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

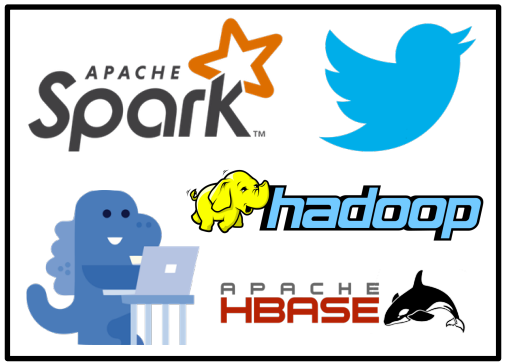
Martin Maas Krste Asanovic John Kubiatowicz

1st CARRV Workshop, October 14, 2017

Berkeley Architecture Research

Managed Languages

Java, PHP, C#, Python, Scala



JavaScript, WebAssembly

Java, Swift, Objective-C

Servers → Web Browser → Mobile

DOI:10.1145/1378704.1378723

Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century

CACM 08/2008

By Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann

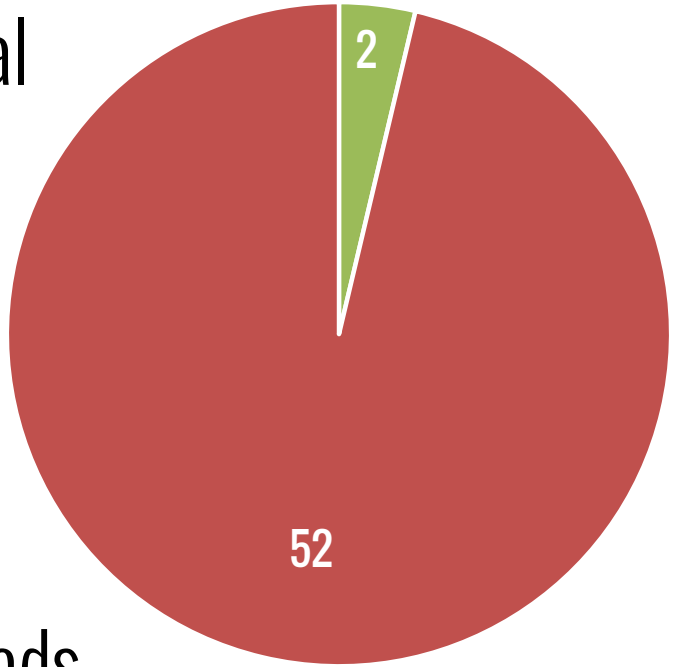
Abstract

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,

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 vari-

ISCA-2017 Proceedings

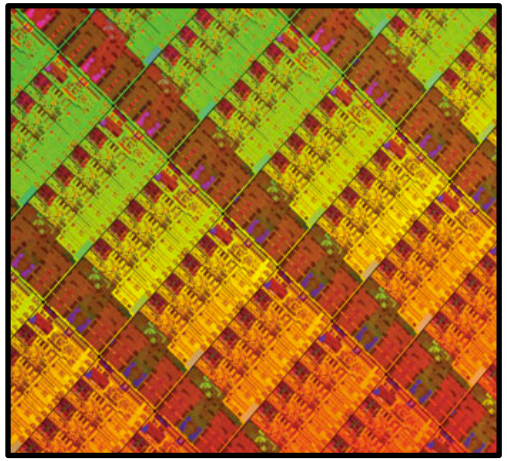
54 papers in total



Evaluated using managed-language workloads

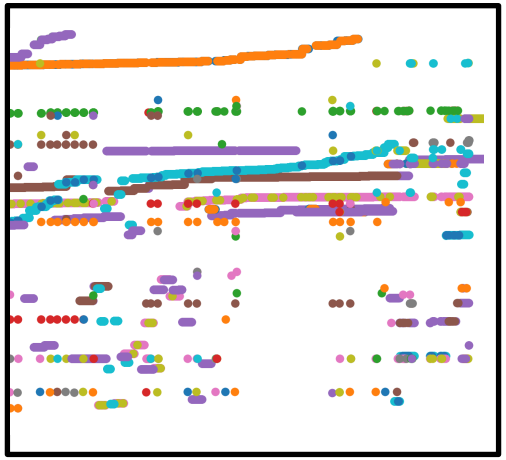
Not evaluated using managed-language workloads

Managed Language Challenges



Long-Running on Many Cores

Concurrent Tasks (GC, JIT)



