

Porting Java™ to AArch64™

Bits of History, Words of Advice

Andrews Dinn and Haley
Red Hat Open Source Java Team
January 2014

In the beginning...

- In October 2011, AArch64, the 64-bit version of the ARM architecture, was announced
- The potential for high-density 64-bit ARM-based servers is very interesting to every company in the market
- There wasn't any public plan for Java, and there was a very real risk that the only implementation would be proprietary

In the beginning...

- It is extremely hard to find engineers with HotSpot porting experience, so
- we took the decision to do the port ourselves, with no experience of porting Java or HotSpot
- In hindsight this looks either brave or foolhardy, depending on your point of view

In the beginning...

- How were we going to estimate the size of the work?
- aph had once heard from someone whose name he has forgotten that two top-class engineers at Sun could port HotSpot in a year: one doing C2, the other doing the assembler, template interpreter, and C1
- So we guessed we could do it in the same time plus 50%
- even though we had no idea what we were doing
- In hindsight this looks either brave or foolhardy, depending on your point of view

In the beginning...

- In fact, we weren't quite as ignorant as that might suggest. We had considerable Java implementation experience: adinn with JikesRVM, and aph with GCJ.
- The 18 months “guesstimate” has turned out to be about right: it's taken slightly longer than that in elapsed time, but neither of us has been able to work on the task full time

In the beginning...

- We started on April Fools' day, more or less:

```
changeset:    0:2b6985c6a732
user:         "Andrew Dinn <adin@redhat.com>"
date:         Mon Apr 02 12:31:21 2012 +0100
summary:      start at implementing simulator for ARM64
```

...which brings us to the simulator

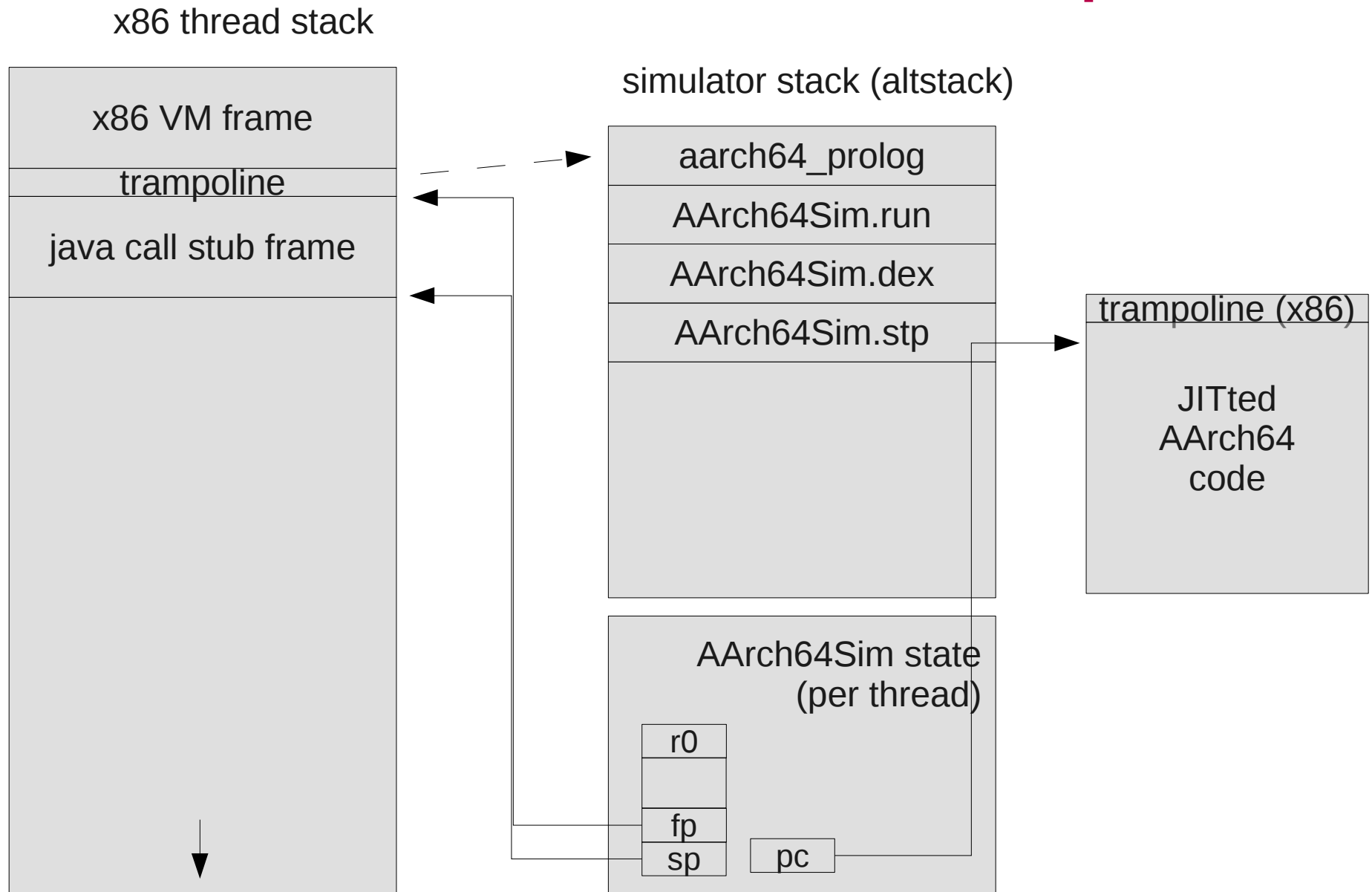
- We decided to write our own AArch64 simulator
- This would be a simple behavioural simulator that would only be used to execute code generated by the JIT compiler(s) and the template interpreter
- We would be independent of the OS-porting team, the GCC team, and ARM's proprietary simulators

