VEX: Where next for Valgrind's dynamic instrumentation infrastructure?

Julian Seward, jseward@acm.org

4 February 2017. FOSDEM. Brussels.

Overview

How it works (roughly)

Problems: register use

Problems: speculation

Proposal for a new framework

How it works 1

Top level loop

(1) Instrumented code runs in code cache

(2) Program jumps to uninstrumented address

(3) Leave code cache

(4) Invoke compiler (VEX) on missing address

(5) Instrumented code added to code cache

(6) Goto 1

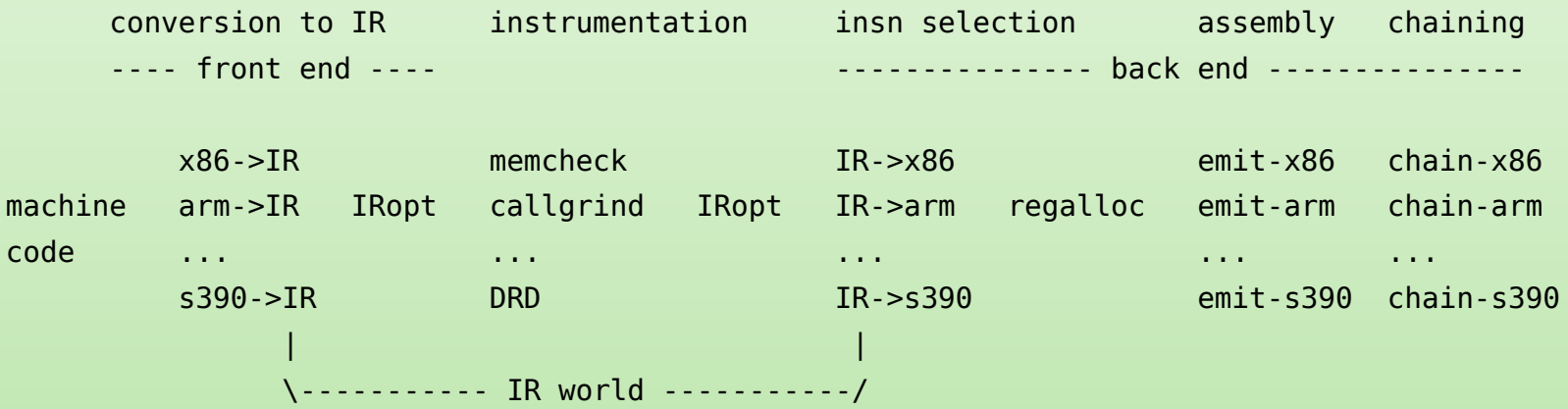
There's a compiler ..

.. and a run-time system.

How it works 2

VEX, a simple extended-basic-block compiler

- Based on a simple intermediate representation (IR)
- Machine code --> IR --> Instrumented IR --> machine code
- Starting at specified insn, up to next branch
- Each insn individually translated
- Optimised over the whole block
- Clean semantics counts!



How it works 3

Prelims:

- IR: simple single-assignment language for straight-line code
  - Loads, stores, assignment to IR temporaries, arithmetic
  - GET and PUT to model register access
  - Side exits (conditional)

- Guest State = struct holding simulated register values
  - GET and PUT reference offsets in it
  - Dedicate a host register to point at it
  - Is not a 1:1 mapping with the architected state

```
struct {
   ..
   UInt guest_R0;
   UInt guest_R1;
   ..
} VexGuestARMState;
```

Running example 1

```
ARM32 guest code    Initial IR                Optimised IR

add r1, r2, r3      add r1, r2, r3            add r1, r2, r3
ldr r4, [r1]           t10 = GET(8)              t10 = GET(8)
mov r1, #27            t11 = GET(12)             t11 = GET(12)
                       t12 = Add32(t10, t11)     t12 = Add32(t10, t11)
                       PUT(4) = t12

                    ldr r4, [r1]              ldr r4, [r1]
                        t13 = GET(4)
                        t14 = LOAD(t13)           t14 = LOAD(t12)
                        PUT(t16) = t14            PUT(t16) = t14

                    mov r1, #27               mov r1, #27
                        PUT(4) = 27               PUT(4) = 27
```