Fine-grained Interactions

Limitations of Simulators

High-performance Emulation

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

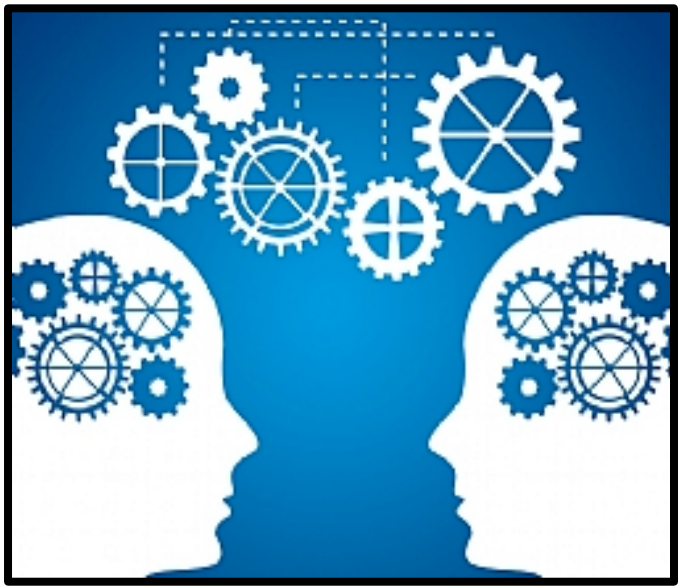
Cycle-accurate Simulation

Too slow to run large-scale Java workloads

Realism

Limitations of Simulators

Realism

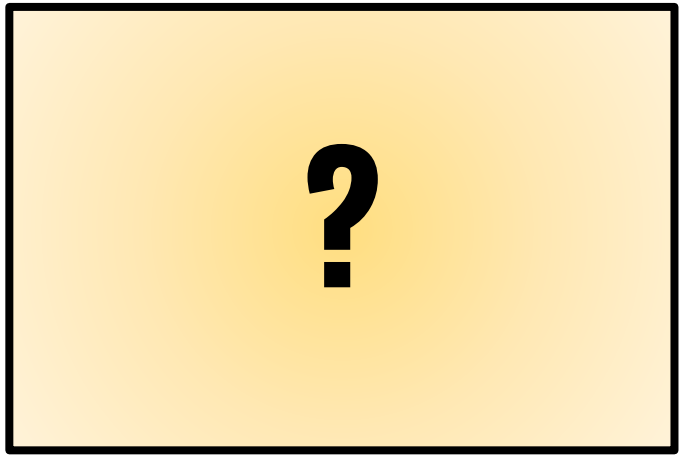


Industry Adoption

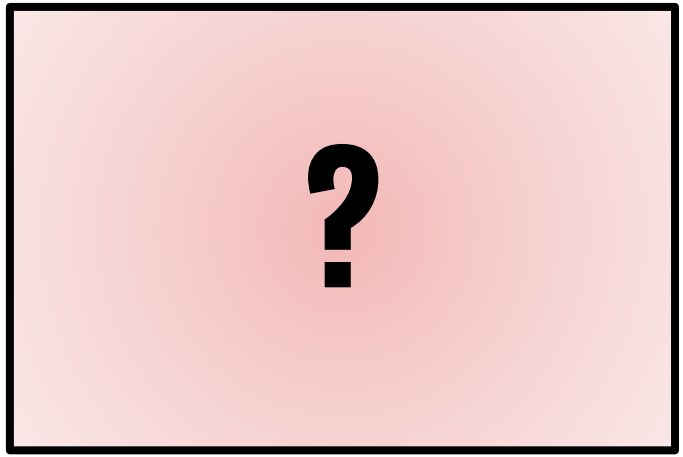

RISC-V

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

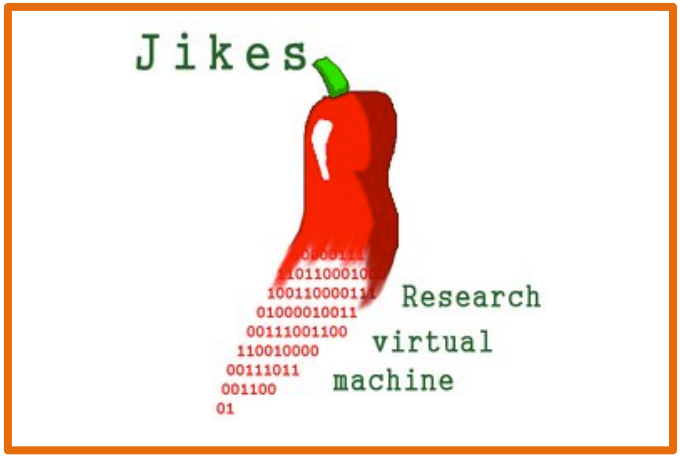
Rocket Chip Ecosystem

?