The simulator

- This approach only works because AArch64 is architecturally compatible with AMD64
 - Same endianness
 - Same word size
 - Similar alignment rules

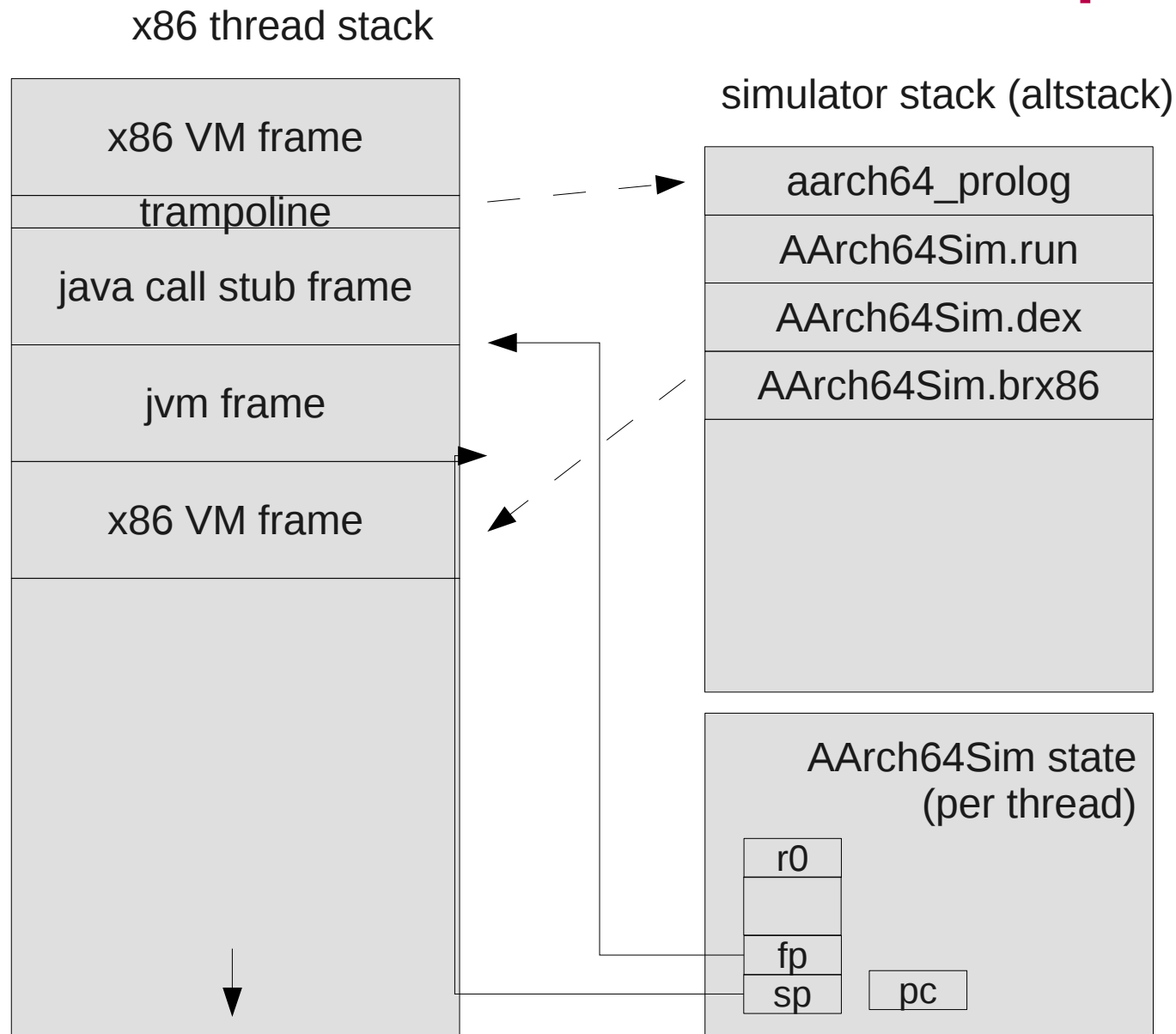
mixed mode execution

Two stacks per thread



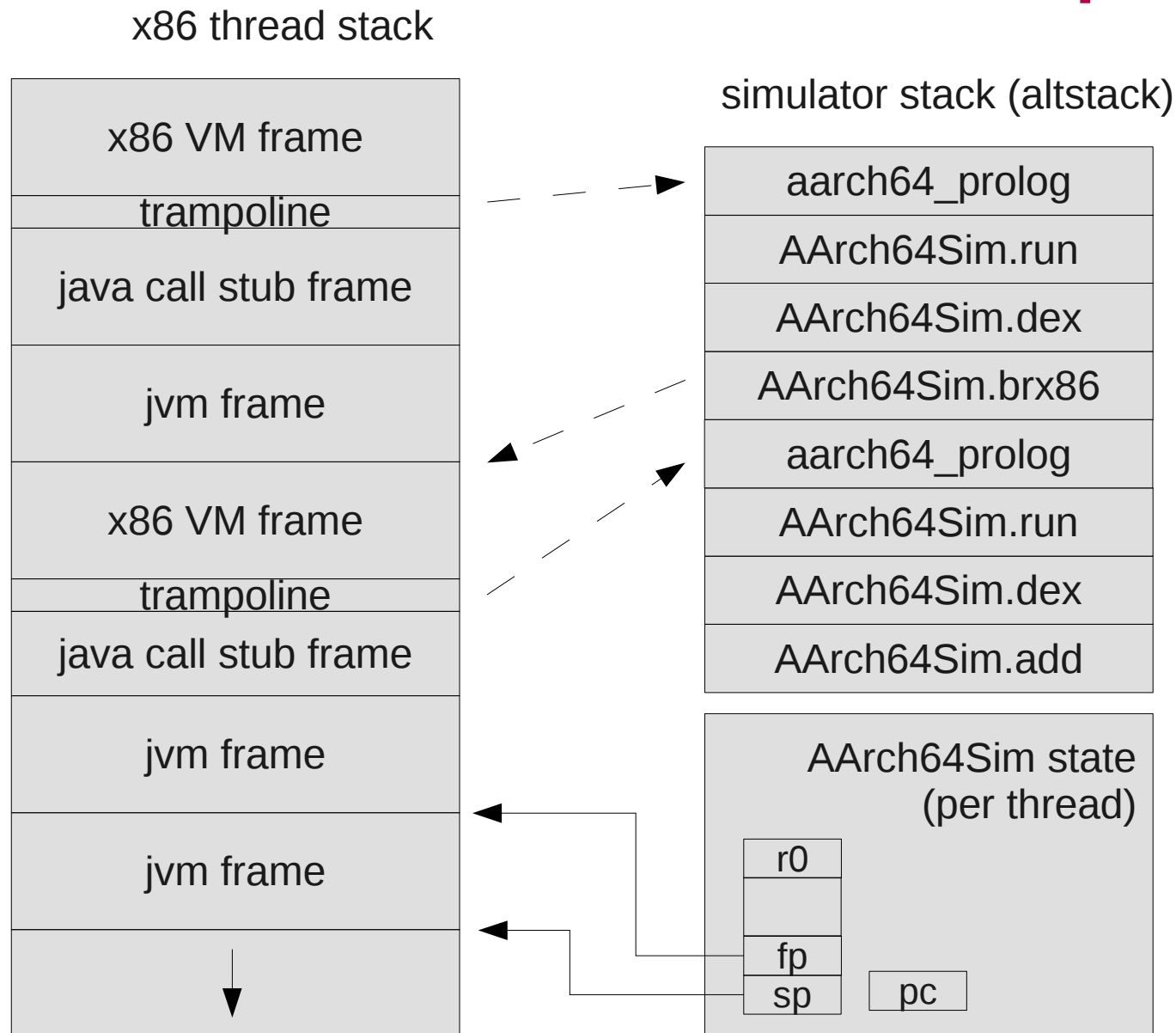
mixed mode execution (2)

