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

ABSTRACT

Managed languages such as Java, JavaScript or Python account for a large portion of workloads, both in cloud data centers and on mobile devices. It is therefore unsurprising that there is an interest in hardware-software co-design for these languages. However, existing research infrastructure is often unsuitable for this kind of research: managed languages are sensitive to fine-grained interactions that are not captured by high-level architectural models, yet are also too long-running and irregular to be simulated using cycle-accurate software simulators.

Open-source hardware based on the RISC-V ISA provides an opportunity to solve this problem, by running managed workloads on RISC-V systems in FPGA-based full-system simulation. This approach achieves both the accuracy and simulation speeds required for managed workloads, while enabling modification and design-space exploration for the underlying hardware.

A crucial requirement for this hardware-software research is a managed runtime that can be easily modified. The Jikes Research Virtual Machine (JikesRVM) is a Java Virtual Machine that was developed specifically for this purpose, and has become the gold standard in managed-language research. In this paper, we describe our experience of porting JikesRVM to the RISC-V infrastructure. We discuss why this combined setup is necessary, and how it enables hardware-software research for managed languages that was infeasible with previous infrastructure.

1 INTRODUCTION

Managed languages such as Java, JavaScript and Python account for a large portion of workloads [16]. A substantial body of work suggests that managed-language runtimes can significantly benefit from hardware support and hardware-software co-design [10, 13, 21, 22]. However, despite their pervasiveness, these types of workloads are often underrepresented in computer architecture research, and most papers in premier conferences use native workloads such as SPEC CPU to evaluate architectural ideas.

While native workloads represent an important subset of applications, they are not representative of a large fraction of workloads in some of the most important spaces, including cloud and mobile. This disconnect between real-world workloads and evaluation was pointed out in a prominent Communications-of-the-ACM article almost 10 years ago [7], but not much has changed since then. A part of the problem is arguably that there is currently no good way to evaluate managed languages in the context of computer architecture research. Specifically, all of the major approaches fall short when applied to managed-language applications:

- High-level full-system simulators do not provide the fidelity to fully capture managed-language workloads. These workloads often interact at very small time-scales. For example, garbage collectors may introduce small delays of $\approx$ 10 cycles each, scattered through the application [10]. Cumulatively, these delays add up to substantial overheads but individually, they can only be captured with a high-fidelity model.
- Software-based cycle-accurate simulators are too slow for managed workloads. These simulators typically achieve on the order of 400 KIPS [17], or 1s of simulated time per 1.5h of simulation (per core). Managed-language workloads are typically long-running (i.e., a minute and more) and run across a large number of cores, which means that simulating an 8-core workload for 1 minute takes around a month.
- Native workloads often take advantage of sampling-based approaches, or use solutions such as Simpoints [20] to determine regions of interest in workloads and then only simulate those regions. This does not work for managed workloads, as they consist of several components running in parallel and affecting each other, including the garbage collector, JIT compiler and features with dynamically changing state (such as biased locks, inline caching for dynamic dispatch, etc.). In addition, managed application performance is often not dominated by specific kernels or regions of interests, which makes approaches that change between high-level and detailed simulation modes (e.g., MARSSx86 [17], Sniper [9]) unsuitable for many of these workloads.

For these reasons, a large fraction of managed-language research relies on stock hardware for experimentation. While this has enabled a large amount of research on improving garbage collectors, JIT compilers and runtime system abstractions, there has been relatively little research on hardware-software co-design for managed languages. Further, the research that does exist in this area typically explores a single design point, often in the context of a released chip or product, such as Azul's Vega appliance [10]. Architectural design-space exploration is rare, especially in academia.

We believe that easy-to-modify open-source hardware based on the RISC-V ISA, combined with an easy-to-modify managed-language runtime system, can provide an opportunity to address this problem and perform hardware-software research that was infeasible before. Both pieces of infrastructure already exists:

On one hand, the RocketChip SoC generator [5] provides the infrastructure to generate full SoCs that are realistic (i.e., used in products), and can target both ASIC and FPGA flows. Using an FPGA-based simulation framework such as Strober [14] enables *simulating* the performance of real RocketChip SoCs at high-fidelity,

1st Workshop on Computer Architecture Research with RISC-V, 10/14/2017, Boston, MA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

with FPGA frequencies of 30-100 MHz. This means that this infrastructure can achieve the realism, fidelity and simulation speed required to simulate managed-language workloads.