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

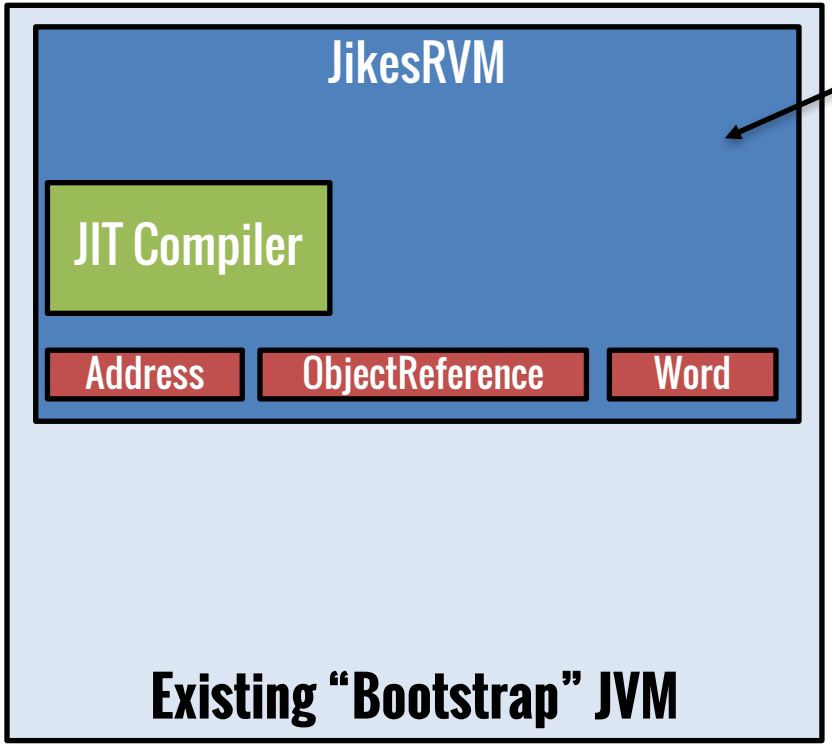
PART I

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

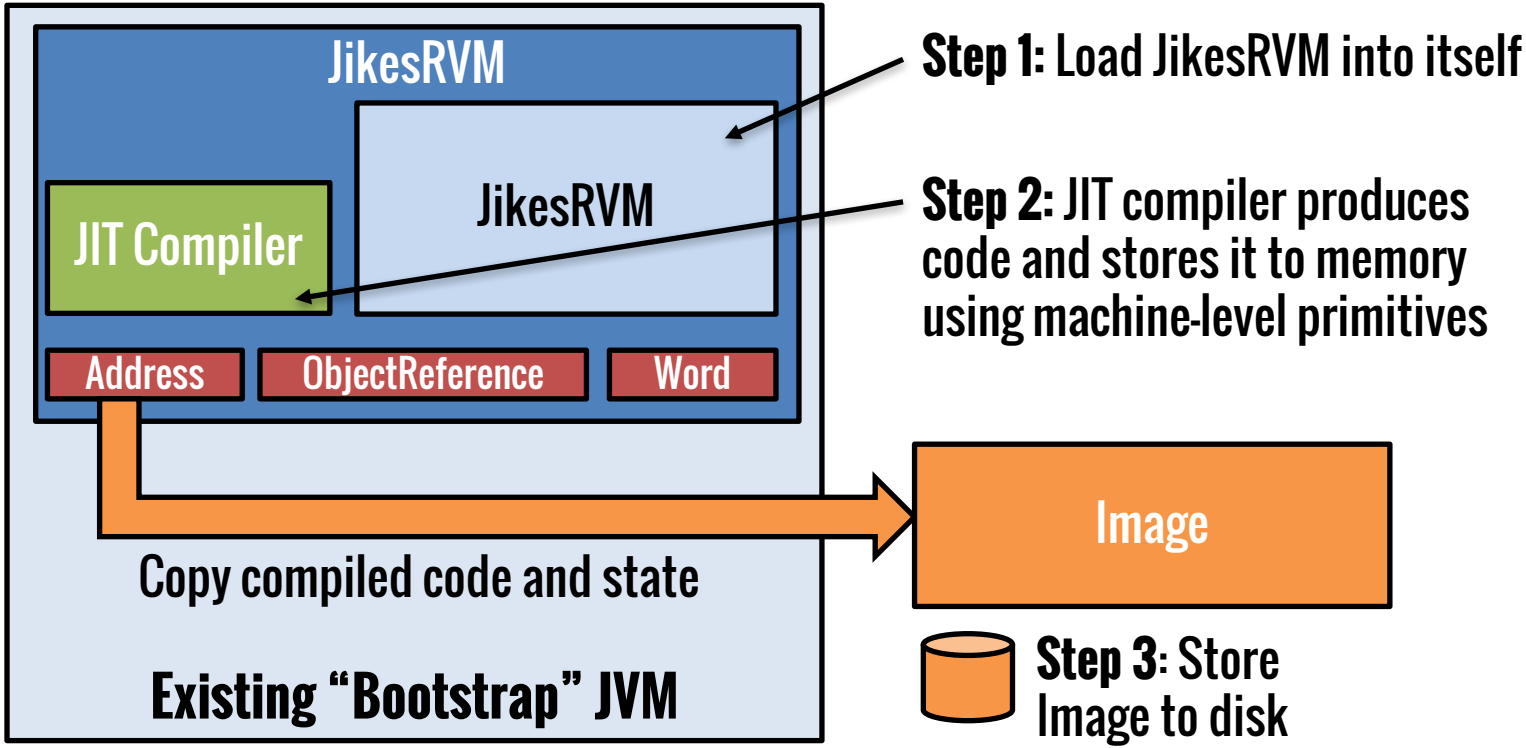
The Jikes Research VM



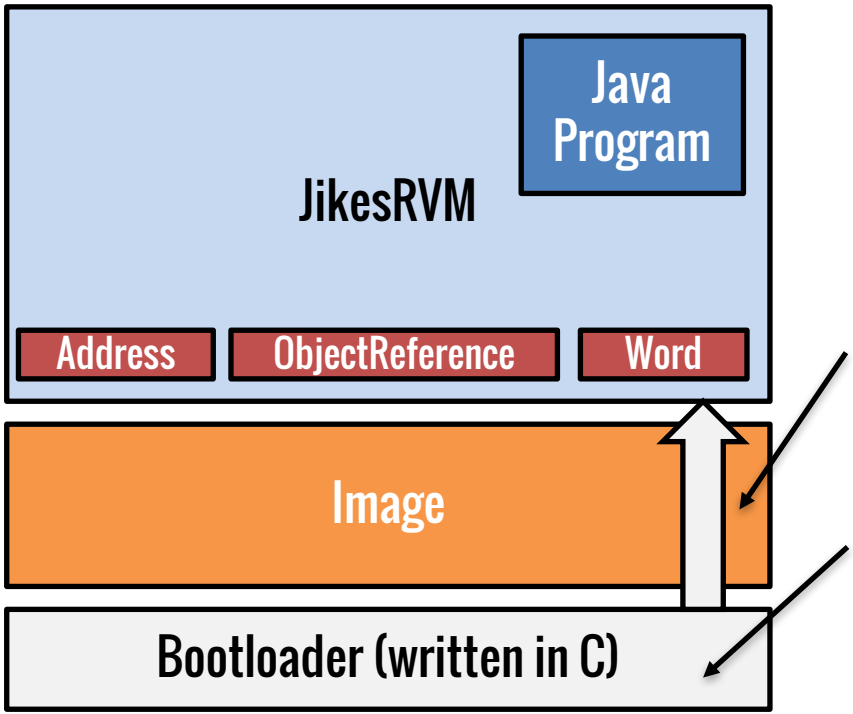
JikesRVM is written in Java

VM Magic library provides primitives for low-level operations

The Jikes Research VM



The Jikes Research VM



Step 5: Jump into pre-compiled boot function

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

Porting
The Jikes Research VM

The Environment: Yocto


```
    http://yoctoproject.org/documentation

For more information about OpenEmbedded see their website:
    http://www.openembedded.org/

You had no conf/bblayers.conf file. The configuration file has been created for
you with some default values. To add additional metadata layers into your
configuration please add entries to this file.

The Yocto Project has extensive documentation about OE including a reference manual
which can be found at:
    http://yoctoproject.org/documentation

For more information about OpenEmbedded see their website:
    http://www.openembedded.org/



### Shell environment set up for builds. ###

You can now run 'bitbake <target>'

maas@a6:/scratch/maas/poky/demo/riscv-poky/build$ bitbake
Parsing recipes:  29% |###############
```

