Dynamic SimpleScalar: Simulating Java Virtual Machines

Xianglong Huang[*] J. Eliot B. Moss[†] Kathryn S. McKinley[*] Steve Blackburn[‡] Doug Burger[*]

[*]Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
{xlhuang, mckinley, dburger}
@cs.utexas.edu

[†]Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003
moss@cs.umass.edu

[‡]Department of Computer Science
Australian National University
Canberra, ACT, 0200, Australia
Steve.Blackburn@anu.edu.au

ABSTRACT

Current user-mode machine simulators typically do not support simulation of dynamic compilation, threads, or garbage collection, all of which Java Virtual Machines (JVMs) require. In this paper, we describe, evaluate, and validate *Dynamic SimpleScalar* (DSS). DSS is a tool that simulates Java programs running on a JVM, using just-in-time compilation, executing on a simulated multi-way issue, out-of-order execution superscalar processor with a sophisticated memory system. We describe the implementation of the minimal support necessary for simulating a JVM in SimpleScalar, including signals, thread scheduling, synchronization, and dynamic code generation, all required by a JVM. We validate our simulator using IBM Research's Jikes RVM, a state-of-the-art JVM that runs on a PowerPC architecture, and show that DSS loyally reflects the performance trends of a real JVM system. We then present a set of results using DSS. On the SPECjvm98 benchmarks, we study the best heap size for three different copying garbage collectors, and measure total, mutator, and collector memory characteristics. We compare our results with previous work, pointing out new insights, differences, and similarities. For example, we show there is a trade off between the locality benefits of copying collectors and the time to collect.

Submitting to the First Annual IEEE/ACM International Symposium On Code Generation and Optimization.
This work is supported by NSF ITR grant CCR-0085792, and DARPA grants F30602-98-1-0101 and F33615-01-C-1892, and International Business Machines Corporation.

1. Introduction

The Java programming environment is appealing because it supports dynamic compilation, is object-oriented, has automatic memory management, and is platform independent. However, the performance of Java currently lags behind more traditional languages such as C and C++. To mitigate Java's performance losses, research in Java hardware support is necessary, as well as understanding and tuning the low-level behavior of the run-time system. For example, further innovation requires that we understand variation in performance across architectural configurations in the presence of different garbage collection algorithms, dynamic code generation, and synchronization.

Simulation is now widely used to perform research on hardware and low-level software support for many different applications. Unfortunately, no widely available, public tools currently provide detailed, cycle-accurate hardware simulation of a Java Virtual Machine, with support for dynamic compilation. Previous work on characterizing and simulating Java programs [14, 15, 20] either used tools that did not provide cycle-level results, were proprietary and therefore unavailable, or lacked key software functionality necessary for comprehensive research in this area. For example, Li *et al.*'s work [15], which used SimOS, did not employ detailed, cycle-level simulation. Although tools such as SimOS, coupled with a detailed architecture model provide this functionality, including all of the operating system behavior is often too expensive and unnecessary. Section 3 discusses these issues.

In this paper, we describe a series of major extensions to the popular SimpleScalar [8] tools. These extensions permit simulation of a full Java run-time environment on a detailed simulated hardware platform. The Java system that runs on our simulated machine is the IBM Jikes RVM system, which provides an aggressive optimizing compiler, including adaptive and just-in-time compilation, and which is itself implemented in Java [1, 2]. The Jikes RVM produces PowerPC or x86 instructions. The Jikes RVM system requires support for multithreading, dynamic code generation, and hardware and software exceptions (Unix signals).

Our new tool, called Dynamic SimpleScalar (DSS), implements support for dynamic code generation, thread scheduling and synchronization, as well as a general signal mechanism that supports exception delivery and recovery. The target microarchitecture modeled by SimpleScalar is quite different from that of our host PowerPC platform, so absolute performance results differ significantly, and a cycle-accurate validation is not possible. However, we show that the functionally complete JVM incorporated into DSS achieves results that closely follow execution performance trends for a set of Java programs from SPECjvm98. Our validation shows that once we normalize for the performance variation caused by the microarchitectural differences, the differences in performance trends do not exceed 5.63%, and average under 1.33%, even though the performance of both systems varies by as much as 386% due to changes in the heap size. These results increase confidence that our methods for incorporating a run-time Java system (with dynamic compilation) into SimpleScalar capture the extensions' behavior accurately.