Two stacks per thread



mixed mode execution (3)

Two stacks per thread



The simulator

- Was this decision correct? Discuss...
- aph thought it was a bit self-indulgent, but that it could be defended
- adinn's experience porting JikesRVM to simulated hardware led him to expect many of the actual benefits
- It's turned out to have been an excellent decision, despite the work involved

The simulator

- The simulator is tightly integrated with both GDB and HotSpot
- You can set symbolic breakpoints on JIT-generated code and bytecode
- The simulator itself is very easy to change to add specific conditions and traces
- It can be recompiled in less than 2 seconds
- The debugging environment is the best that we have ever seen
- So, in hindsight:

The simulator

- The decision to write our own simulator turns out to have been one of the best decisions we have ever made
- We estimate that it's saved months of effort and considerable frustration

Verifying the simulator and assembler

- There is a real risk from being totally independent of other tools: you might end up creating an entirely self-consistent world of your own that is different from the real hardware when it arrives!
- We had GNU binutils, so we wrote a Python program that generates assembly source for every instruction, runs GNU assembler, and saves the binary. It also generates source for the same instructions for HotSpot's assembler
- We do a bit-for-bit comparison whenever HotSpot starts in debug mode
- This is a really good idea. We found lots of errors.

Verifying the simulator and assembler

- When, much later, we wanted to run on other simulators – and indeed real hardware – we discovered only one major bug:
- The carry flag was the wrong way up: ARM subtract clears the carry on an overflow
- Who would have guessed that? The simulator and the template interpreter were in complete agreement
- It didn't take long to fix

Working in parallel

- It's very hard to share the work of porting HotSpot
- You have to alternate between writing the template interpreter and the runtime
- We don't think it's possible to have more than two people doing this, and even then it's difficult

Template Interpreter

- HotSpot's template interpreter is a hand-coded assembly language bytecode interpreter
- It's used at startup time to gather profile data that drives the JIT compilers
- There is also the C++ interpreter, which is slower, but it means that you don't have to spend development time writing a template interpreter

A template

```
void TemplateTable::dup()  
{  
    transition(vtos, vtos);  
    __ ldr(r0, Address(esp, 0));  
    __ push(r0);  
    __ // stack: ..., a, a  
}
```

Template Interpreter

- Writing the Template Interpreter got us used to the architecture
- Java startup for “Hello, World!” executes 750k bytecodes
- Was it really a good idea to write a template interpreter for AArch64? Should we have used the C++ interpreter? Not sure; discuss...