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

```
martin@virtualbox:~/src/riscv/toolchain$ ./poky-riscv-glibc-x86_64-meta-toolchain-riscv64-toolchain-2.0+snapshot.sh
Poky (Yocto Project Reference Distro) SDK installer version 2.0+snapshot
==========================================================================
Enter target directory for SDK (default: /opt/poky-riscv/2.0+snapshot): ~/src/riscv/toolchain/sdk
You are about to install the SDK to "/home/martin/src/riscv/toolchain/sdk". Proceed[Y/n]? Y
Extracting SDK.............done
Setting it up...done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the environment setup script e.g.
 $ . /home/martin/src/riscv/toolchain/sdk/environment-setup-riscv64-poky-linux
martin@virtualbox:~/src/riscv/toolchain$ . /home/martin/src/riscv/toolchain/sdk/environment-setup-riscv64-poky-linux
```

Assembler Port

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

```
def make_jikesrvm():
  for name in namelist:
    args = arguments[name]

    funargs = []
    substargs = []

    register_order = ['rd', 'rs1', 'rs2', 'rs3']

    # loads/stores have different order of registers
    if name in ['lb', 'lh', 'lw', 'ld', 'flw', 'fld', 'sb', 'sh', 'sw', 'sd', 'fsw', 'fsd']:
      register_order = ['rd', 'rs2', 'rs1']

    fcvt_float2int = []
    fcvt_int2float = []
    fcmp = []

    for f in ['d', 's']:
      for i in ['l', 'w', 'lu', 'wu']:
          fcvt_float2int.append('fcvt.%s.%s' % (i,f))
          fcvt_int2float.append('fcvt.%s.%s' % (f,i))

      for instr in ['fle', 'flt', 'feq']:
        fcmp.append('%s.%s' % (instr, f))

    for r in register_order:
      if r in args:
        if name.startswith('f'):
                if name in ['flw', 'fld', 'fsw', 'fsd'] and r == 'rs1':
                        funargs.append('GPR ' + r)
                elif name in fcvt_float2int and r == 'rd':
                        funargs.append('GPR ' + r)
                elif name in fcvt_int2float and r == 'rs1':
                        funargs.append('GPR ' + r)
```

```
public final void emitLH(GPR rd, GPR rs1, int imm12) {
  int mi = 0x1003 | rd.value() << 7 | rs1.value() << 15 | make_imm12(imm12);
  mIP++;
  mc.addInstruction(mi);
}

public final void emitLW(GPR rd, GPR rs1, int imm12) {
  int mi = 0x2003 | rd.value() << 7 | rs1.value() << 15 | make_imm12(imm12);
  mIP++;
  mc.addInstruction(mi);
}

public final void emitLD(GPR rd, GPR rs1, int imm12) {
  int mi = 0x3003 | rd.value() << 7 | rs1.value() << 15 | make_imm12(imm12);
  mIP++;
  mc.addInstruction(mi);
}

public final void emitLBU(GPR rd, GPR rs1, int imm12) {
  int mi = 0x4003 | rd.value() << 7 | rs1.value() << 15 | make_imm12(imm12);
  mIP++;
  mc.addInstruction(mi);
}

public final void emitLHU(GPR rd, GPR rs1, int imm12) {
  int mi = 0x5003 | rd.value() << 7 | rs1.value() << 15 | make_imm12(imm12);
  mIP++;
  mc.addInstruction(mi);
}

public final void emitLWU(GPR rd, GPR rs1, int imm12) {
  int mi = 0x6003 | rd.value() << 7 | rs1.value() << 15 | make_imm12(imm12);
  mIP++;
  mc.addInstruction(mi);
}
```

Baseline JIT Compiler

Call into assembler

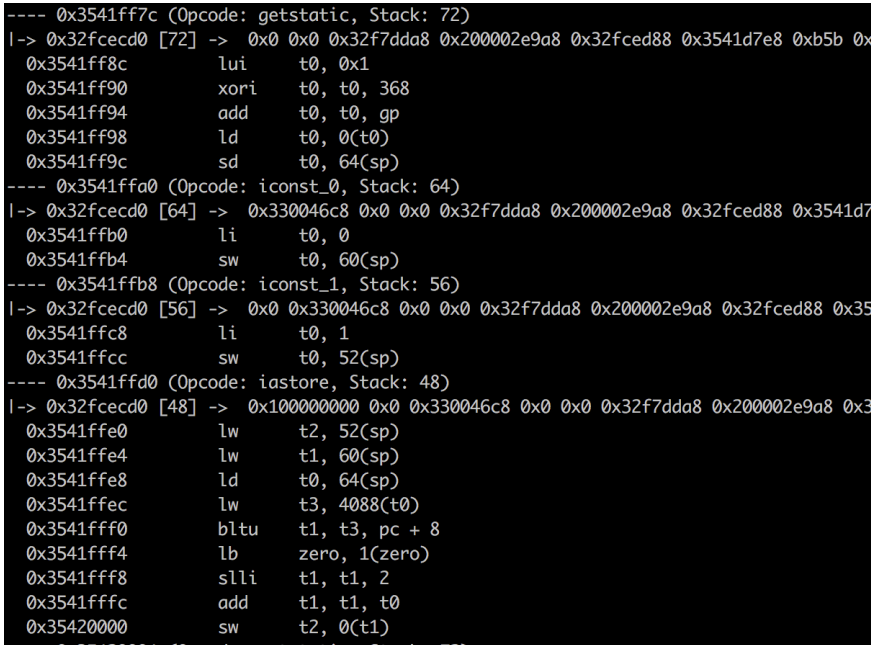
Complex bytecodes are implemented in software

```
@Override
protected final void emit_multianewarray(TypeReference typeRef, int dimensions) {
  asm.emitLDtoc(T0, ArchEntrypoints.newArrayArrayMethod.getOffset());
  asm.emitLVAL(A0, method.getId());
  asm.emitLVAL(A1, dimensions);
  asm.emitLVAL(A2, typeRef.getId());
  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 args
  asm.emitADD(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_athrow() {
  asm.emitLDtoc(T0, Entrypoints.athrowMethod.getOffset());
  peekAddr(A0, 0);
  asm.emitJALR(RA, T0, 0);
}
```

