Full-System Simulation of Java Workloads With RISC-V and the Jikes Research Virtual Machine

Martin Maas  Krste Asanovic  John Kubiatowicz
1st CARRV Workshop, October 14, 2017
Managed Languages

- Java, PHP, C#, Python, Scala
- JavaScript, WebAssembly
- Java, Swift, Objective-C

Servers
Web Browser
Mobile
Abstract
Evaluation methodology underpins all innovation in experimental computer science. It requires relevant workloads, appropriate experimental design, and rigorous analysis. Unfortunately, methodology is not keeping pace with the changes in our field. The rise of managed languages such as Java, C#, and Ruby in the past decade and the imminent rise of commodity multicore architectures for the next decade pose new methodological challenges that are not yet widely understood. This paper explores the consequences of our collective inattention to methodology on innovation, and suggests a remedy for addressing this problem.

Many developers today choose managed languages, which provide: (1) memory and type safety, (2) automatic memory management, (3) dynamic code execution, and (4) well-defined boundaries between type-safe and unsafe code (e.g., JNI and Pinvoke). Many such languages are also object-oriented. Managed languages include Java, C#, Python, and Ruby. C and C++ are not managed languages; they are compiled-ahead-of-time, not garbage collected, and unsafe. Unfortunately, managed languages add at least three new degrees of freedom to experimental evaluation: (1) a space–time trade-off due to garbage collection, in which heap size is a control variable, (2) nondeterminism due to adaptive optimization and just-in-time (JIT) compilation, and (3) managed language-specific features, such as dynamic class loading and many methods, that are not amenable to trivial empirical methods.
ISCA-2017 Proceedings

54 papers in total

Not evaluated using managed-language workloads

52

4 evaluated using managed-language workloads
Managed Language Challenges

Long-Running on Many Cores

Concurrent Tasks (GC, JIT)

Fine-grained Interactions
Limitations of Simulators

High-performance Emulation

Qemu

Cannot account for fine-grained details (e.g., barrier delays of ~10 cycles)

Cycle-accurate Simulation

Gem5

Too slow to run large-scale Java workloads

Realism
Limitations of Simulators

Realism  Qemu

Industry Adoption
Run managed workloads on real RISC-V hardware in FPGA-based simulation to enable modifying the entire stack
Two Pieces of Infrastructure

Easy-to-modify Hardware

Easy-to-modify Runtime System
Two Pieces of Infrastructure

Rocket Chip Ecosystem

Easy-to-modify Runtime System
Two Pieces of Infrastructure

Rocket Chip Ecosystem

Jikes Research Virtual Machine
Talk Outline

1. Porting JikesRVM to RISC-V
  Lessons learnt porting JikesRVM to RISC-V

2. Running JikesRVM on RocketChip
   Demo of JikesRVM running on real RISC-V hardware

3. Managed-Language Use Cases
   New research that is enabled by this infrastructure
PART I

1. Porting JikesRVM to RISC-V
   Lessons learnt porting JikesRVM to RISC-V

2. Running JikesRVM on RocketChip
   Demo of JikesRVM running on real RISC-V hardware

3. Managed-Language Use Cases
   New research that is enabled by this infrastructure
JikesRVM on RISC-V

- 15,000 lines of code in 86 files to port the non-optimizing baseline compiler
- Runs full JDK6 applications, including the Dacapo benchmark suite (no JDK7)
- Passes JikesRVM core test suite
The Jikes Research VM

JikesRVM is written in Java

VM Magic library provides primitives for low-level operations

Existing “Bootstrap” JVM
The Jikes Research VM

Step 1: Load JikesRVM into itself

Step 2: JIT compiler produces code and stores it to memory using machine-level primitives

Copy compiled code and state

Existing “Bootstrap” JVM

Step 3: Store Image to disk
The Jikes Research VM

Step 4: Bootloader loads image file into memory, implements syscalls, etc.

Step 5: Jump into pre-compiled boot function
Porting
The Jikes Research VM
The Environment: Yocto

Clone riscv-poky (GitHub: riscv/riscv-poky) and run: `bitbake meta-toolchain`

To add missing dependencies, edit: `meta-riscv/recipes-core/images/core-image-riscv.bb`

Install the resulting SDK image
Assembler Port

Don’t write the assembler by hand, use riscv/riscv-opcodes repository and generate it!
Baseline JIT Compiler

Call into assembler

Complex bytecodes are implemented in software

```java
@override
protected final void emit_multianewarray(TypeReference typeRef, int dimensions) {
    asm.emitLdFrom(T0, ArchEntryPoints.newArrayArrayType.getMethod().getOffset());
    asm.emitLval(A0, methods.getMethodId());
    asm.emitLval(A1, dimensions);
    asm.emitLval(A2, typeRef.getMethodId());
    asm.emitAddI(A3, FP, spTopOffset); // offset from FP to expression stack top
    asm.emitSllI(T1, A1, LOG_BYTES_IN_ADDRESS); // number of bytes of array dimension arg
    asm.emitAddD(A3, T1, A3); // offset from FP to expression stack top
    asm.emitJalR(RA, T0, 0);
    discardSlots(dimensions);
    pushAddr(A0);
}

@override
protected final void emit_arraylength() {
    popAddr(T0);
    asm.emitLwOffset(T1, T0, ObjectModel.getArrayLengthOffset());
    pushInt(T1);
}

@override
protected final void emit_atthrow() {
    asm.emitLdFrom(T0, ArchEntryPoints.atthrowMethod.getMethod().getOffset());
    peekAddr(A0, 0);
    asm.emitJalR(RA, T0, 0);
}
```
At the start of every Bytecode, emit a text sequence:
LD X0, 1024(X0) # SEGFAULT
(Number of instructions)
(Opcode)
(Stack Offset)

