A RISC-V JAVA UPDATE
Running Full Java Applications on FPGA-Based
RISC-V Cores with JikesRVM

Martin Maas  Krste Asanovic  John Kubiatowicz
7th RISC-V Workshop, November 28, 2017
Managed Languages
Managed Languages

Java, PHP, C#, Python, Scala
Managed Languages

Java, PHP, C#, Python, Scala

JavaScript, WebAssembly

Servers

Web Browser
Managed Languages

Java, PHP, C#, Python, Scala

JavaScript, WebAssembly

Java, Swift, Objective-C

Servers | Web Browser | Mobile
Java on RISC-V
Java on RISC-V

OpenJDK/Hotspot JVM

High-performance production JVM
Java on RISC-V

OpenJDK/Hotspot JVM

High-performance production JVM

Jikes Research VM

Easy-to-modify research JVM
Java on RISC-V

OpenJDK/Hotspot JVM

High-performance production JVM

Jikes Research VM

Easy-to-modify research JVM
Talk Outline
Talk Outline

1. Running JikesRVM on Rocket Chip

Executing JikesRVM on FPGA-based RISC-V hardware
Talk Outline

1. Running JikesRVM on Rocket Chip
   Executing JikesRVM on FPGA-based RISC-V hardware

2. Managed-Language Use Cases
   New research that is enabled by this infrastructure
Talk Outline

1. **Running JikesRVM on Rocket Chip**
   Executing JikesRVM on FPGA-based RISC-V hardware

2. **Managed-Language Use Cases**
   New research that is enabled by this infrastructure

3. **The State of Java on RISC-V**
   Progress, Challenges and Announcements
PART I

1. Running JikesRVM on Rocket Chip
   Executing JikesRVM on FPGA-based RISC-V hardware

2. Managed-Language Use Cases
   New research that is enabled by this infrastructure

3. The State of Java on RISC-V
   Progress, Challenges and Announcements
JikesRVM on RISC-V

- Runs **full JDK6 applications**, including the Dacapo benchmark suite (no JDK7)
- Passes JikesRVM core test suite
- **15,000 lines of code in 86 files** to port the non-optimizing baseline compiler
Porting
The Jikes Research VM
CARRV-2017 Workshop Paper

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

Martin Maas
University of California, Berkeley
maas@eecs.berkeley.edu
Krste Asanović
University of California, Berkeley
krste@eecs.berkeley.edu
John Kubiatowicz
University of California, Berkeley
kubitron@eecs.berkeley.edu
init: version 2.88 booting

random: fast init done

EXT2-fs (generic-blkdev): warning: mounting unchecked fs, running e2fsck is recommended

bootlogd: cannot find console device 4:0 under /dev

hwclock: can't open '/dev/misc/rtc': No such file or directory
Thu Sep 21 17:11:37 UTC 2017

hwclock: can't open '/dev/misc/rtc': No such file or directory

INIT: Entering runlevel: 5

-- JIKES RVM TEST ENVIRONMENT --

chmod: driver-test: No such file or directory

bash: cannot set terminal process group (138): Inappropriate ioctl for device
bash: no job control in this shell
bash-4.4#
bash-4.4#
bash-4.4# ./rvm -X:verboseBoot=10 CARRV

random: crng init done
Booting
Setting up current RVMThread
Doing thread initialization
Setting up memory manager: bootrecord = 0x0000000031000018
Initializing baseline compiler options to defaults
Fetching command-line arguments
Early stage processing of command line
Collector processing rest of boot options
Initializing bootstrap class loader: jksvm.jar: rvmrt.jar
Running various class initializers
running class initializer for java.util.WeakHashMap
invoking method < BootstrapCL, Ljava/util/WeakHashMap; >.<clinit> ()V
running class initializer for org.jikesrvm.classloader.Atom$InternedStrings
invoking method < BootstrapCL, Lorg/jikesrvm/classloader/Atom$InternedStrings; >.<clinit> ()V
running class initializer for gnu.classpath.SystemProperties
invoking method < BootstrapCL, Lgnu/classpath/SystemProperties; >.<clinit> ()V
running class initializer for java.lang.Throwables$StaticData
invoking method < BootstrapCL, Ljava/lang/Throwables$StaticData; >.<clinit> ()V
running class initializer for java.lang.Runtime
invoking method < BootstrapCL, Ljava/lang/Runtime; >.<clinit> ()V
running class initializer for java.lang.System
invoking method < BootstrapCL, Ljava/lang/System; >.<clinit> ()V
running class initializer for sun.misc.Unsafe
invoking method < BootstrapCL, Lsun/misc/Unsafe; >.<clinit> ()V
running class initializer for java.lang.Character
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 class loader, to SystemAppCL