Debugging



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
```

```
bash-4.4#
```

MIDAS Performance Results

Benchmarks	Instructions (B)	Simulated Time (s)
avrora	118.0	311.8
luindex	47.4	103.5
lusearch	263.5	597.2
pmd	158.5	346.8
sunflow	504.8	1,352.9
xalan	190.8	466.4

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

Martin Maas Krste Asanović John Kubiatowicz
University of California, Berkeley

ABSTRACT

Many big data systems are written in garbage-collected languages and GC has a substantial impact on throughput, responsiveness and predicability of these systems. However, despite decades of research, there is still no "Holy Grail" of GC: a collector with no measurable impact, even on real-time applications. Such a collector needs to achieve freedom from pauses, high GC throughput and good memory utilization, without slowing down application threads or using substantial amounts of compute resources.

In this paper, we propose a step towards this elusive goal by reviving the old idea of moving GC into hardware. We discuss the trends that make it the perfect time to revisit this approach and present the design of a hardware-assisted GC that aims to reconcile the conflicting goals. Our system is work in progress and we discuss design choices, trade-offs and open questions.

1. INTRODUCTION

A substantial portion of big data frameworks – and large-scale distributed workloads in general – are written in languages with Garbage Collection (GC), such as Java, Scala, Python or R. Due to its importance for a wide range of workloads, Garbage Collection has seen tremendous research efforts for over 50 years. Yet, we arguably still don't have what has been called the "Holy Grail" of GC [1]: a pause-free collector that achieves high memory utilization and high GC throughput (i.e., sustaining high allocation rates), preferably without a large resource cost for the application.

Many recent GC innovations have focused on the first three goals, and modern GCs can be made effectively pause-free at the cost of slowing down application threads and using a substantial amount of resources. Moreover, these approaches oftentimes ignore another factor that is very important in warehouse-scale computers: energy consumption. Previous work [2] has shown that GC can account for up to 25% of energy and 40% of execution time in common workloads (10% on average). Worse, as big data systems are processing ever larger heaps, these numbers will likely increase.

We believe that we can reconcile low pause times and energy efficiency by revisiting the old idea of moving GC into hardware. Our goal is to build a GC that simultaneously achieves high GC throughput, good memory utilization, pause times indistinguishable from LLC misses and energy efficiency. We build on an algorithm that performs well on the first three criteria but is resource-intensive [3]. Our key insight is that this algorithm can be made energy efficient by moving it into hardware, combined with some algorithmic changes.