On the other hand, infrastructure exists for managed-language research. Specifically, the Jikes Research Virtual Machine (JikesRVM) is a Java VM geared towards experimentation. JikesRVM is easy to modify, thanks to being written in a high-level language (Java) and using a modular software design that facilitates changing components such as the object layout, GC or JIT passes.

We believe that bringing these two projects together will enable novel hardware-software research. In this paper, we present one important step towards this vision, by porting JikesRVM to RISC-V. We first discuss why such a port is necessary. We then describe the porting effort in detail, in the hope that it will be helpful for others porting managed runtime systems to RISC-V. Finally, we demonstrate the running system, and show the research it enables.

2 BACKGROUND

The shortcomings of existing infrastructure to perform managed-language research have been well-established. For example, Yang et al. demonstrated that sampling Java applications at 100 KHz or less misses important performance characteristics [23].

Another example is a 2005 paper by Hertz and Berger [11]: In order to investigate trade-offs between manual and automatic memory management, the authors had to instrument an existing runtime system to extract allocated memory addresses, and – in a second pass – inject addresses produced by an oracle. The authors found that this was difficult to achieve in software, as the software instrumentation led to a 2-33% perturbation in execution time, which was larger than the effect they were trying to measure. They therefore decided to use a software simulator (Dynamic SimpleScalar [12]) for these experiments. While appropriate in this setting, this approach is often problematic in terms of simulation speed and the reliability of the resulting performance numbers.

To facilitate this type of research, several projects have tried to enable simulation of managed workloads. Zsim [18] enables long-running multi-core workloads by using dynamic instrumentation, but this approach sacrifices accuracy and cannot account for fine-grained interactions such as write-barriers in garbage collectors. Other examples are MARSSx86 [17] and Sniper [9], which are full-system emulators that can fast-forward to regions of interest and then simulate those regions at high fidelity. Both simulators have been used to simulate Java workloads [8, 19]. However, this approach is only appropriate if short, representative regions can be found, and architectural state does not build up slowly. Both are problems for managed workloads, and it is unclear how many managed applications are amenable to this methodology.

We believe that FPGA-based simulators are emerging as the most promising candidate for managed-language research. While these simulators were traditionally constrained by the size of available FPGAs, this has changed in recent years, and there are now large FPGA boards available – even for rent in the public cloud [1] – that can address enough DRAM to run managed workloads on simulated multi-core SoCs. This infrastructure can achieve both high simulation speed and fidelity by using the FPGA to perform cycle-accurate simulation of the on-chip RTL, and using cycle-accurate

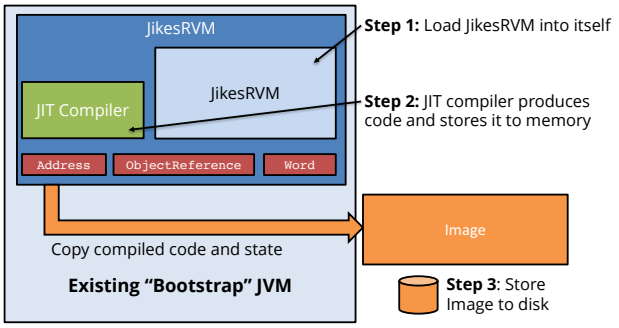


Figure 1: Building the JikesRVM.

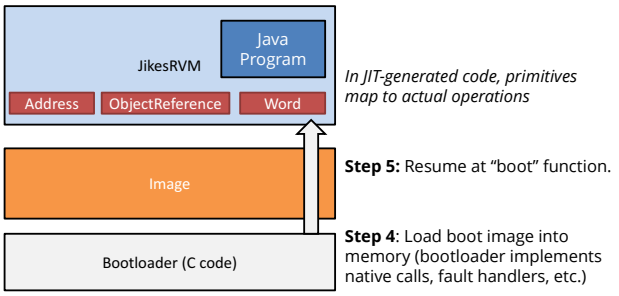


Figure 2: Running the JikesRVM.

timing models for off-chip components such as DRAM (running either on the FPGA or on a host machine). This approach can realistically simulate the performance of an ASIC implementation, and provides a combination of accuracy, simulation speed and modifiability that makes hardware and software co-design feasible.

3 THE JIKES RVM

To experiment with managed runtimes, we require a runtime system that can be easily modified. We picked the Jikes Research VM [4], which is the de facto standard in managed-language research.

Jikes is a VM for Java, and is highly representative of other managed runtime systems. We ported JikesRVM and its non-optimizing *Baseline* JIT compiler to RISC-V. To our knowledge, this results in the first full-system platform for hardware-software research on Java applications, allowing modification of the entire hardware and software stack. In the following section, we describe our port. We particularly focus on aspects that will be useful for authors of future managed-runtime ports for RISC-V.

