OpenJDK on ARM: *Quo vadis?*

Andrew Haley

redhat

Where are we?

- Several VMs are supported on ARM: HotSpot Zero, JamVM, cacao, etc
- None of them has a full-blown server JIT
- I’m mostly going to talk about Ed Nevill’s ARM Port Without A Name

redhat

Ed’s port

- is based on the C++ interpreter
- uses the same data structures as Gary Benson’s Zero
- is in two parts: a bytecode interpreter written in ARM assembly language and a JIT written in C++ that generates Thumb 2™ code

  This means that the JIT requires a fairly recent ARM, but the assembler interpreter will run on old ARM 5s (ARM 4s? I don't know. Probably)

redhat

Ed's port

- was written a couple of years ago, on contract from ARM Ltd
- Once the contract ended it wasn't maintained, so it had to be deleted
- Many changes were needed for HotSpot 20, the hard-fp ABI, and multi-core ARMs (which didn't exist when Ed did the work)
- Most OpenJDK maintainers seemed to believe it was "unmaintainable", which I interpreted as a challenge
- ... I wanted to prove my point!

redhat

- The Thumb 2™ JIT is very simple and fast, but the code can be of surprisingly high quality:

```
   54 : 99 00 30       ifeq
0x40736738:     ldr.w   r3, [r4], #4
0x4073673c:     cmp     r3, #0
0x4073673e:     beq.w   0x40736878
   57 : 15 07          iload
   59 : 04             iconst_1
   60 : 60             iadd
   61 : 36 08          istore
0x40736742:     adds.w  sl, r5, #1
----- Basic Block -----
   63 : 15 08          iload
   65 : 15 04          iload
0x40736746:     ldr     r3, [r4, #100]  ; 0x64
   67 : a2 00 14       if_icmpge
0x40736748:     cmp.w   sl, r3
0x4073674c:     bge.w   0x4073679c
   70 : 19 06          aload
0x40736750:     ldr     r3, [r4, #92]   ; 0x5c
   72 : 15 08          iload
   74 : 04             iconst_1
   75 : 64             isub
0x40736752:     subs.w  r2, sl, #1
   76 : 2d             aload_3
   77 : 15 08          iload
   79 : 32             aaload
0x40736756:     ldr     r0, [r7, #8]
0x40736758:     cmp.w   sl, r0
0x4073675c:     it      cs
0x4073675e:     blcs    0x40731196
0x40736762:     adds.w  r0, r7, sl, lsl #2
0x40736766:     ldr     r1, [r0, #12]
```

redhat

Ed's port

A C++ interpreter stack frame looks like this:

```
 0x639fe9a4: stack_word[3]         = 0x00000002
 0x639fe9a8: stack_word[2]         = 0x639fe9fc
 0x639fe9ac: stack_word[1]         = 0x43775cc0
 0x639fe9b0: stack_word[0]         = 0x43820148
 0x639fe9b4: istate->_thread       = 0x41271548
 0x639fe9b8: istate->_bcp          = 0x5ee167cb (bci 131)
 0x639fe9bc: istate->_locals       = 0x639fea28
 0x639fe9c0: istate->_constants    = 0x5ee18610
 0x639fe9c4: istate->_method       =
java.util.concurrent.CopyOnWriteArrayList.remove(Ljava/lang/Object;)Z
 0x639fe9c8: istate->_mdx          = 0x5ee18610
 0x639fe9cc: istate->_stack        = 0x639fe9a8
 0x639fe9d0: istate->_msg          = 0x00000000
 0x639fe9d4: istate->_result       = 0x639fe9b4
 0x639fe9d8: (istate->_result)     = 0x00000000
 0x639fe9dc: (istate->_result)     = 0x639fea20
 0x639fe9e0: istate->_prev_link    = 0x4073586c
 0x639fe9e4: istate->_oop_temp     = 0x00000000
 0x639fe9e8: istate->_stack_base   = 0x639fe9b4
 0x639fe9ec: istate->_stack_limit  = 0x639fe9a0
 0x639fe9f0: istate->_monitor_base = 0x639fe9b4
 0x639fe9f4: istate->_self_link    = 0x639fe9b4
 0x639fe9f8: frame_type            = INTERPRETER_FRAME
 0x639fe9fc: next_frame            = 0x639fea74
```

redhat

Ed’s port

- Creating that stack frame is *extremely* expensive, for both the interpreter and the JIT
- To do a really good job requires us to fix this

redhat

Ed's port

However, there are benefits to this approach.