Running example 2

Just for fun .. let's generate x86 code from the IR.

```
ebp --> VexGuestARMState

Optimised IR                  Host code

add r1, r2, r3
    t10 = GET(8)              movl 8(ebp), eax      | READING
    t11 = GET(12)             movl 12(ebp), ebx     |    from guest state
    t12 = Add32(t10, t11)     leal (eax, ebx), ecx

ldr r4, [r1]
    t14 = LOAD(t12)           movl (ecx), edx
    PUT(t16) = t14            movl edx, 16(ebp)     | WRITING
                                                    |    to guest state
mov r1, #27                                         |
    PUT(4) = 27               movl $27, 4(ebp)      |
```

What's good? We cached a guest register (R1) in a host register (ECX) for the block

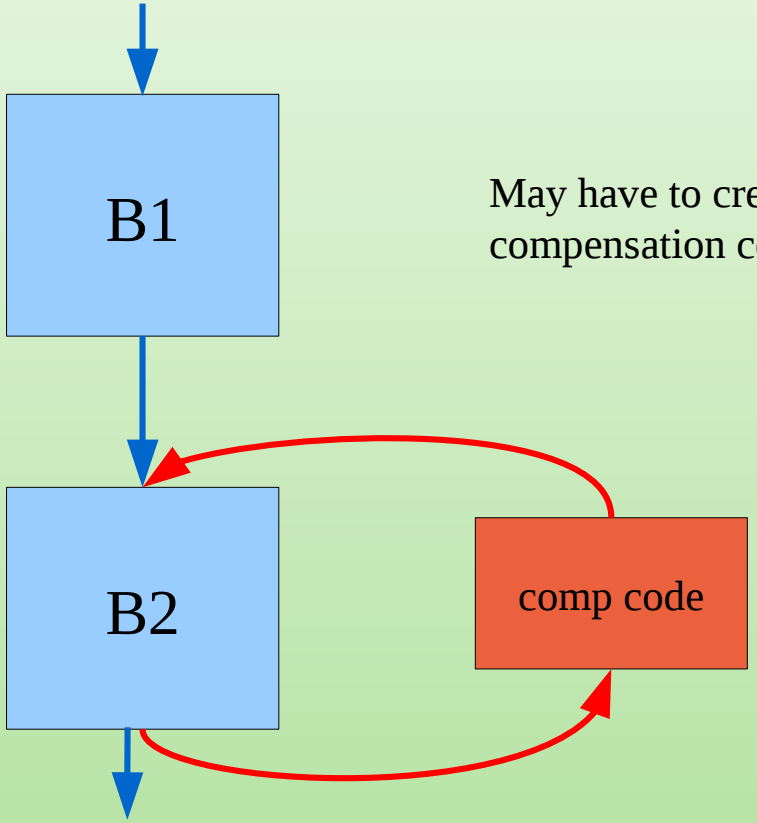
What's bad? Host registers aren't live between blocks

=> lots of memory traffic

Running example 3

Better: cache some guest regs in host regs across boundaries

- Implies compensation code between blocks
- Up to 3 times as many guest regs as host regs
- Not easy to decide on a mapping



May have to create
compensation code in the “wrong” order

Precise Exceptions 1

The precise exception problem:

```
add r1, r2, r3
   t10 = GET(8)
   t11 = GET(12)
   t12 = Add32(t10, t11)
                  <---- guest_R1 is not up to date
ldr r4, [r1]
   t14 = LOAD(t12)
   PUT(t16) = t14

mov r1, #27
   PUT(4) = 27
```

If the load faults, we don't have consistent guest state to resume with

Mostly doesn't matter ... except when it does

Precise Exceptions 2

PX fixes:

- Don't do redundant-PUT removal (current kludge)

- Store metadata that shows where every (architected) value is
- Don't optimise away any architected value
- Be very careful about effects sequencing in initial translation

- Program counter is a special and important case
  - compute it from the host PC
  - portable: no!
  - how do we recover host PC in helper calls?