3.1 Jikes's Software Design

In order to make it easy to modify, JikesRVM embraces object-oriented design principles and is written in Java. This design is often called a *metacircular* runtime system (i.e., a runtime system written in the same language it executes).

This approach introduces new challenges, as Java is not intended for the low-level system programming required to implement a runtime system such as a JVM. Jikes solves this problem by providing a library called *VM Magic*, with classes representing low-level primitives such as pointers (`Address`) or references (`ObjectReference`).

From a Java perspective, these primitives are normal objects with methods such as `Address.loadInt(addr)`. However, Jikes's own JIT compiler detects them and handles them specially.

3.2 Bootstrap Process

JikesRVM requires an existing "bootstrap" JVM, such as OpenJDK's Hotspot JVM (Figure 1). To compile Jikes, it is first loaded into this existing JVM, as a normal Java program where the VM Magic primitives are regular objects with an implementation that emulates their intended behavior. Once JikesRVM runs in the bootstrap VM, it loads an instance of itself, which results in Jikes's own classes being loaded and compiled by Jikes's JIT compiler. However, as this is now Jikes's JIT (and not the bootstrap VM's), it will detect calls to VM Magic classes and replace them with the actual code executing low-level operations, such as memory stores.

In a final step, the instantiated objects belonging to the nested JikesRVM instance (including their JIT-compiled code) are taken and copied into an image, which is then stored to disk. This image now contains compiled code for all of Jikes's core classes, which can be executed without the bootstrap JVM in place.

3.3 Running Jikes RVM

Once Jikes has been compiled, it can be run by executing a small bootloader program (written in C), which takes the image generated during the boostrapping process and maps it into its address space (Figure 2). This part of the address space represents the initial heap that the JVM is executing on. The bootloader then sets up the Java stack and jumps into a boot function which initializes the different components of the JVM. This process involves many steps and requires loading and executing initializers for 93 classes.

Once the JVM has booted up, it parses the command line arguments, uses them to determine a `.jar` or `.class` file to load, and then jumps into the main function of the program.

4 PORTING THE JIKES RVM TO RISC-V

Porting JikesRVM to a new ISA is complicated by Jikes's metacircular nature. Fortunately, the JVM already supports two ISAs (x86 and PowerPC), and therefore has infrastructure in place to factor out ISA-specific portions of code (such as the assembler, compiler, native-function interface, stack walker, etc.). Porting JikesRVM therefore primarily required creating RISC-V implementations of these different components. Overall, our port involved modifications to 86 files and added around 15,000 lines of code.

4.1 Bringing up the Environment

The first step in porting JikesRVM was to bring up an environment that contains all the dependencies required by JikesRVM. Specifically, this included a Linux distribution with a basic set of tools and libraries, including `glibc`, `bash`, etc. JikesRVM also requires compiling the GNU Classpath class library for a RISC-V target, which has further dependencies on various different libraries.

To facilitate building these different dependencies, we rely on a RISC-V port of the Yocto Linux distribution generator [3]. Yocto provides an environment that can cross-compile the Linux kernel and a range of packages on a host system, and generates an image that can then booted in a RISC-V emulator or on actual RISC-V hardware. We used Yocto to generate an image which we then use as the environment to run JikesRVM within `riscv-qemu`.

```
    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)
            elif name in fcmp and r == 'rd':
                    funargs.append('GPR ' + r)
            elif name in ['frflags', 'fsflags', 'fsflagsi']:
                    funargs.append('GPR ' + r)
            else:
                    funargs.append('FPR ' + r)
    else:
            funargs.append('GPR ' + r)
```

```
  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);
  }
```

Figure 3: Part of the Python script that auto-generates the assembler, and the code that it generates.

In addition to generating the image, targeting JikesRVM to RISC-V also required us to have the cross-compiler and libraries available during the build process, to compile components such as the bootloader or the C libraries backing GNU Classpath. Yocto facilitates this by creating an *SDK*, which is a package that includes the entire cross-compile toolchain and development packages such as common libraries or `autoconf`. This SDK can be installed on any machine, and contains a script that adds the cross-compilers to the current command-line environment. Using a Yocto SDK provides us with all the tools and libraries we need to build Jikes, without building a full RISC-V development environment.