We are not the first to propose hardware support for GC [3–7]. However, none of these schemes has been widely adopted. We believe that there are three reasons:

Garbage-collected languages are widely used, but they are rarely the only workload on a system. Systems designed for specific languages mostly lost out to general-purpose cores, partly due to Moore's law and economies of scale allowing these cores to quickly outperform the specialized ones. This is changing today, as the slow-down of Moore's law makes it more attractive to use chip area for accelerators to improve common workloads, such as garbage-collected applications.

Most garbage-collected workloads run on servers (note that there are exceptions, such as Android applications). Servers traditionally use commodity CPUs and the bar for adding hardware-support into such a chip is very high (take Hardware Transactional Memory as an example). However, this is changing: cloud hardware and rack-scale machines in general are expected to switch to custom SoCs, which could easily incorporate IP 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.

We therefore think it is time to revisit hardware-assisted GC. In contrast to many previous schemes, we focus on making our design sufficiently non-invasive to incorporate into a server or mobile SoC. This requires isolating the GC logic into a small number of IP blocks and limiting changes outside these blocks to a minimum.

In this paper, we describe our proposed design. It exploits two insights: First, overheads of concurrent GCs stem from a large number of small but frequent slowdowns spread throughout the execution of the program. We move the culprits (primarily barriers) into hardware to alleviate their impact and allow out-of-order cores to speculate over them. Second, the most resource-intensive phases of a GC (marking and relocation) are a poor fit for general-purpose cores. We move them into accelerators close to DRAM, to save power and area.

2. BACKGROUND

An extensive body of work has been published on GC. Jones and Lins [9] provide a general introduction.

There are two fundamental GC strategies: tracing and reference counting. Tracing collectors start from a set of *roots* (such as static or stack variables), perform a

1

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

Motivating Example

> 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

Quantifying the Performance of Garbage Collection vs. Explicit Memory Management

Matthew Hertz *
Computer Science Department
Canisius College
Buffalo, NY 14208
matthew.hertz@canisius.edu

Emery D. Berger
Dept. of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
emery@cs.umass.edu

ABSTRACT