We then present performance results obtained with the SPECjvm98 benchmarks using DSS. We determine the best heap size for 3 copying collectors (semi-space, fixed-nursery generational, and an Appel-style flexible-nursery generational [4]) and explore the trade-off between the frequency of garbage collection (GC) and the locality benefits of copying. We compare the effect of the collectors on program behavior and quantify the contributions from the mutator and collector phases. (This experiment is not possible using performance counters since applications cannot turn them on and off as currently implemented [3, 10]) We find the mutator cycles and memory behavior of Java programs are barely affected by the choice of copying collector or heap size. The choice of collector and heap size, however, dramatically affects total performance. The differences among different collectors is the number of times GC is triggered during program execution. The cache miss rates of the different copying collectors are similar across heap sizes. These results differ from the study by Kim et al. [14], which showed that the results for mark-sweep collectors vary with heap size.

The remainder of the paper is organized as follows. We first present background on the Jikes RVM. Then we discuss the features of several current simulators and compare these features to the requirements of simulating the Jikes RVM. In the simulator extensions section, we describe the original SimpleScalar and the extensions we implement, such as the support for signals, thread scheduling and synchronization, and support for dynamic code generation. In our validation section, we compare running SPECjvm98 on DSS against executing the benchmarks directly on a PowerPC machine. Section 6 gives a sample application of our simulator that explores the relationship between heap size and total performance with a semi-space collector, a fixed-nursery generational collector, and an Appel-style flexible-nursery generational garbage collector. Section 7 summarizes our contributions on how to build a simulator for JVMs and our initial results from using this tool.

2. The Jikes RVM

In this section, we describe the basic structure of the Jikes RVM and explain its requirements that prevent the unmodified SimpleScalar tools from simulating Java run-time systems.

The Jikes RVM is a virtual machine for Java programs written in Java and developed by IBM's T. J. Watson Research Center. Since the Jikes RVM was designed for research and experimentation, particular attention was given to making it modular and easily extensible. Its dynamic optimizing compiler exploits modern hardware features to generate high-quality machine code. Its adaptive compiler uses sampling to trigger selective optimization of frequently executed methods. Because the Jikes RVM is written in Java, compiling the optimizing compiler is optionally part of running the system.

A running Java program involves four layers of functionality: the user code, the virtual machine, the operating system, and the hardware. By moving the boundary between Java and non-Java below the virtual machine rather than above it, the Jikes RVM reduces the boundary-crossing overhead and opens up more opportunities for optimizations. These and other advanced techniques used in the Jikes RVM, such as dynamic optimization in its compiler, make its performance competitive with top commercial systems.

To expose issues of simulating the Jikes RVM and identify new features we add to SimpleScalar, we briefly discuss each of five key components of the Jikes RVM: an object model, the run-time subsystem, the thread and synchronization subsystem, the memory management subsystem, and the compiler subsystem [2, 1].

2.1 The Object Model and Memory Layout

The object model provides fast access to fields and array elements, fast virtual method dispatch, and memory protection. For example, dereferencing a null pointer results in referring a Unix `segv` signal to the user program. The Jikes RVM intercepts this signal and raises a Java null pointer exception. The SimpleScalar tools do not support a memory protection model, nor do they check for violations.

2.2 Run-time Subsystem

Run-time services include (Java) exception handling, dynamic type checking, dynamic class loading, interface invocation, and I/O. They are usually implemented in *native* methods written in C, C++, or assembly. In the Jikes RVM, these services are implemented primarily in Java, but also rely on the signal mechanism of the OS (e.g. the Jikes RVM uses a software exception to handle an array bounds check failure). To simulate these features, we added simulation of signals to SimpleScalar.

2.3 Thread and synchronization subsystem

Simulation of the thread and synchronization subsystem in the Jikes RVM require several changes to SimpleScalar.

Thread Scheduling

The Jikes RVM uses Posix kernel-level threads (pthreads), which are light-weight processes, to do scheduling among CPUs. The operating system scheduler takes care of the scheduling of pthreads. The Jikes RVM usually maps one pthread to each CPU, and does its own multiplexing of many Java threads onto what is typically a smaller number of pthreads. Since SimpleScalar is a uniprocessor simulator, we limit the number of CPUs to one, thus scheduling all Java threads on one CPU. With this technique, we can simulate multi-threaded Java programs running on one processor.