    `__builtin_return_address()` ? Urrrrrk

Rearchitecting the framework 1

More performance means:

- (better use of host registers)
- Optimising over larger pieces of code
- Enabling proper if-then-else and speculation

Larger bits of code?

- VEX follows uncond branches and calls to known destinations
- Pretty feeble -- avg block size ~ 10 guest insns

Why does this help?

- Remove more dead register updates, especially cond codes
- More opportunities for folding, CSE
- Amortises block-to-block costs better

Is not something the JIT can do by itself. Requires RTS support.

Rearchitecting the framework 2

Do the “standard” thing: profiling

- Profile-guided trace selection
  - cold block cache, short blocks, profiling
  - assemble “hot path” and reoptimise
  - various trace selection algorithms, eg NET (Next Executing Tail)

- Or ...
  - No fixed distinction between hot and cold blocks
  - All blocks have counters for the exit branch(es)
  - when tail counts get high enough, extend, regenerate

Can reduce JIT overheads, too

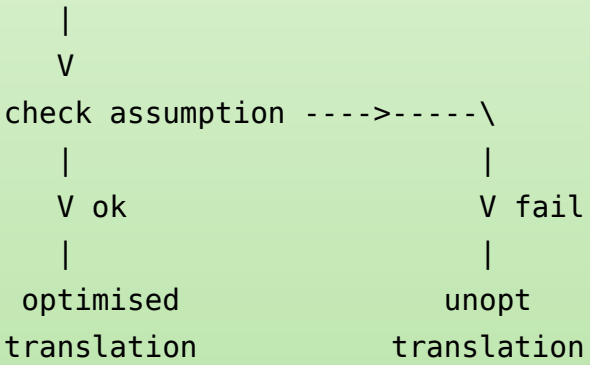
- Avoids optimising cold blocks
- Move recompilation into a helper thread, keep going

Rearchitecting the framework 3

Do another “standard” thing: speculation

- Translate/optimise trace with some assumption
- Check at start of trace. If failed, leave and run a less optimised version.
- eg
  - These two memory addresses are in the same page
  - Loaded/stored data is completely defined
  - Load/store addresses are defined
  - x87 FP register stack will not overflow

Traditionally:



Bad:

- Multiple translations
- Can't rejoin trace later
- icache space very limited

Rearchitecting the framework 4

Proposal: IR with nested control-flow diamonds

IR is a sequence of statements:

- PUT/GET
- Arithmetic
- Exit (now unconditional)

but also

```
if (cond) { statements } else { statements }  # with hint
```

Gives flexibility:

- Speculate but stay on trace:

```
if (cond) { fast-case } else { slow-case }  # hint = likely true
```

- Speculate and leave trace:

```
if (cond) { fast-case } else { Exit }  # hint = likely true
```

Rearchitecting the framework 5

Clean semantics is important! It got VEX where it is today ..

eg we can sink code into slow paths:

```
   X // only relevant for cold case
   if (cond) { hot } else { cold }
=>
   if (cond) { hot } else { X; cold }
```

eg we can merge duplicate tests, clone, specialise

```
   if (cond) { hot1 } else { cold1 }
   X
   if (cond) { hot2 } else { cold2 }
=>
   if (cond) { hot1; X; hot2 } else { cold1; X; cold2 }
```

Gives much more flexibility with instrumentation code

Trivial to decide when such a transformation is valid

(is it worthwhile? Ha! that's a much harder question :-)

Rearchitecting the framework 6

What would this entail?

- A step closer to real SSA:
  - Introduction of phi-nodes (~ control flow merges) in the IR
  - But no dominance frontiers (yay!)
- Reimplement IR optimiser to deal with phi nodes
- Redo all instruction selectors similarly
- Redo register allocator similarly
- Have assemblers that group hot host-code blocks together

Note!

- Is independent of profile-guided trace selection improvements
- But will benefit from longer traces
- Could be implemented independently

So, in conclusion ..

We saw ..

.. some stuff about how VEX works
.. some “low level” problems with registers, and possible fixes
.. a proposal for a new framework with more performance headroom

We didn't ..

.. consider whether any of this is worthwhile (sounds FUN though)
.. consider how it might get done

Thank you for listening!

Questions?