Garbage collection yields numerous software engineering benefits, but its quantitative impact on performance remains elusive. One can compare the cost of *conservative* garbage collection to explicit memory management in C/C++ programs by linking in an appropriate collector. This kind of direct comparison is not possible for languages designed for garbage collection (e.g., Java), because programs in these languages naturally do not contain calls to `free`. Thus, the actual gap between the time and space performance of explicit memory management and *precise*, copying garbage collection remains unknown.

We introduce a novel experimental methodology that lets us quantify the performance of precise garbage collection versus explicit memory management. Our system allows us to treat unaltered Java programs as if they used explicit memory management by relying on oracles to insert calls to `free`. These oracles are generated from profile information gathered in earlier application runs. By executing inside an architecturally-detailed simulator, this "oracular" memory manager eliminates the effects of consulting an oracle while measuring the costs of calling `malloc` and `free`. We evaluate two different oracles: a liveness-based oracle that aggressively frees objects immediately after their last use, and a reachability-based oracle that conservatively frees objects just after they are last reachable. These oracles span the range of possible placement of explicit deallocation calls.

We compare explicit memory management to both copying and non-copying garbage collectors across a range of benchmarks using the oracular memory manager, and present real (non-simulated) runs that lend further validity to our results. These results quantify the time-space tradeoff of garbage collection: with five times as much memory, an Appel-style generational collector with a non-copying mature space matches the performance of reachability-based explicit memory management. With only three times as much memory, the collector runs on average 17% slower than explicit memory management. However, with only twice as much memory, garbage collection degrades performance by nearly 70%. When physical memory is scarce, paging causes garbage collection to run an order of magnitude slower than explicit memory management.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Dynamic storage management; D.3.4 [**Processors**]: Memory management (garbage collection)

General Terms

Experimentation, Measurement, Performance

Keywords

oracular memory management, garbage collection, explicit memory management, performance analysis, time-space tradeoff, throughput, paging

1. Introduction

Garbage collection, or automatic memory management, provides significant software engineering benefits over explicit memory management. For example, garbage collection frees programmers from the burden of memory management, eliminates most memory leaks, and improves modularity, while preventing accidental memory overwrites ("dangling pointers") [50, 59]. Because of these advantages, garbage collection has been incorporated as a feature of a number of mainstream programming languages.