4.2 Debugging Infrastructure

To achieve a fast compile loop, we used a Python script that cross-compiles JikesRVM on the host system, copies the output into the Yocto-generated image and runs this image in QEMU. We also modified the image with a custom `/etc/inittab` script that launches JikesRVM, pipes the output into a file and then shuts down the QEMU instance. This gives us a fast turnaround for debugging.

After setting up the scripts, the next step consisted of porting JikesRVM's bootloader code. The code only includes a small number of architecture-dependent portions, specifically the assembly code that sets up the Java stack and jumps into a Java function.

Once this step was completed, the next task was to port the JIT compiler. To do this incrementally, we added test code at the beginning of the JVM's boot function (`VM.boot()`), which is the first function the bootloader jumps into after setting up the stack. This allowed us to first implement simple Java opcodes such as integer operations, and then build up from there.

4.3 Porting the Assembler

Before we could start porting the JIT compiler, we had to implement an assembler that can generate RISC-V instructions. While Jikes's assemblers for PPC and x86 are hand-written, we were able to automate this process for RISC-V, thanks to the `riscv-opcodes` repository [2]. This repository provides a machine-readable version of all RISC-V instructions. Building on a Python script that is available as part of `riscv-opcodes`, we generated most of the assembler automatically, creating an `emitX()` function for every instruction `X` in the instruction set (Figure 3).

One case that needed special attention were branches. The JIT compiler often generates branches with placeholders for the target offset, which are rewritten at a later point. In RISC-V, we had to be careful to ensure to distinguish between short branches (that fit into the branch instruction's 12-bit offset) and general branches, for which we need to emit a branch followed by a `jal` instruction. The assembler provides functions to emit both types of branches. If the target is unknown in advance, a general branch is emitted.

4.4 Porting the JIT Compiler

The non-optimizing JIT compiler contains a set of functions corresponding to Java bytecode instructions. Each of these functions calls into the assembler to emit a RISC-V instruction sequence that implements the specific Java opcode. The JIT compiler also provides instruction sequences for the VM Magic functions described in Section 3.1. Finally, the JIT compiler provides functions that emit code for special cases, such as prologues, epilogues and yield points (yield points are emitted at certain points throughout the program and check whether a thread is supposed to block – e.g., because of garbage collection or revoking a biased lock).

We started by implementing prologues, epilogues and several basic integer instructions. This allowed us to run small test programs by injecting them into Jikes's boot function. However, for programs that were more complex, we required more information to debug the execution. As JIT-compiled code does not provide symbol tables, it is difficult to debug this code with traditional debuggers such as GDB. We therefore chose a different approach.

We instrumented the JIT compiler to emit a trace of its execution. For each executed opcode, we print the name of the opcode, the corresponding instruction sequence, and the current state (the top of the stack). We achieve this by prefixing the instruction sequence for each opcode with an invalid load that will trigger a `SEGFAULT`. Additionally, we also include auxiliary information:

```
0x...000: LD X0, 1024(X0) # SEGFAULT
0x...004: (Number of instructions)
0x...008: (Opcode)
0x...00c: (Stack Offset)
```

When the load is reached, it will trigger an exception that can be caught in the bootloader program. The bootloader then reads the auxiliary information and outputs the desired debug information, including a disassembled version of the instructions associated with this bytecode (Figure 4). Note that we did not have to write our own disassembler for this. Instead, we simply printed `DASM(INST)` to the command line, and piped the result through the `spike-dasm` program that ships with the Spike ISA simulator.

```
---- 0x35420004 (Opcode: getstatic, Stack: 72)
|-> 0x32fcecd0 [72] ->  0x0 0x0 0x32f7dda8 0x200002e9a8 0x32fced88 0x3541d7e8
  0x35420014        lui     t0, 0x1
  0x35420018        xori    t0, t0, 368
  0x3542001c        add     t0, t0, gp
  0x35420020        ld      t0, 0(t0)
  0x35420024        sd      t0, 64(sp)
---- 0x35420028 (Opcode: iconst_1, Stack: 64)
|-> 0x32fcecd0 [64] ->  0x330046c8 0x0 0x0 0x32f7dda8 0x200002e9a8 0x32fced88
  0x35420038        li      t0, 1
  0x3542003c        sw      t0, 60(sp)
---- 0x35420040 (Opcode: iconst_1, Stack: 56)
|-> 0x32fcecd0 [56] ->  0x100000000 0x330046c8 0x0 0x0 0x32f7dda8 0x200002e9a8
  0x35420050        li      t0, 1
  0x35420054        sw      t0, 52(sp)
```

Figure 4: Debug output for the JIT compiler.