Template Interpreter

- Arguments in favour:
 - Performance matters because a lot of code is interpreted
 - It's a great learning exercise before you cut your teeth on the compilers

Mistakes we made

- Stack alignment:
 - We didn't realize that SP has to be 16-aligned or it will trigger a bus error. This means that you can't use SP for the interpreter's expression stack pointer. We had to rewrite a chunk of code to use another register for ESP. This restriction also means that the machine SP has to be adjusted whenever we enter or leave the interpreter, to make room for the interpreter's expression stack.

Mistakes we made

- Patching:
 - ARM has tight rules about which instructions can be patched while threads are operating concurrently. aph didn't realize, and wasted some time writing the C1 compiler's patching code. We now deoptimize whenever it's necessary to patch.

C1 and C2 compilers

- C1 is a quick 'n dirty JIT compiler that generates code from simple patterns
- The code it generates isn't pretty to look at, but it can be generated rapidly at startup time
- 64-bit HotSpot targets don't usually run C1
- Should we have written C1? We did, in order to get something working quickly and test the shared runtime code
- Was this really a good idea? Discuss...

C1 and C2 compilers

- Arguments in favour of writing C1:
 - It's a great learning exercise before you cut your teeth on C2
 - In particular, the SharedRuntime code, which is used by both compilers, is a lot easier to write and, more significantly, test when you
 - can easily understand how the compiler works and
 - are able easily to engineer code that will be compiled to a desired native sequence
 - It's useful as the first level of tiered compilation – for javac, etc.

C1 and C2 compilers

- Arguments against writing C1:
 - HotSpot, by default, doesn't even bother to build C1 for x86-64, even though it does work
 - We don't know why this is ... maybe it's not worth it?

C1 and C2 compilers

- C2 is the server JIT: a heavyweight optimizing compiler that uses the profile data produced by C1 and the template interpreter to compile and recompile code
- There is no choice: you must have C2 for a high-performance Java implementation
- We were scared by rumours of how difficult C2 was going to be
- In hindsight, it wasn't so bad. It was a lot of work, though.

A C2 pattern

Java:

```
result = (n >>> 12) & 7;
```

- This is the equivalent of C's bitfield extraction

A C2 pattern: bitfield extract

```
instruct ubfxwI(iRegINoSp dst, iRegI src,
    immI rshift, immI_bitmask mask)
%{
    match(Set dst (AndI (URShiftI src rshift) mask));
    ins_cost(DEFAULT_COST);
    format %{ "ubfxw $dst, $src, $mask" %}
    ins_encode %{
        int rshift = $rshift$$constant;
        long mask = $mask$$constant;
        int width = exact_log2(mask+1);
        __ ubfxw(as_Register($dst$$reg),
            as_Register($src$$reg), rshift, width);
    %}
    ins_pipe(pipe_class_default);
%}
```

C2 patterns

- If you're going to generate high-quality code you're going to have to write a lot of patterns
- Some of them can be automatically generated by means of evil m4 scripts
- For all of the gory details, the source is online

Here come the cavalry

- Linaro's governing board decided that they wanted to help us
- They weren't quite sure how, and to begin with we resisted
- In the end they, and in particular Ed Nevill, have been very helpful
- They tested on ARM's own simulators and on ...

Welcome to the real world

- It turned out that our simulator was very accurate
- Whether by luck or good judgment, Linaro found only a very few discrepancies when running in a real AArch64/Linux environment

Where are we now?

- Interpreter, C1, C2, and runtime code are all done
- We are ready to make a Beta release at the same time as JDK8 general availability
- There is performance tuning to be done when AArch64 hardware is more generally available

Where are we now?

- To-do list:
 - We don't use the Advanced SIMD unit at all. There are some significant optimization opportunities to be had
 - We haven't created a pipeline model – this means that C2 doesn't do any instruction scheduling. The reasons that you might want to schedule are complex, and depend on the microarchitecture of the CPU.
 - We don't have any C2 peepholes at all. We're not sure that there would be any point.

Where are we now?

- To-do list:
 - The C2 patterns could be tuned to optimize performance on real hardware
 - We've run JDK7, and it's fine. We have now pushed it to the public repo and will be releasing it along with JDK8. We're not sure how much use JDK7 will be.

To-do list:

- This is free software – you all know the situation
- We don't bite
- Questions?