- We can jump directly from compiled code to interpreting bytecode simply by setting the bytecode pointer and jumping to the entry point of the interpreter
- This means that we don't have to generate code for weird special cases
- It's not as good as real deoptimization, but it's good enough for most things

redhat

The ARM

- may be the processor of the future, for many application areas in addition to phones and tablets. Maybe not; who knows?
- has a very different architecture from the x86 we all know and, er, love
- presents new challenges to programmers, especially those who implement Java virtual machines

redhat

Weak memory consistency

x86 processors have a magic property: when you write to a series of memory locations from one core, every core sees that series of writes in the same order.

This means that you can do

```
a = 5; b = 7; ready = true;
```

and in some other thread

```
while (! ready) sched_yield();
foo (a, b);
```

ARM does not have this property!

redhat

Weak memory consistency

- On ARM, *when* other cores see changes to memory is largely undefined
- The only way to force stores from local cores to shared visible memory is to use a memory barrier
- The only way to force stores from other cores to local cache memory is to use a memory barrier
- This reveals many, many bugs written by programmers who think that “All the world’s an x86” or who don’t even understand how a memory model can affect them

redhat

Weak memory consistency

- There was no code in Ed's interpreter to handle *volatile*
- *volatile* requires barriers
- I've defined a macro GO_IF_VOLATILE that marks a branch between non-volatile and volatile code

```
(igetfield) igetfield {
        ldrb    r1, [jpc, #2]
        DISPATCH_START 3
        POP     tmp1
        add     tmp2, constpool, r1, lsl #12
        add     tmp2, tmp2, r2, lsl #4
        DISPATCH_NEXT
        GO_IF_VOLATILE r3, tmp2, 3f
        ldr     tmp2, [tmp2, #CP_OFFSET+8]
        DISPATCH_NEXT
.abortentry78:
        ldr     tmp2, [tmp1, tmp2]
        DISPATCH_NEXT
        DISPATCH_NEXT
        PUSH    tmp2
        DISPATCH_FINISH

3:
        VOLATILE_VERSION
        ldr     tmp2, [tmp2, #CP_OFFSET+8]
        DISPATCH_NEXT
.abortentry78_v:
        ldr     tmp2, [tmp1, tmp2]
        FullBarrier
        DISPATCH_NEXT
        DISPATCH_NEXT
        PUSH    tmp2
        DISPATCH_FINISH
```

redhat

Weak memory consistency

- GO_IF_VOLATILE looks like:

```
.macro  GO_IF_VOLATILE reg, cp_cache, label
ldr     \reg, [\cp_cache, #CP_OFFSET+CP_CACHE_FLAGS]
tst     \reg, #(1<<CP_CACHE_VOLATILE_FIELD_FLAG_BIT)
bne     \label
.set    dispatch_saved, dispatch_state
.endm
```

That's not so bad—just three instructions—but it slows down all accesses. The JIT doesn't have this problem because it knows at code generation time whether a field is volatile.

redhat

Weak memory consistency

- 64 bits, 2 words: `ldrd` is not atomic
- `ldrexd` might be atomic, but no-one at ARM wants to say that it is; the specification leaves it vague
- This sequence definitely is atomic, but it is not very efficient, and, what's worse, it writes to the location it's reading from
- As far as I can tell, it is correct!

```
0:   ldrexd    tmp2, tmp1, [r3]
     strexd    r2, tmp2, tmp1, [r3]
     teq       r2, #0
     bne       0b
```

redhat

Safepoints

- The Thumb2™ JIT didn’t have safepoints
- This means that if one thread is in a tight loop, GC never happens
- And neither does Ctrl-C
- This has been the most difficult part of the update, leading to many, many bugs
- Which nearly drove Xerxes and me around the bend
- The problem is that the bytecode pointer *must be correct* at a safepoint
- HotSpot JITs don’t do this: they look at the saved PC, but because this JIT pretends to be an interpreter...

redhat

Hard-fp ABI

- An entirely new ABI for ARM Linux that passes arguments in fp registers rather than on the stack and in int registers
- will be the default on most new Linux ports
- BUT... requires fp registers, so some popular targets like Raspberry Pi will still need the “old” ABI
- Supporting two ABIs is going to be a real PITA

redhat

It’s real Java™ compatible

- We passed the TCK last week!

redhat

Quo vadis?

- Can this design ever be competitive with a “real” HotSpot template interpreter and JIT?
- Let’s maintain this port on AARM32

redhat

Quo vadis?

- The future is ARM8, which includes AARM64
- The ARM architectures will be AARM32, Thumb 2, and AARM64
- Real ARM8 hardware is some time away

redhat

ARM 8

AARM64 is a totally different design:

- 32 64-bit registers
- A more “conventional” RISC
- Not so much a port from AARM32 as a rewrite

redhat

Quo vadis?

- There seems to be a real shortage of skilled assembly language programmers in the community outside Oracle
- Is anyone taught assembler any more?

redhat.

Questions?