As the test programs grew, we found that the debug output became too cumbersome to work with. We therefore added a modification to JikesRVM which allows us to only selectively inject this instrumentation. Specifically, we added a `@SoftwareBreakpoints` annotation which can be attached to any function in JikesRVM. If this annotation is present, the instrumentation code will be injected by the JIT compiler (and we will get a trace of its execution), otherwise the function will be compiled normally.

4.5 Foreign-Function Calls

One of the most challenging aspects of porting Jikes was to support foreign-function calls. Jikes provides two mechanisms to call into C code: JNI calls (which is Java's mechanism to call into C functions) and a simpler mechanism named *syscalls*. JNI is a complex mechanism that enables calls in both directions (C to Java and Java to C). This makes it possible to have a mix of both Java and C stack-frames co-exist on the same stack. Jikes therefore needs to be able to unwind both types of frames for delivering exceptions, and scan them for spilled pointers at the beginning of GC passes. This means that JNI calls require maintaining a side table of pointers (for stack scanning), check for yield points when crossing a language barrier, and support the full C calling convention, including varargs.

Avoiding this complexity, Jikes's syscalls mechanism is intended to implement simple functions such as writing bytes to a stream or executing math functions like `sqrt`. Instead of supporting the full calling convention, it only supports simple calls, does not check for yield points and cannot call back into Java. For debugging purposes, we found it important to implement syscalls early, but JNI functions require a large amount of work and we decided to leave them to the end. Syscalls are emitted by the JIT, while JNI calls are generated by a special `JNICompiler`. Implementing syscalls is sufficient to run test programs with simple command line output.

4.6 Exceptions & Run-time Checks

Java checks for a number of corner cases and triggers exceptions if necessary (e.g., array bounds checks or divide-by-zero checks). We found that the best approach for this in RISC-V was to trigger exceptions through loads to invalid addresses. This causes execution to drop back into the bootloader, where we can determine which exception was triggered (based on the failing instruction) and then jump into a Java function that delivers the exception and unwinds the stack. The exception delivery itself requires architecture-specific code for unwinding both Java and JNI (native) stack-frames.

4.7 Additional Features

While the features discussed so far enable increasingly large test programs (and executing a large part of the `VM.boot` function), completing the full boot sequence requires a large number of architecture-specific features, including locks, lazy compilation trampolines, dynamic bridges (which are necessary for JIT-compiling a function by running the JIT on the same stack, and then transparently transferring execution to the JITed code) and implementations of interface method tables (which require synthesizing architecture-specific code to traverse a search tree). To complete the boot sequence and run real programs, all of these features are required.

4.8 Summary

Our port of the baseline compiler is complete in that it passes all of the unit tests that are part of JikesRVM and runs the subset of Dacapo [6] benchmarks that are supported by our version of JikesRVM (avrora, luindex, lusearch, pmd, sunflow, xalan). We successfully ran these benchmarks both in simulation, and on a RocketChip instance mapped to an FPGA board.

Figure 5 shows one of these benchmarks running on RISC-V (captured from an FPGA setup). Being able to run Dacapo benchmarks gives us a high degree of confidence in the correctness of our port, as the Dacapo suite consists of large and complex benchmarks. For example, the benchmarks presented here include a raytracer, the Lucene search engine, and a code analyzer.

5 EVALUATION

With a complete port of JikesRVM, we now have the ability to run Java workloads on RISC-V systems and modify both the JVM and the underlying hardware. To demonstrate this experimental setup, we ran JikesRVM on RocketChip in FPGA-based simulation. We use Xilinx ZC706 development boards, which are comprised of an Zynq XC7Z045 FPGA with 8 GiB of fabric-attached DRAM. We use an FPGA-based simulation framework similar to Strober [14], with timing models to simulate DRAM accesses. This setup simulates a single Rocket 5-stage in-order CPU, with 16 KiB L1 instruction and data caches and a simulated 1 MiB L2 cache.

Using this setup, we achieved effective simulation speeds of 10 MIPS and more. We simulate a design with an operating frequency of 1 GHz, a L2 latency of 23 cycles and a DRAM latency of 80 cycles. Running these experiments for the Dacapo benchmarks (default input size) allowed us to collect performance data, as well as instruction counts and other metrics on our platform. Executing the full set of benchmarks takes over 1.2 trillion instructions, which would take 35 days if simulated at 400 KIPS.

