumentation Time

```
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);
}
```

Example: Functions called at Execution Time

```
/ 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);
}
```

Starting Up

- Valgrind
  - loads the core, chosen tool and guest 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
  - 3x times rewritten!

Complications

- System calls
  - Valgrind does not trace into kernel
  - calls to custom before/after handlers can be requested
  - Valgrind knows about input/output of all Linux syscalls
- Signals
  - interception at registration/delivery needed
- Threads
  - get serialized by Valgrind

Client Requests

- Trap-door mechanism
  - An unusual no-op instruction sequence
  - Under Valgrind, it transfers control to core/tool
  - guest 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 macro
  - Tools only need to include a header file to use them
  - They do nothing when running natively

More Complex Tools

- Slowdown
  - naïve way: easily > 100x
  - optimization: merge/spezialize at instrumentation time
- meta information
  - VG provides “shadow memory”
  - arbitrary meta information for every original memory cell
    - memcheck: valid / initialized?

Pro & Contra (i.e. Simulation vs. Real Measurement)

Usage of Valgrind

- – driven only by user-level instructions of one process
- – slowdown (call-graph tracing: 15-20x, + cache simulation: 40-60x)
  - “fast-forward mode”: 2-3x
- ✓ allows detailed (mostly reproducable) observation
- ✓ does not need root access / can not crash machine

Example Callgrind: Simple Cache model

- – “not reality”: synchronous 2-level inclusive cache hierarchy (size/associativity taken from real machine, always including LLC)
- ✓ easy to understand / reconstruct for user
- ✓ reproducible results independent on real machine load
- ✓ derived optimizations applicable for most architectures

Questions?

Backup

Numerical Stability of Algorithms

- Setting/Forcing FP Rounding Modes in VEX

```
/* Variants of the above which produce a 64-bit result but which
        round their result to a IEEE float range first. */
/* :: IRRoundingMode(I32) x F64 x F64 -> F64 */
      Iop_AddF64r32, Iop_SubF64r32, Iop_MulF64r32, Iop_DivF64r32,
```

- Catch FP Ops in own instrumentation function
  - replace rounding mode by wished one

- Unfortunately, this currently works only with ARM/PPC
- x86 is missing correct handling in
  - frontend: rounding not correctly set (easy)
  - backend: generate optimized x86 code to set requested mode

KCachegrind: Usage

```
{k,q}cachegrind callgrind.out.<pid>
```

- left: “Dockables”
  - list of function groups groups according to
    - library (ELF object)
    - source
    - class (C++)
  - list of functions with
    - inclusive
    - exclusive costs
- right: visualization panes