Inlining of
Memcheck helper function fast paths

Julian Seward, jseward@acm.org

Memcheck

Is a memory access checker:

• checks memory access at byte granularity
• checks definedness at bit granularity
• uses a combination of in-line and out-of-line code

Basic data structure: array of 2-bit values for each byte in address space

• enum { NOACCESS, UNDEFINED, DEFINED, PARTDEFINED }  
• For PARTDEFINED, have an auxiliary table. Seldom used.
• Naive implementation even for 32-bit target infeasible ...
• ... array would require 1 GB
Two-level map scheme

For a 32-bit address space:
- Divide address space into 64KB chunks -- Secondary Maps
- Have a 64K entry Primary Map
Two-level map scheme

For a 32-bit address space:

- Divide address space into 64KB chunks -- Secondary Maps
- Have a 64K entry Primary Map

- Distinguished secondary map makes reads faster
- No need for a NULL check
32-bit load fast path

Goal: given an address
  • check we can read all 4 bytes, report errors if not
  • get the 32 definedness bits for the address

Optimise for common case
  • address is 4-aligned
  • location is accessible
  • location contains defined data

Actions
  • alignment check: check addr is of form X--(30)--X00 (test, branch)
  • read pri map: sm = pri_map[addr >> 16] (shift, load)
  • read sec map: vbits8 = sm[(addr >> 2) & 0x3FFF] (shift, and, load)
  • check defined: check vbits8 == 0xAA (cmp, branch)

Total cost
  • 2 loads
  • 2 conditional branches, totally predictable
  • 5 shots on the ALU
Fast paths, as currently integrated

- JIT generated code
- gcc generated code

This is crazy! (but at least it's simple :-) )

Call/return overheads are larger than the fast path cost (at least, superficially ..)
- caller: spill caller-save regs before call ..
- caller: .. and restore afterwards
- caller: shuffle args into arg regs, and out of result regs
- callee: save regs in prologue ..
- callee: .. and restore in epilogue

… can be terrible if gcc is having a bad day
So, what to do?

What we want
- fast paths in-line
- precise control of fast path insns (that includes “No Spilling Please”)
- ... but architecture neutral
- no massive code bloat (icache misses, and JIT slowdowns)

The obvious answer ...
- generalise existing basic-block-at-a-time JIT
- to add arbitrary control flow

... implies years of work ...
- rewrite entire JIT
- add CFG, dominance frontiers, phi nodes, new IR optimiser, new reg allocator

... is a losing proposition.
Plan B: Cheat.

- Keep existing basic-block-at-a-time JIT as-is
- Replace helper calls by machine-code templates
- Template is a single “big instruction”
  - travels through the JIT pipeline unchanged
  - reg alloc treats it like any other insn: gives it in/out/scratch regs
  - when it finally arrives at the assembler, we finally have to Do Something
    - instantiate the template
  - That's pretty much all
32-bit load template

NCode [r1] [a1] [s1] {
    hot:
0  test.w  a1, $MASK
1  bne    cold.4
2  shr.w  s1, a1, $16
3  ld.w   s1, [$PRIMARY_MAP + s1 << 2]   // SM
4  and.w r1, a1, $0xFFFF              // SMoff
5  shr.w r1, r1, $2                   // SMoff
6  ldub.w r1, [s1 + r1]                // AVbits
7  cmp.w  r1, $0xa9
6  bne    cold.0
9  imm    r1, $0
10  nop

cold:
0:  mov.w  s1, r1   // AVBits
1:  imm.w  r1, $0xffffffff
2:  cmp.w  s1, $055
3:  beq    hot.10
4:  call   [r1] = LOADV32le_SLOW [a1]
5:  b      hot.10
}

- Single-entry single-exit, but split into hot/cold code
- Stylised 3-address code
- Template registers: R(result), A(argument), S(scratch)
After register allocation

NCode [r1] [al] [s1] {
  hot:
  0  test.w  a1, $MASK
  1  bne    cold.4
  2  shr.w  s1, a1, $16
  3  ld.w   s1, [$$PRIMARY_MAP + s1 << 2] // SM
  4  and.w  r1, a1, $0xFFFF         // SMOff
  5  shr.w  r1, r1, $2             // SMOff
  6  ldub.w r1, [s1 + r1]         // AVbits
  7  cmp.w  r1, $0xaa
  8  bne    cold.0
  9  imm    r1, $0
 10  nop

cold:
  0: mov.w  s1, r1 // AVBits
  1: imm.w  r1, $0xffffffff
  2: cmp.w  s1, $0x55
  3: beq    hot.10
  4: call    [r1] = LOADV32le_SLOW [al]
  5: b       hot.10
}