The following table presents the number of dynamic instructions for each of the benchmarks, as well as their simulated runtime:

Benchmarks	Instructions (B)	Runtime (s)
avrora	118.0	311.8
luindex	47.7	103.5
lusearch	263.5	597.2
pmd	158.5	346.8
sunflow	504.8	1,352.9
xalan	190.8	466.4

```
Setting default thread count for MMTk to minimum of default thread count 1 and
 maximal thread count 2147483647 supported by current GC plan.
New default thread count value is 1
Setting actual thread count for MMTk to minimum of desired thread count 1 and
maximal thread count 2147483647 supported by current GC plan.
New actual thread count is 1
[GC 1 Start 33.97 s   20480KB -> 12424KB   11825.91 ms]
[GC 2 Start 73.41 s   24576KB -> 17136KB   13466.31 ms]
[GC 3 Start 137.59 s   31740KB -> 20040KB   15324.46 ms]
[GC 4 Start 176.83 s   27688KB -> 20180KB   15999.70 ms]
===== DaCapo head-runknown avrora starting =====
[GC 5 Start 241.26 s   37888KB -> 30732KB   19705.49 ms]
[GC 6 Start 319.30 s   50176KB -> 36896KB   22377.38 ms]
[GC 7 Start 422.90 s   63488KB -> 45776KB   26419.36 ms]
[GC 8 Start 854.55 s   78848KB -> 47232KB   30183.02 ms]
[GC 9 Start 1941.52 s   79872KB -> 43472KB   30285.10 ms]
[GC 10 Start 3053.29 s   79872KB -> 43648KB   31433.10 ms]
===== DaCapo head-runknown avrora PASSED in 2849150 msec =====
[End 3084.72 s]
```

Figure 5: Output of JikesRVM running one of the Dacapo benchmarks, with verbose GC output (processed log file).

JikesRVM is configured to use the *Mark & Sweep* garbage collector. With a 100MB maximum heap size, the JVM spends the following fraction of time in GC for each benchmark:

Benchmarks	GC Pauses	Time in GC
avrora	10	6%
luindex	8	13%
lusearch	90	35%
pmd	26	30%
sunflow	52	9%
xalan	39	27%

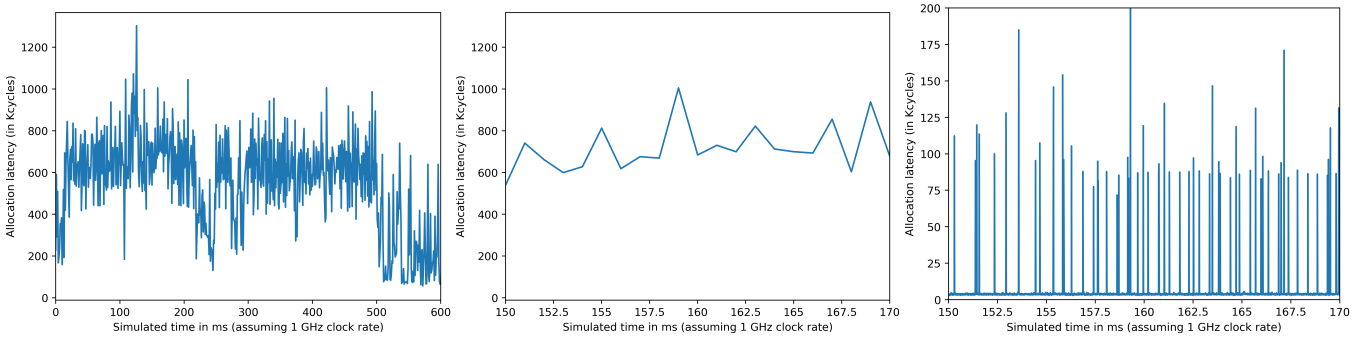
While the JVM's performance can be improved substantially, note that the baseline compiler's primary responsibility is to run code that executes rarely. In order to generate performance-competitive code, we need to port the optimizing JIT compiler as well (for which the baseline compiler is a prerequisite).

6 RESEARCH CASE STUDY

We believe that FPGA-based full-system simulation of JikesRVM workloads on RISC-V hardware enables studies that are difficult to perform in a traditional setup. Specifically, we can modify the software stack as well as the underlying hardware, while collecting accurate numbers that can capture fine-grained interactions for full workload executions with short simulation times. This enables design-space explorations that modify both hardware and software layers, and detailed instrumentation of the entire system.

To demonstrate these capabilities, we conduct a study that addresses challenges similar to those described by Hertz and Berger's paper [11] from Section 2. Many interactions within managed runtime systems are fine-grained and therefore difficult to measure. One example of these interactions are memory allocations, which occur frequently but complete quickly most of the time. We are often interested in the causes of long allocations.