The Jikes RVM uses simple time slicing within each pthread to schedule the Java threads assigned to that pthread/CPU. Therefore the Jikes RVM thread scheduler requires timer signals to induce Java thread switches. After the timer is initialized, the scheduler proceeds as follows:

1. A timer expires at a regular interval and generates a timer signal, which is sent to the JVM.
2. When the Jikes RVM C signal handler catches the timer signal, it sets a special bit in the running thread's state that indicates the end of the time-slice.
3. At compile time, the Jikes RVM compiler inserts frequent tests of this bit at *safe points*, i.e., points where thread switching will preserve program semantics.
4. If a check occurs when the bit is set, the code invokes the scheduler, which selects and runs a new thread and resets the bit.

In DSS, we incorporated an interval timer and its signal to support the Jikes RVM thread scheduling scheme, as we discuss in Section 4.

Locks

To support system and user synchronization, the Jikes RVM uses three kinds of locks: *processor locks*, *thin* locks, and *thick* locks. All three locks are built using the `lwarx` and `stwcx` instructions. These instructions perform an atomic read-modify-write operation to storage. The program first issues a `lwarx` instruction to reserve the memory address, and then a `stwcx` instruction attempts to store possibly new data to the address. If the store succeeds, then no other processor or mechanism has modified the target memory location between the time the `lwarx` instruction is executed and the time the `stwcx` instruction completes. If the store fails, then generally the software retries the operation. We added these instructions to DSS, since they are not supported in SimpleScalar.

2.4 Compiler Subsystem

The Jikes RVM has three compilers: the *baseline* compiler, which essentially macro-expands each byte code using an explicit model in memory of the Java evaluation stack, and hence does not generate high-quality code; the *optimizing* compiler, which applies traditional static compiler optimizations as well as a number of optimizations specific to object-oriented features and the dynamic Java context; and the *adaptive* compiler, which first applies the baseline compiler and then, using dynamic measurements of frequency of execution of code, selects methods for optimizing compilation. All the compilers generate machine code at run time, which requires DSS to support dynamic code generation. This feature requires us either to predecode dynamically, or to eliminate predecoding. In our system, the latter is less expensive.

2.5 Memory Management Subsystem

Memory management includes object allocation and garbage collection. Because Java code lives in the heap, the garbage collector can move it to another memory location, which requires DSS to perform operations such as invalidating the instruction cache. A more detailed description of the instructions we support for movement and modification of code appears in Section 4.

3. Related Work

In this section, we first present several simulators that we considered using to simulate the Jikes RVM. Then we discuss related work on characterizing and simulating Java.

3.1 Simulators

In the previous section, we described several features of the Jikes RVM which are often not supported by simulators, such as the signals, dynamic code generation, and support for concurrency and synchronization. We now discuss a number of simulators and their features in light of the demands of simulating the Jikes RVM and our requirement of detailed cache and memory simulation. Because the Jikes RVM only runs on PowerPC (with AIX or Linux) and x86 architectures, our discussion focuses on PowerPC simulators.

PSIM

PSIM is a program that emulates the ISA of the PowerPC microprocessor family [9]. It was developed by Andrew Cagney and his colleagues and is now an open source program bundled with the GNU debugger, `gdb`. The version of PSIM that can run Linux programs (it does not support AIX) does not have a detailed cache and memory simulation, which is essential for our experiments. However, PSIM does not support dynamic code generation, nor does it fully implement signals.

RSIM and L-RSIM

RSIM [16] has detailed cycle-level simulation of a dynamically scheduled processor and memory hierarchy with a multiplexed system bus. L-RSIM [19] is a simulation environment for I/O intensive workloads based on RSIM. The original L-RSIM added an I/O subsystem which consists of a real time clock, a PCI bus and a SCSI adaptor with one or more disk devices connected to it. The simulation of I/O requires some similar techniques to the simulation of signals. Neither RSIM nor L-RSIM support dynamic code generation as they predecode programs before simulation. Furthermore, although L-RSIM simulates I/O, it is inadequate for our requirement of simulating signals.

SimOS

SimOS [13, 17, 12] simulates hardware in sufficient detail to run a complete commercial operating system. We explored SimOS-PPC, a version of SimOS developed by IBM which simulates the PowerPC processor and runs AIX. SimOS simulates both signal handling and dynamic code generation and therefore satisfies the requirements for running the Jikes RVM. However SimOS-PPC does not have a detailed processor model and does not support all of the instructions used by the Jikes RVM. Moreover, SimOS simulates the entire machine, including the operating system, which is computationally expensive and unnecessary in the context of simulating a JVM.