• we have (eg)   r1 = %edi   a1 = %ebx   s1 = %esi
• and (eg)       live-after = {%eax, %ebx, %ecx, %edi, %xmm1, %xmm4}
• use live-after to calculate spill-sets around the call
After generating native code

NCode [%edi] [%ebx] [%esi] { // [r1] [a1] [s1]
    hot:
        0  test  $MASK, %ebx  // test.w a1, $MASK
        1  jnz  cold.4  // bne cold.4
        2  movl  %ebx,%esi; shrl $16, %esi  // shr.w s1, a1, $16
        3  movl  $PRI_MAP(,%esi,2), %esi  // ld.w s1, [$PRI_MAP + s1 << 2]
        4  movzw %ebx, %edi  // and.w r1, a1, $0xFFFF
        5  shrl  $2, %edi  // shr.w r1, r1, $2
        6  movzbl (%esi,%edi), %edi  // ldub.w r1, [s1 + r1]
        7  cmp  $0xaa, %edi  // cmp.w r1, $0xaa
        8  jnz  cold.0  // bne cold.0
        9  movl  $0, %edi  // imm r1, $0
        10  (no-code)  // nop
    cold:
        0: movl  %edi, %esi  // mov.w s1, r1
        1: movl  $0xffffffff, %edi  // imm.w r1, $0xffffffff
        2: cmpl  $0x55, %esi  // cmp.w s1, $0x55
        3: jz   hot.10  // beq hot.10
        4: save-some-regs; movl %ebx,%eax; call LOADV32le_SLOW;
           movl %eax,%edx; restore-some-regs
           // call [r1] = LOADV32le_SLOW [a1]
        5: jmp  hot.10  // b hot.10
    }

• hot.2: 3-vs-2 address bites us, but only once
• cold.4: call overheads still present, but confined to cold path only
Instantiation summary

Instantiator's duties
- reg-alloc specifies a real register for each template register
- generate native code, using that mapping
- reg-alloc also gives live-after set
- use this to spill around C calls
- .. so don't put C calls on the hot path

What the generic JIT framework does for us
- concatenates all hot sections and all cold sections
- .. so the “main trace” for entire instrumented basic block is straight-line code
- .. conforming to “forward-branches-not-taken” rule
- performs relocations for jumps

What's the per-architecture burden?
- instantiator -- map to native insns
- calls -- need to spill/restore around call
- calls -- need to marshal arg and result values
Challenges

- Template must be architecture neutral.
  Yet generate good code
  Tricky, for: x86, amd64, s390x, ppc32, ppc64, mips32, mips64, arm32, arm64
  Unavoidable kludging for 64-bit loads/stores on 32 bit targets

- Verifying that templates are correct
  A serious worry

- 2-addr or 3-addr in the templates?
  `shr.w r1, a1, $2` is not 1 insn on x86. Must generate `mov; shr`
  Strategy: keep 3-addr in templates
  So as not to disadvantage 3-address archs (eg arm32)

- Avoiding JIT slowdowns
  We're generating 50%-100% more code

- Making sense of observed performance changes
  Are we trashing the icache?
Current status

Status:

- amd64 (x86_64) proof of concept up and running
- templates for 32- and 64-bit loads only
- generates hot section code identical to gcc-4.9.2
- 0% to 14% perf improvement (perf/tinycc.c)
- approx 50% code bloat (15:1 --> 22:1)

- svn://svn.valgrind.org/valgrind/branches/NCODE
- entire JIT pipeline and Memcheck almost unchanged
- (a few hundred lines of diff)
- amd64 template expander is < 1000 lines
What next?

Wrap up initial amd64 work
- add templates for 16- and 8-bit loads
- see if I can hit 20% perf improvement on Haswell
- measure hot vs cold code sizes, and I/D cache effects

Verify sanity on a second arch: arm32
- Implement arm32 template expander
- It's important that arm32 works well

Improve testing of Memcheck shadow memory
- this hackery is a correctness hazard

Tidy up, implement all archs

Testers, hackers, experimenters welcome!

Questions?