



Porting Valgrind to Linux on Itanium

Julian Seward

`julian@valgrind.org`

<http://www.valgrind.org>



What is Valgrind?

A framework for building
simulation-based debugging/profiling tools
and
A standard set of tools

Tools built with Valgrind:

- 2 memory checkers (Memcheck, Helgrind)
- 3 profilers (Cachegrind, Massif, Calltree)
- Various other experimental tools
- Runs on {x86,amd64,ppc32,ppc64}-linux

- Complete coverage, even without source code, for C/C++/Fortran
- Robust, scalable (25Mloc apps known to run)
- Open source, GNU GPL v2



Standard tools

- **Memcheck – memory error detector**
 - Find invalid memory accesses, use of uninitialised data, and leaks
- **Cachegrind - low level time profiler**
 - Simulates an I1/D1/L2 hierarchy, counting insns, mem refs and misses
 - Shows results at program, function, line or insn level
- **Massif - space profiler**
 - Find out who malloc'd what, why, when, where, for how long
- **Callgrind/KCachegrind**
 - Extended version of Cachegrind, with caller/callee cost attribution
 - Impressive GUI for navigating profiling results
- **Helgrind** (currently broken)
 - Data race detector for threaded apps
 - Finds shared data accessed without adequate locking



Memcheck: a memory-error detector

Addressing errors: read/write freed mem, block overruns

```
char* a = malloc(100); free(a); .. a[50] .. // read freed
char* a = malloc(100); x = a[100];          // read off end
```

Invalid use of malloc/free, leaks, including leaked cycles

```
Bar* a = new Bar[100]; delete a;           // new[]/delete mismatch
char* a = malloc(100); a = malloc(10);    // leak
char* a = &local[0]; free(a);             // invalid free
```

Definedness errors: use of undefined values (incl single bits)

```
int a[10];                                // undefined
a[5] |= 4;                                 // set one bit
if (a[5] & 4) ...                          // no error reported
if (a[5] & 8) ...                          // error reported
write(fd, a, 10)...                       // error: passing garbage to the kernel
```

We know of no other system capable of this

Performance overhead: in the region of 30x slowdown



About the project

Chronology

- 1999-2002 Preliminary exploration
- August 2002 First public release (1.0.0)
- 2002-2005 Consolidation on x86-linux
- 2005-now Maturity (3.X series):
stability, structure, maintainability, portability

Infrastructure

- Public Subversion server
- Mailing lists, bug tracker
- Regression test suite, automated overnight build & email
- Performance analysis suite

Users

- OpenOffice.org Mozilla Opera Qt KDE GNOME AbiWord Evolution MySQL PostgreSQL Perl PHP Mono Samba GIMP OggVorbis UnrealTournament and hundreds of others (see <http://www.valgrind.org/gallery/users.html>)



Valgrind = Core + Tools

Building simulation-based tools is hard

- Instrumentation code is often straightforward
- Getting it into a program is difficult

Core provides common infrastructure (~180 k loc)

- CPU virtualisation, threads, syscalls, signals, debug info
- Dynamic-translation based: no relink/recompile
- Complete coverage: all libraries and languages

Tools become relatively simple (eg 1-10 k loc)

- Allowed to instrument architecture-neutral code representation
- Allowed to see significant events (malloc, free, syscall start/finish, thread state changes, calls, returns)
- No obligation to do any of these things



What it does

With no instrumentation (--tool=none)

- Cohabits process/address space with client program
- Simulates CPU
- Intercepts syscalls, hands to kernel
- Catches signals, hands to client
- Manages address space
- Sequentialises threads

With instrumentation (any other tool)

- Allows tool to instrument code
- Allows tool to shadow memory and registers
- Manages debug info
- Manages errors



Porting to ia64-linux: system aspects

Mostly harmless

- Program startup -- ELF
- Address space management -- kernel-specific
- Signals/syscalls -- kernel-specific (mostly)
- Debug info -- Dwarf2/3 (stabs?)