To record these allocations in a traditional system, we would have two options: we could either use an instrumentation-based approach or a sampling-based approach. However, the former introduces observer effects and perturbs the execution time, while the latter traditionally achieves low sampling frequencies and can hide important details. Figure 6 shows an example of this: While



(a) 600ms of execution sampled at 1 KHz **(b) 20ms slice sampled at 1 KHz** **(c) The same 20ms in FPGA simulation**

Figure 6: Contrasting an approach that samples time spent in allocation at a 1 KHz-granularity with recording every allocation in hardware without introducing observer effects to the application (numbers are from the `pmd` Dacapo benchmark). Results were collected in a single run and aggregated to demonstrate the effect of different sampling rates.

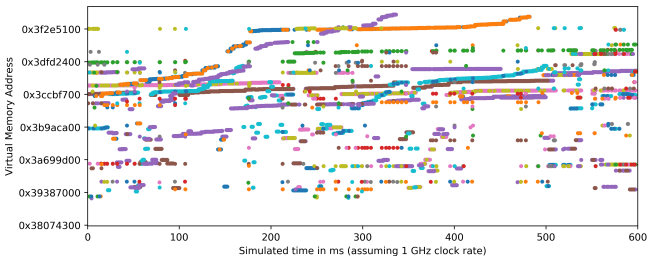


Figure 7: Addresses returned by the JikesRVM free-list allocator. Colors indicate the allocation size class.

sampling at 1 KHz helps us understand the macro-behavior of the application (Figure 6a), it does not tell us about individual allocations (Figure 6b). To gain real insight into the behavior of the memory allocator, we need to be able to record every single allocation latency – without perturbing the execution time.

One approach to this problem would be to use a system like SHIM [23], which enables high-resolution sampling while minimizing the observer effect. It achieves this by running an observer thread in a second hardware context on an SMT-enabled multiprocessor, sampling at a resolution of 1,200 cycles at only 2% overhead. SHIM can also perform measurements at an extremely fine-grained solution of ≈ 15 cycles, but then the perturbation becomes large, at an overhead of 61%. While this is sufficient to understand program behavior, it is limited to counters exposed by the hardware. Specifically, SHIM is designed for existing SMT processors and cannot instrument arbitrary signals in a modifiable hardware design.

Infrastructure such as JikesRVM running on RocketChip enables us to do this: by adding on-chip buffers to the RTL design, we can record every allocation in the execution of the program, and produce a detailed trace without perturbing the execution time. Specifically, we record the start and end time, as well as the size class and memory address associated with every allocation.

Figure 6c shows the result: by looking at the duration of every allocation, we can see that most allocations complete in a small amount of time (≈ 4,000 cycles), while some allocations take 10-100 × longer. This gives us insight into the behavior of the memory allocator (in this case, a segregated free-list allocator). In the common case, the memory allocator consumes a set of per-size-class free lists, and completes quickly if a block is available on the list. If not, the allocator has to remove a new block from the global free list, zero the block's memory, and create a new free list.

There are other insights that can be gained from this trace as well. For example, looking at allocations for the same size class and counting how many of them use the fast path, we can deduce the amount of memory fragmentation. We can also analyze locality: looking at the addresses that are returned by the allocator (Figure 7), we see that subsequent allocations to the same size class are typically contiguous, but overall locality is low. This confirms that segregated free-list allocators produce poor locality.

Memory allocators are only one example of experiments that are possible with this infrastructure, and we believe that it will open up new research directions in a wide range of areas. One specific area that we are targeting is hardware support for garbage collection, and we are in the process of building a prototype system [15].

7 CONCLUSION

In this paper, we presented our port of JikesRVM to RISC-V, and demonstrate it running on FPGA-based RISC-V hardware. We believe that the combination of a managed-runtime system and hardware that can be easily modified will enable new kinds of hardware-software research, as demonstrated by our case study.

ACKNOWLEDGEMENTS

We want to thank David Biancolin, Chris Celio, Donggyu Kim, Jack Koenig and Sagar Karandikar for running the FPGA experiments, and for their work on implementing the FPGA-based simulation infrastructure that these experiments are based on. Thanks is also owed to Palmer Dabbelt, Howard Mao and Andrew Waterman for their help with the RISC-V infrastructure while working on the port. We also want to thank Xi Yang for feedback on an earlier draft of this paper. This research was partially funded by DOE grant #DE-AC02-05CH11231, the STARnet Center for Future Architecture Research (C-FAR), and ASPIRE Lab sponsors and affiliates Intel, Google, HPE, Huawei, and NVIDIA.