3.2 Simulating Java

There have been several studies on the characteristics of the memory behavior and performance of Java programs by simulation [14, 15, 20].

Kim *et al.* [14] studied memory behavior by feeding memory access traces to cache simulators. The garbage collection algorithm they studied was mark and sweep GC. In our study, we examine the behavior of Java programs in the context of semi-space, fixed nursery generational and variable nursery generational (Appel-style) garbage collectors. The Appel-style collector is the best performing generational copying collector [6].

Li *et al.* [15] studied the performance characteristics of SPECjvm98 Java programs. They used SimOS in their experiments. They did not differentiate the impact of mutator and GC, which, as we will show later, exhibit different memory behaviors. As we stated before, SimOS does not have a cycle-level processor model, affecting the accuracy of their results.

Shuf *et al.* [20] use a very similar methodology to Kim *et al.* They generated traces and simulated memory behavior by using the trace on a cache simulator. They adopted a very large heap size, essentially ignoring the impact of GC. Also, because of their use of unusually large heaps, TLB misses are a significant feature of their results. In our study, we use different heap sizes and study the effects of GC and the interaction between mutator and GC.

4. SimpleScalar and DSS

We now introduce the SimpleScalar simulator tool set [8] and explain the extensions required to support simulating the Jikes RVM. We then group the issues and mechanisms needed into the following categories and discuss them in turn: the signal mechanism, support for thread scheduling and synchronization, and support for dynamic code generation.

4.1 SimpleScalar

We started with the version of SimpleScalar that supports the PowerPC instruction set architecture [18, 8]. The SimpleScalar tool set provides functional emulation of a target machine's ISA (in this case PowerPC), dynamic trace-driven evaluation of program impact on underlying hardware such as caches and branch predictors, and timing simulation of an out-of-order issue microprocessor core with the associated memory system.

We made only one major change to the simulator internals; the other changes, described below, mainly add functionality with the appropriate hooks. In SimpleScalar, the simulated program is predecoded before the simulation starts, to speed simulation by making instruction emulation more efficient. SimpleScalar predecodes every instruction by looking up the function that simulates the instruction's opcode, and replacing the instructions in the simulated memory with pointers to the simulation functions for those instructions.

Since dynamic compilation systems (such as the Jikes RVM) generate and modify code during execution, the predecoding as provided needed to be modified or extended. The two possibilities are re-decoding any new or modified code, or decoding each instruction on the fly. We found that it was cheaper simply to decode as the target system's instructions are fetched from its memory, as discussed further in Section 4.2.

The predecoding issue created a separate issue when handling system calls. SimpleScalar simulates only user mode instructions, implementing system calls by using the host machine as a proxy to execute the system call. When the simulated program makes a system call, the simulator obtains the arguments passed to the call and makes the call at the source level by calling the corresponding user level function call. Since the PowerPC binaries make calls to C library routines that ultimately call the operating system kernel, the PowerPC system replaces each of those library calls with a special instruction to signal a proxy call (the `sc` instruction), which is not used in user-level code by AIX-generated binaries. When the simulator encounters the `sc` instruction it emulates the system call determined by arguments in the current register values.

We needed to ensure that the `sc` instruction appeared in all code, including that code produced dynamically. Since the Jikes RVM has system calls only within a small static portion of the system written in C, we did not need to rewrite system calls in dynamically generated code. Thus we do a single rewriting pass over only the static code available when the system starts up.

The other major change to DSS internals was to add a virtual memory model that includes support for signaling a segmentation violation when a program attempts to access unmapped virtual memory. The Jikes RVM needs this functionality to support its mechanism for detecting attempts to dereference a null pointer. This extension is significant in that it affects all simulator functions that model access to the simulated memory. The segmentation violation functionality also relies on the signal support we introduced in DSS.