Less straightforward

- Dispatcher -- handwritten assembly
- ABI considerations can be a problem (eg ppc64-linux)



Porting to ia64-linux: CPU aspects

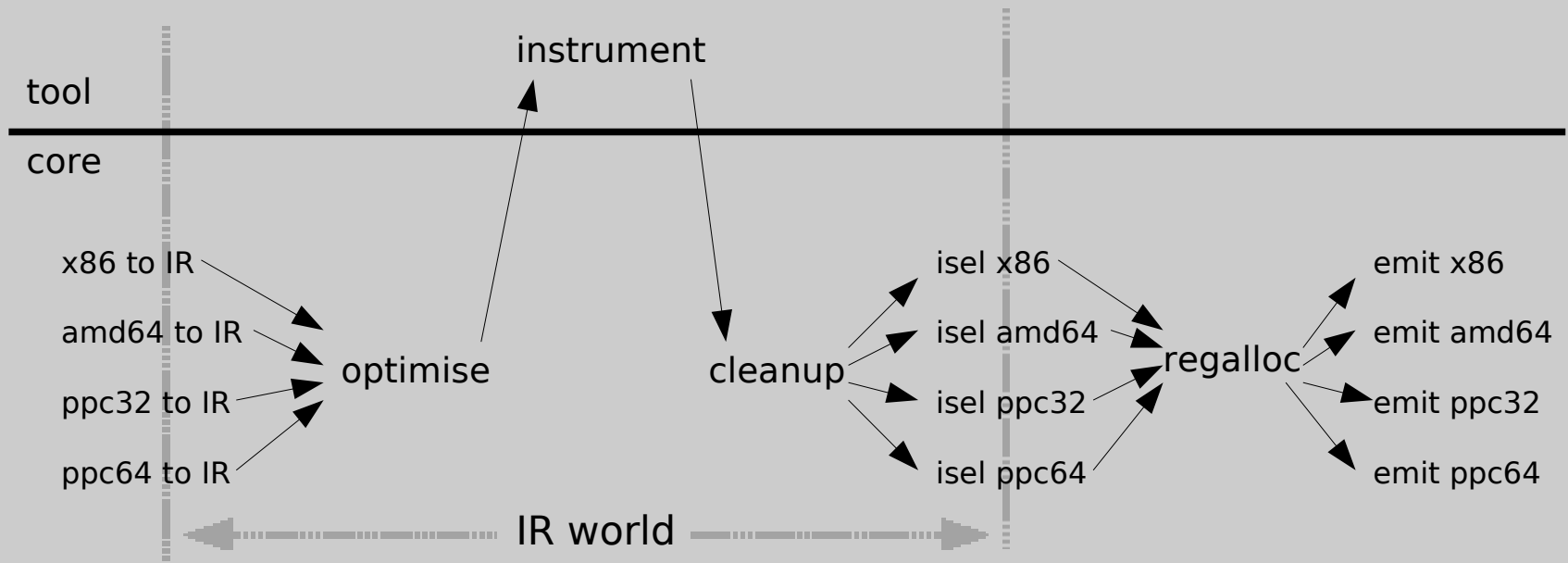
Design goals

- Low overhead instruction set simulation (user space only)
- Comprehensive and correct
- Exposes most user space visible changes to architected state
- Instrumentable, to the level of shadowing registers and memory
- Insulates tools from the underlying instruction and register set

Dynamic-translation based system

- Unit of translation is extended basic block
- Translate all instruction sets via a common intermediate representation (IR)
- Translations cached for reuse; about 400k stored

Translation pipeline



- Optimise: Get-Put forwarding, const fold/prop, CSE, dead code, unrolling
- Isel: build trees, then top-down maximal-munch matching
- RegAlloc: Linear scan, live range splitting, move coalescing



The Intermediate Representation, IR

Encodes everything: integer, FP, SIMD, condition codes

- Simple SSA-based, typed, intermediate representation

What's in it

- Top level: straight line sequence of IR statements, with side exits
- Statements: assign temporary, write guest reg, stores, helper call, conditional exit
- IR Expr: load, read guest reg, helper call, const, temp, 1/2/3/4-ary op, “: ? exprs”
- Rich set of primitive ops, including SIMD
- Calls to helper functions (both pure and impure)
- Expressions can be flat or nested