REFERENCES

[1] 2017. Amazon EC2 F1 Instances. (2017). http://aws.amazon.com/ec2/instance-types/f1/

[2] 2017. riscv-opcodes: RISC-V Opcodes. (Aug. 2017). https://github.com/riscv/riscv-opcodes original-date: 2011-08-30T19:14:12Z.

[3] 2017. riscv-poky: Port of the Yocto Project to the RISC-V ISA. (Aug. 2017). https://github.com/riscv/riscv-poky original-date: 2014-10-15T22:40:26Z.

[4] B. Alpern, S. Augart, S.M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K.S. McKinley, M. Mergen, J. E B Moss, T. Ngo, V. Sarkar, and M. Trapp. 2005. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal* 44, 2 (2005), 399–417. https://doi.org/10.1147/sj.442.0399

[5] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[6] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. https://doi.org/10.1145/1167473.1167488

[7] 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 Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2008. Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century. *Commun. ACM* 51, 8 (Aug. 2008), 83–89. https://doi.org/10.1145/1378704.1378723

[8] Benjamin C. Lee. 2016. Datacenter Design and Management: A Computer Architect's Perspective. 11 (02 2016), 1–121.

[9] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. ACM, New York, NY, USA, Article 52, 12 pages. https://doi.org/10.1145/2063384.2063454

[10] Cliff Click, Gil Tene, and Michael Wolf. 2005. The Pauseless GC Algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE '05)*. ACM, New York, NY, USA, 46–56. https://doi.org/10.1145/1064979.1064988

[11] Matthew Hertz and Emery D. Berger. 2005. Quantifying the Performance of Garbage Collection vs. Explicit Memory Management. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 313–326. https://doi.org/10.1145/1094811.1094836

[12] Xianglong Huang, J Eliot B Moss, Kathryn S McKinley, Steve Blackburn, and Doug Burger. 2003. Dynamic simplescalar: Simulating java virtual machines. *University of Texas at Austin, Department of Computer Sciences, Technical Report TR-03-03* (2003).

[13] José A. Joao, Onur Mutlu, and Yale N. Patt. 2009. Flexible Reference-counting-based Hardware Acceleration for Garbage Collection. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 418–428. https://doi.org/10.1145/1555754.1555806

[14] Donggyu Kim, Adam Izraelevitz, Christopher Celio, Hokeun Kim, Brian Zimmer, Yunsup Lee, Jonathan Bachrach, and Krste Asanović. 2016. Strober: Fast and Accurate Sample-based Energy Simulation for Arbitrary RTL. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 128–139. https://doi.org/10.1109/ISCA.2016.21

[15] Martin Maas, Krste Asanovic, and John Kubiatowicz. 2016. Grail Quest: A New Proposal for Hardware-assisted Garbage Collection. In *Sixth Workshop on Architectures and Systems for Big Data (ASBD 2016)*. http://acs.ict.ac.cn/asbd2016/Papers/ASBD2016_paper_3.pdf

[16] Leo A. Meyerovich and Ariel S. Rabkin. 2013. Empirical Analysis of Programming Language Adoption *(OOPSLA '13)*. ACM, New York, NY, USA, 1–18. https://doi.org/10.1145/2509136.2509515

[17] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. 2011. MARSSx86: A Full System Simulator for x86 CPUs *(DAC'11)*.

[18] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 475–486. https://doi.org/10.1145/2485922.2485963

[19] Jennifer B. Sartor, Wim Heirman, Stephen M. Blackburn, Lieven Eeckhout, and Kathryn S. McKinley. 2014. Cooperative Cache Scrubbing. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2628071.2628083

[20] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. ACM, New York, NY, USA, 45–57. https://doi.org/10.1145/605397.605403

[21] David Ungar, Ricki Blau, Peter Foley, Dain Samples, and David Patterson. 1984. Architecture of SOAR: Smalltalk on a RISC. In *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA '84)*. ACM, New York, NY, USA, 188–197. https://doi.org/10.1145/800015.808182

[22] Greg Wright, Matthew L. Seidl, and Mario Wolczko. 2005. *An Object-aware Memory Architecture.* Technical Report. Sun Microsystems, Inc., Mountain View, CA, USA.

[23] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. 2015. Computer Performance Microscopy with SHIM. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 170–184. https://doi.org/10.1145/2749469.2750401