Print out instructions as
DASM(0x12345678)
and pipe the output through spike-dasm
Foreign-Function Calls

• **Two mechanisms**: JNI and syscalls

• **JNI** is bidirectional, requires yield point, keeping track of references, supports varargs, exceptions

• **Syscalls** are minimal Java-to-C calls
Much More...

- Exception delivery and bounds checks (across C and Java stack frames)
- **Dynamic linker** (trampolines, bridges)
- Yield points, root scanning for GC, Interface Method Tables, etc.
PART II

1. Porting JikesRVM to RISC-V
   Lessons learnt porting JikesRVM to RISC-V

2. Running JikesRVM on RocketChip
   Demo of JikesRVM running on real RISC-V hardware

3. Managed-Language Use Cases
   New research that is enabled by this infrastructure
running class initializer for java.util.logging.Logger
invoking method < BootstrapCL, Ljava/util/logging/Logger; >.<clinit> ()V
Initializing runtime compiler
Late stage processing of command line
[VM booted]
Extracting name of class to execute
Initializing Application Class Loader
Turning back on security checks. Letting people see the ApplicationClassLoader.
running class initializer for java.lang.ClassLoader$StaticData
invoking method < BootstrapCL, Ljava/lang/ClassLoader$StaticData; >.<clinit> ()V
RVMClassLoader.getApplicationClassLoader(): Initializing Application ClassLoader, with repositories: `.'...
RVMClassLoader.getApplicationClassLoader(): ...initialized Application classloader, to SystemAppCL
Creating main thread
Constructing mainThread
Starting main thread
Boot sequence completed; finishing boot thread
# MIDAS Performance Results

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Instructions (B)</th>
<th>Simulated Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>118.0</td>
<td>311.8</td>
</tr>
<tr>
<td>luindex</td>
<td>47.4</td>
<td>103.5</td>
</tr>
<tr>
<td>lusearch</td>
<td>263.5</td>
<td>597.2</td>
</tr>
<tr>
<td>pmd</td>
<td>158.5</td>
<td>346.8</td>
</tr>
<tr>
<td>sunflow</td>
<td>504.8</td>
<td>1,352.9</td>
</tr>
<tr>
<td>xalan</td>
<td>190.8</td>
<td>466.4</td>
</tr>
</tbody>
</table>

Default input sizes, >1 trillion instructions
PART III

1. Porting JikesRVM to RISC-V
   Lessons learnt porting JikesRVM to RISC-V

2. Running JikesRVM on RocketChip
   Demo of JikesRVM running on real RISC-V hardware

3. Managed-Language Use Cases
   New research that is enabled by this infrastructure
Hardware-Software Co-Design

Grail Quest: A New Proposal for Hardware-Assisted Garbage Collection

6th Workshop on Architectures and Systems for Big Data (ASBD '16), Seoul, Korea, June 2016
We found that the distortion introduced [by the method] unacceptably large and erratic. For example, with the GenMS collector, the [benchmark] reports a 12% to 33% increase in runtime versus running [without].

**Modifiable hardware enables fine-grained measurement and injection of language-level data without disturbing the application performance.**
Memory Allocation Latency

Sampling Rate: 1 KHz
Memory Allocation Latency

Sampling Rate: 1 KHz
Memory Allocation Latency

**Sampling Rate:** 1 KHz

**Every Allocation**
Logging Memory Allocations

All memory allocations in a program (color indicates allocation class size)
DRAM Row Misses

Dacapo Java Benchmarks on FPGA RISC-V core, FCFS open-page memory access scheduler. 800 Billion cycles @ 30MHz
DRAM Row Misses

Simulated time in seconds (assuming 1 GHz clock rate)
Conclusion
Conclusion

Combining RISC-V and JikesRVM enables new hardware-software research that modifies the hardware, operating system and managed runtime.
The RISC-V Foundation is launching a new J Extension Work Group to add managed-language support to RISC-V!

If you would like to get involved, talk to me or David Chisnall (david.chisnall@cl.cam.ac.uk)
Thank you! Any Questions?

+Thanks to David Biancolin, Christopher Celio, Sagar Karandikar, Donggyu Kim and all of UCB-BAR

Acknowledgements: Research was partially funded by DARPA Award Number HR0011-12-2-0016, DOE grant #DE-AC02-05CH11231, the STARnet Center for Future Architecture Research (C-FAR), and ASPIRE Lab sponsors and affiliates Intel, Google, HPE, Huawei, LGE, NVIDIA, Oracle, and Samsung.