What isn't in it

- if-then-else
- loops



IR Example

Initial IR

```
0x400388A: movl 0x21C(%ebx),%eax
```

```
----- IMark(0x400388A, 6) -----  
t0 = Add32(GET:I32(12),0x21C:I32)  
PUT(0) = LDle:I32(t0)
```

```
0x4003890: testl %eax,%eax
```

```
----- IMark(0x4003890, 2) -----  
PUT(60) = 0x4003890:I32  
t3 = GET:I32(0)  
t2 = GET:I32(0)  
t1 = And32(t3,t2)  
PUT(32) = 0xF:I32  
PUT(36) = t1  
PUT(40) = 0x0:I32  
PUT(44) = 0x0:I32
```

```
0x4003892: jz-8 0x4003897
```

```
----- IMark(0x4003892, 2) -----  
PUT(60) = 0x4003892:I32  
if (32to1(x86g_calculate_condition  
    (0x4:I32,GET:I32(32),GET:I32(36),  
    GET:I32(40),GET:I32(44)):I32))  
    goto {Boring} 0x4003897:I32  
goto {Boring} 0x4003894:I32
```

After IR opt and tree building

```
----- IMark(0x400388A, 6) -----  
t6 = LDle:I32(Add32(GET:I32(12),0x21C:I32))  
PUT(0) = t6
```

```
----- IMark(0x4003890, 2) -----  
PUT(32) = 0xF:I32  
PUT(36) = t6  
PUT(40) = 0x0:I32  
PUT(44) = 0x0:I32
```

```
----- IMark(0x4003892, 2) -----  
PUT(60) = 0x4003892:I32  
if (32to1(1Uto32(CmpEQ32(t6,0x0:I32))))  
    goto {Boring} 0x4003897:I32  
goto {Boring} 0x4003894:I32
```



Itanium-specifics in the IR

Rotating registers

- Mechanism (GetI, PutI) already in place for x87
- Extend to deal with Itanium rotating sections
- Extend associated IR transformations

Register stack engine

- Need way to indicate block load/store of registers, so tools can track data movement
- Needs to be done conditionally

ALAT

- Simulate a null (always-invalidate) ALAT ? Problematic
- Simulate a simple direct-mapped ALAT ?
- Open question



Itanium-specifics in the IR (2)

NaTs / NaTVals

- Treat NaT bits as a separate register bank
- When doing reg-reg opts, compute resulting NaT bits explicitly
- Explicitly represent propagation of deferred exception tokens
- Need way to do if-then-else (lazily)

Control speculation

- Need a new IR statement kind: speculative load (gives value/token & NaT bit)
- Need way to raise NaT consumption fault on guest

Predication

- Need way to do if-then-else (lazily)
- Need a bunch of IR folding/merging transformations



Performance consequences

Will be slower than existing ports, due to:

- Rotating register overhead
- Tracking of NaT bits
- Simulating propagation of deferred exception tokens
- Simulating the ALAT
- Insn packing/scheduling in backend

Longer BBs will help ..

- .. but raises complication level still further (profile-guided trace selection)



CPU aspects: overall consequences

Redesign IR to support if-then-else

- Redo IR optimiser (now have to do phi-node maintenance)
- Redo register allocator (ditto)
- Redo instruction selectors for all existing targets
- Redo existing tools to handle new IR

Open questions

- How to simulate ALAT
- Tool operation for speculative loads
- Performance

Other risks

- Insn packer in backend? Need scheduling?
- Simulation of atomic compare/swap insns
- Abl complications (for function intercept/wrap)
- Calls/returns very slow?



Other things that would help

Access to a machine

- With relevant compilers
- Both for development and ongoing maintenance
- "If it wasn't auto-tested last night, it's broken" => run nightly builds forever

Access to a guru

- Person with excellent understanding of Itanium code
- .. and of kernel/glibc/ABI/debug info aspects

Access to willing victims

- People to try out builds, report failures, repeat until it works



Questions ?