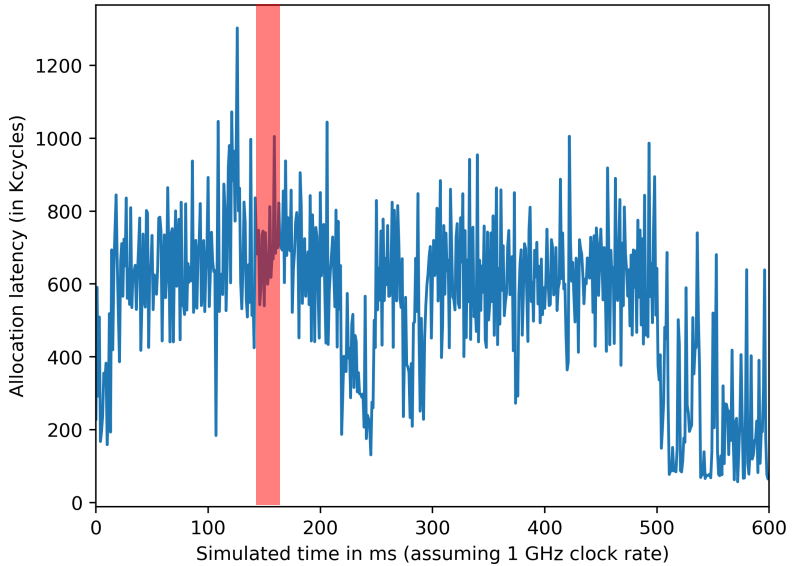
Garbage collection can improve programmer productivity [48], but its impact on performance is difficult to quantify. Previous researchers have measured the runtime performance and space impact of *conservative*, non-copying garbage collection in C and C++ programs [19, 62]. For these programs, comparing the performance of explicit memory management to conservative garbage collection is a matter of linking in a library like the Boehm-Demers-Weiser collector [14]. Unfortunately, measuring the performance trade-off in languages designed for garbage collection is not so straightforward. Because programs written in these languages do not explicitly deallocate objects, one cannot simply replace garbage collection with an explicit memory manager. Extrapolating the results of studies with conservative collectors is impossible because precise, relocating garbage collectors (suitable only for garbage-collected languages) consistently outperform conservative, non-relocating garbage collectors [10, 12].

It is possible to measure the costs of garbage collection activity (e.g., tracing and copying) [10, 20, 30, 36, 56] but it is impossible to subtract garbage collection's effect on mutator performance. Garbage collection alters application behavior both by visiting and reorganizing memory. It also degrades locality, especially when physical memory is scarce [61]. Subtracting the costs of garbage collection also ignores the improved locality that explicit memory managers can provide by immediately recycling just-freed memory [53, 55, 57, 58]. For all these reasons, the costs of precise,

* Work performed at the University of Massachusetts Amherst.

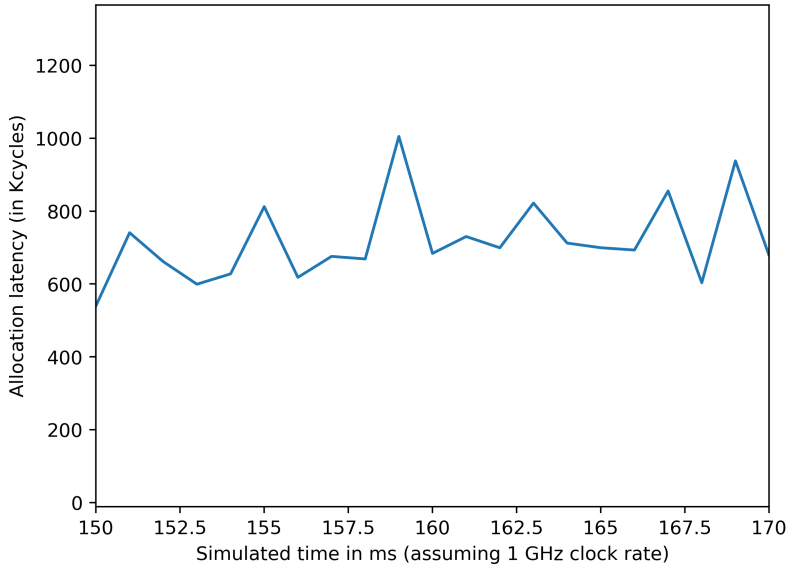
OOPSLA'05, October 16–20, 2005, San Diego, California, USA.

Memory Allocation Latency



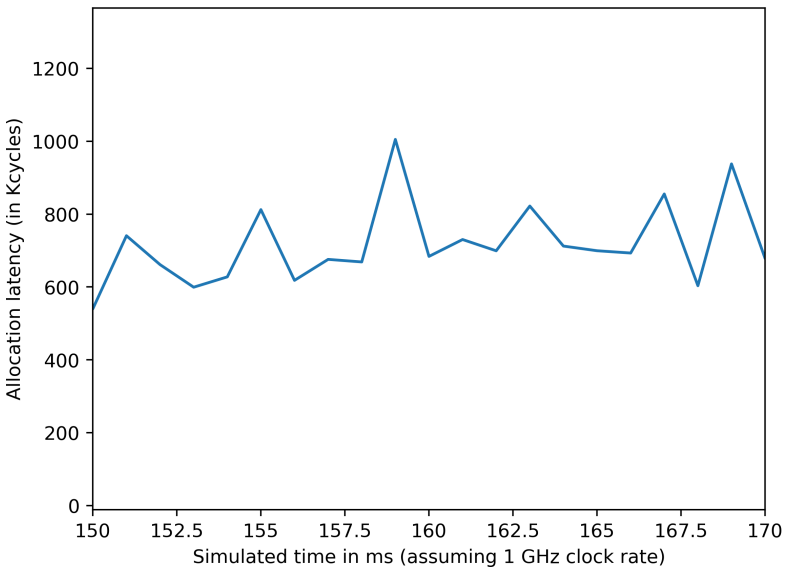
Sampling Rate: 1 KHz

Memory Allocation Latency

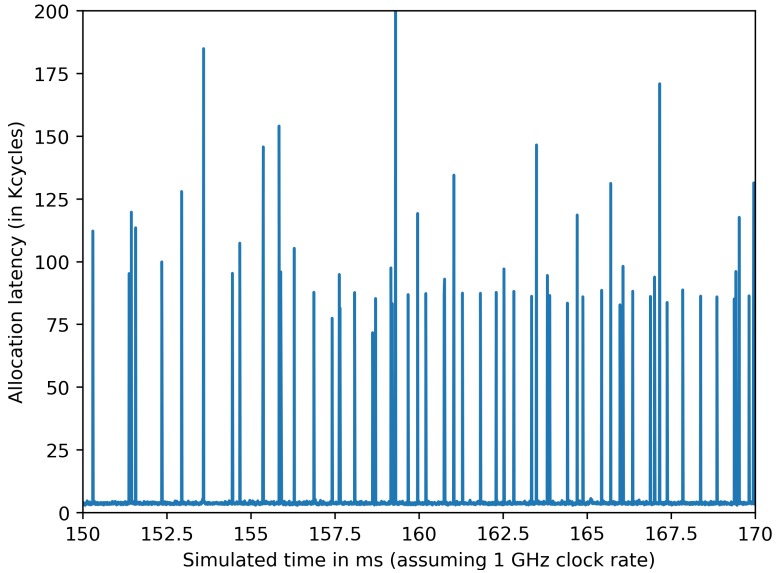


Sampling Rate: 1 KHz

Memory Allocation Latency

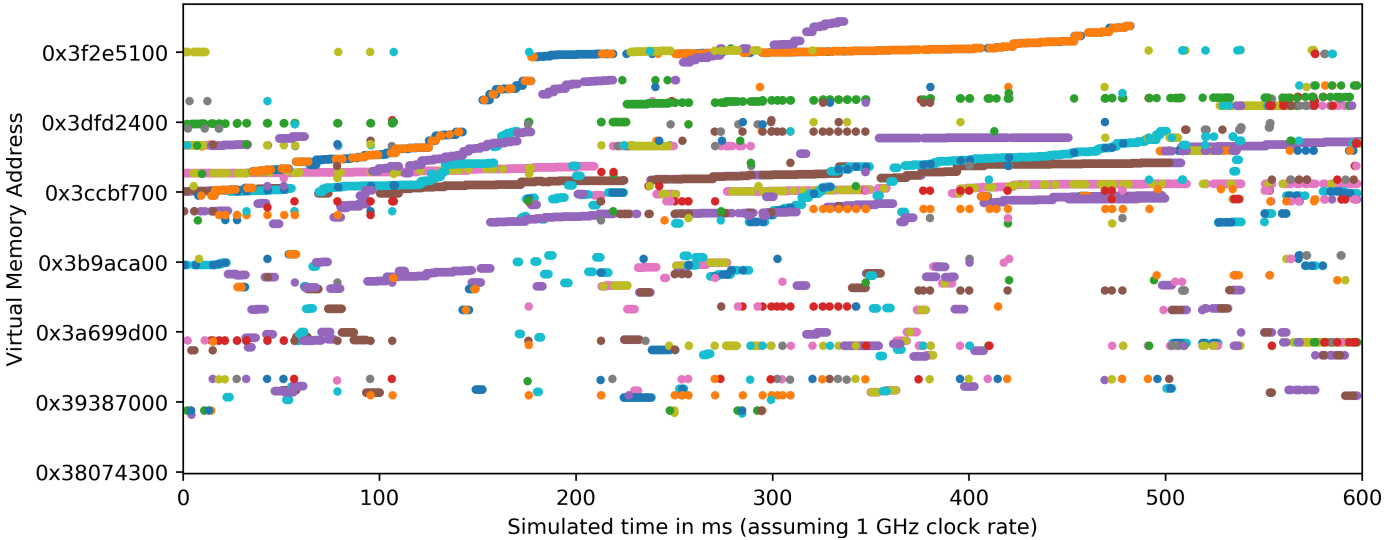


Sampling Rate: 1 KHz



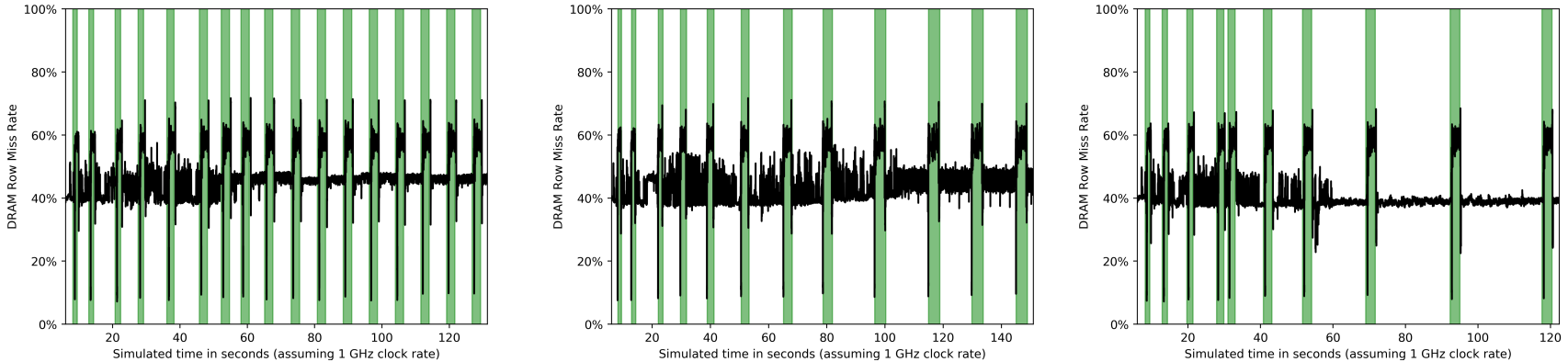
Every Allocation

Logging Memory Allocations



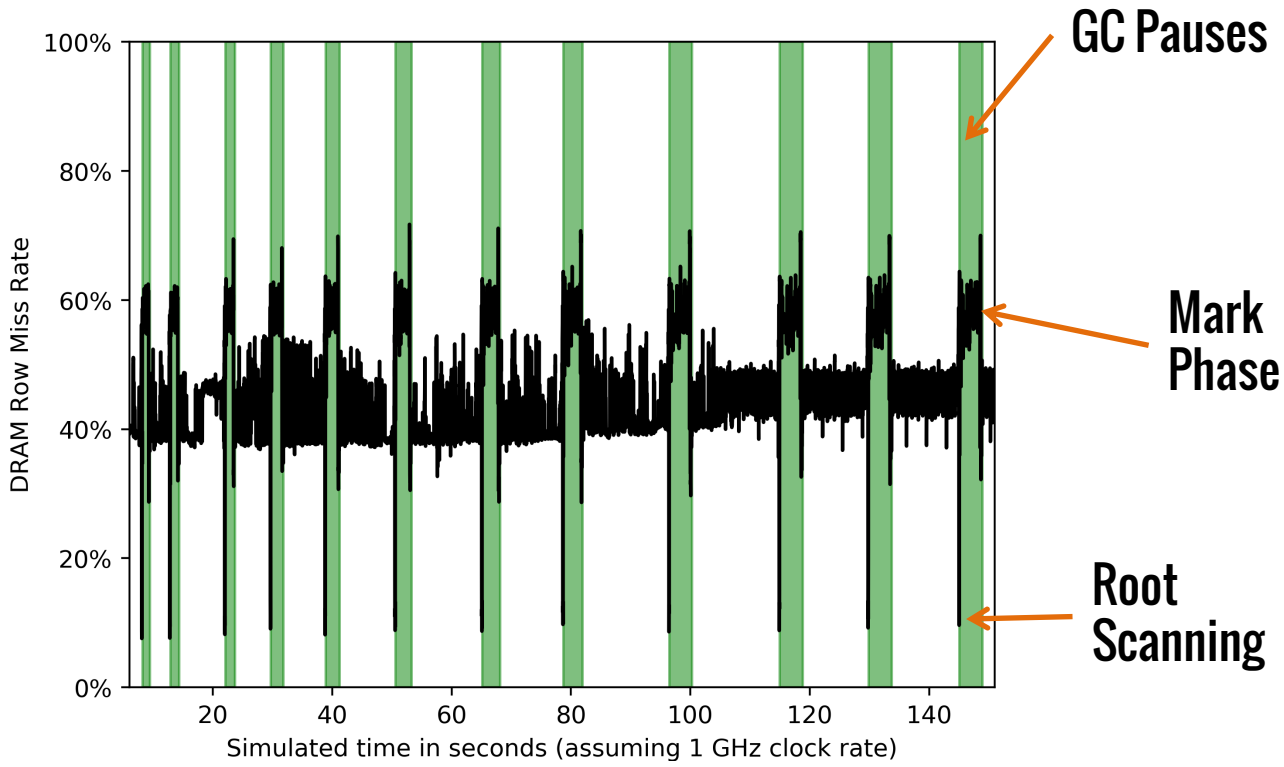
All memory allocations in a program
(color indicates allocation class size)

DRAM Row Misses



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

DRAM Row Misses



Conclusion

Conclusion

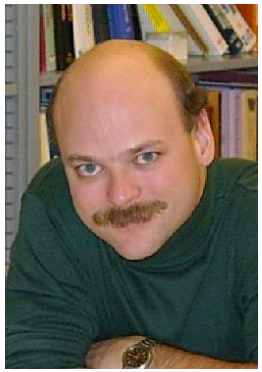
Combining **RISC-V and JikesRVM** enables new hardware-software research that **modifies the hardware, operating system and managed runtime**

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)

Thank you! Any Questions?



Martin Maas **Krste Asanovic** **John Kubiatowicz**

+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.