Creating main thread
Constructing mainThread
Starting main thread

Boot sequence completed; finishing boot thread

CARRY
## FPGA 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 II

1. Running JikesRVM on Rocket Chip
   Executing JikesRVM on FPGA-based RISC-V hardware

2. Managed-Language Use Cases
   New research that is enabled by this infrastructure

3. The State of Java on RISC-V
   Progress, Challenges and Announcements
Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century


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 recommends solutions 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 sampling technologies, and (3) system management, (3) dynamic code execution, and (4) well-defined boundaries between type-safe and unsafe code (e.g., JNI and Pinvoke).
Managed Language Challenges
Managed Language Challenges

Long-Running on Many Cores
Managed Language Challenges

Long-Running on Many Cores

Concurrent Tasks (GC, JIT)
Managed Language Challenges

- Long-Running on Many Cores
- Concurrent Tasks (GC, JIT)
- Fine-grained Interactions
Limitations of Simulators
Limitations of Simulators

High-performance Emulation

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

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

- **Cycle-accurate Simulation**
  Too slow to run large-scale Java workloads
Limitations of Simulators

High-performance Emulation

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

Cycle-accurate Simulation

Too slow to run large-scale Java workloads

Realism
Limitations of Simulators

Realism
Limitations of Simulators

Realism

Industry Adoption
Run managed workloads on real RISC-V hardware in FPGA-based simulation to enable modifying the entire stack.
Grail Quest: A New Proposal for Hardware-assisted Garbage Collection

Martin Mao  Keith Aramvati  John Kubiatowicz
University of California, Berkeley

1. INTRODUCTION

A substantial portion of big data frameworks — and large-scale distributed workloads in general — are written in garbage-collected languages (GC) such as Java, Scala, Python or R. Due to its importance for a wide range of workloads, Garbage Collection has even been pushed into hardware: energy consumption. Previous work [2] has shown that GC can account for up to 25% of energy consumption in a server or mobile SoC. This requires isolating the GC logic into a small number of IP blocks and limiting the dependencies on external components [5, 7, 8]. We believe an approach has to be relatively non-invasive to be adopted. The current trend to accelerators and processing near memory may make it easier to adopt similar techniques for GC without substantial modifications to the architecture.

Most garbage-collected workloads run on servers and we discuss design choices, trade-offs and running a substantial number of experiments. However, these approaches typically ignore another factor that is very important in warehouse-scale computing — coherency. Modern GCs can be made energy efficient by moving them into hardware close to DRAM, to save power and area. Unfortunately, this is a poor fit for general-purpose cores. We move them into accelerators close to DRAM, to save power and area. However, this is a poor fit for general-purpose cores, partly due to Moore’s law and the bar for adding hardware-support into such a chip is very high. Yet, high-speed Transistor-transistor logic (TTL) circuits may be an option. However, that is changing, and new hardware and new machine architectures are expected to enable this vision, which could reasonably hope to improve GC performance and efficiency.

Many proposals were very invasive and would require re-architecting of the memory system or other components [5, 7, 8]. We believe an approach has to be relatively non-invasive to be adopted. The current trend to accelerators and processing near memory may make it easier to adopt similar techniques for GC without substantial modifications to the architecture.

Despite these trends that make it the perfect time to revisit this approach, we are not the first to propose hardware support for GC. There are two fundamental GC strategies: tracing and reference counting. Tracing collectors start from a root and follow references, while reference collectors repair the set of roots (such as static or stack variables) and perform a fast initialization. The latter has been widely adopted. We believe that there are three reasons why garbage-collected languages are widely used, but they are rarely the only workloads on a system. Often, they are written in garbage-collected languages but only a small portion of the workload is garbage-collected. Many proposals are very invasive and would require re-architecting of the memory system or other components [5, 7, 8]. We believe an approach has to be relatively non-invasive to be adopted. The current trend to accelerators and processing near memory may make it easier to adopt similar techniques for GC without substantial modifications to the architecture.

In this paper, we describe our proposed design. It extends the 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

35
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
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

- GC Pauses
- Mark Phase
- Root Scanning

Simulated time in seconds (assuming 1 GHz clock rate)
PART III

1. Running JikesRVM on Rocket Chip
   Executing JikesRVM on FPGA-based RISC-V hardware

2. Managed-Language Use Cases
   New research that is enabled by this infrastructure

3. The State of Java on RISC-V
   Progress, Challenges and Announcements
JVM on RISC-V Progress

- **Jikes Research JVM:**
  - Baseline JIT, no optimizing JIT port yet
- **OpenJDK HotSpot JVM:**
  - Runs with zero backend, but no high-performance JIT compiler port yet
We need your help!

Are you interested in working on the OpenJDK port?
Announcement

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)