4.2 Major Extensions

System call	Description
mmap	manages virtual memory existence and protection
sigprocmask	changes the list of currently blocked signals
sigstack	sets and gets signal stack context
sigaction	specifies the action to take when a signal happens
kill	sends a signal to a running process
gettimerid	allocates an interval timer
incinterval	sets the value of a timer to a given offset

Table 1: List of system calls implemented in Dynamic SimpleScalar

Instruction	Description
`dcbst`	update memory from data cache
`sync`	wait for memory operations
`icbi`	invalidate code in instruction cache
`isync`	perform instruction fetch synchronization
`mfspr`	move from special purpose register
`mftb`	move from time base
`lwarx`	load-and-reserve (a.k.a. load-linked)
`stwcx`	store-conditionally
`eieio`	enforce in-order execution of I/O
`twi`	trap when a specified condition is true
`tw`	trap when a specified condition is true

Table 2: List of instructions added/changed in Dynamic SimpleScalar

We now describe the major extensions to the SimpleScalar functionality, including support for threads, dynamic compilation, and signals. We add a number of operating system features, but do not move to whole system simulation, for two reasons. One is that a user-mode simulator will almost certainly run significantly faster than a whole system simulator. The other is that whole system simulation is much more complex, both the implement, and to use, since one must accurately model many more details of the hardware, including protected mode and attached hardware devices.

Tables 1 lists the system calls we added to DSS, and Table 2 gives the hardware instructions we added.

The Signal Mechanism

The original SimpleScalar offered no support for Unix signals. Since the Jikes RVM uses a number of signals and traps (which turn into signals), thread scheduling and other common functions, we chose to implement a general signal mechanism.

Signal Generation, Delivery, and Handling

Our signal implementation includes: signal generation, receiving (or blocking) signals, handling signals, and recovering from signal handlers. The signal masks and signal handlers are set at the beginning of simulation; *sigprocmask* and *sigaction* calls can change them during the simulation.

Figure 1 offers an overview of our signal delivery system. Our implementation maintains a stack of not-yet-delivered signals. The diamond in the figure labeled "A signal arrived" tests whether the stack of pending (unmasked) signals is non-empty.

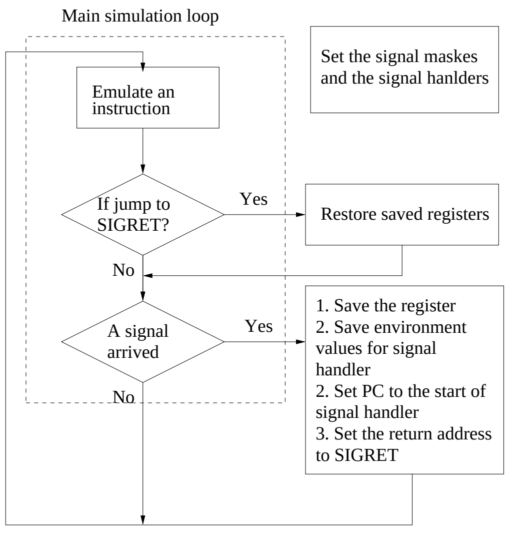
Main simulation loop
Set the signal maskes and the signal hanlders
Emulate an instruction
If jump to SIGRET?
Yes
Restore saved registers
No
A signal arrived
Yes
1. Save the register
2. Save environment values for signal handler
2. Set PC to the start of signal handler
3. Set the return address to SIGRET
No

Figure 1: Overview of signal system in simulator

We describe the implementation in more detail below.

1. Generating signals: If the simulated program issues the *kill* system call, then DSS generates the signal (pushes it on the

signal delivery stack). DSS also generates the following three signals internally:

(a) `SIGSEGV`: Whenever a memory operation tries to access an invalid memory address, DSS sends a `SIGSEGV` to the simulated program.

(b) `SIGALRM`: The Jikes RVM sets up the timer by calling *gettimerid* and *incinterval*. These functions set the value of the timer and start it. DSS updates the timer with the simulated time, which is proportional to the number of cycles elapsed since the program starts. When the timer goes off, DSS generates a `SIGALRM` and delivers it to the Jikes RVM. Depending on the parameters of *incinterval*, the Jikes RVM starts or resets the timer, or terminates the thread, to implement thread scheduling.

(c) `SIGTRAP`: The PowerPC has trap instructions, `tw` and `twi`, which generate the `SIGTRAP` signal. The Jikes RVM uses these instructions to generate exceptions such as array bounds checking, and divide-by-zero checking.

2. Delivering signals: As previously mentioned, we construct a stack that stores the most recent signal on top. At the end of emulation of each instruction, DSS checks the stack for signals that arrived during execution of the current instruction. If there is a signal, but the signal is blocked, DSS ignores the signal. (A signal could be blocked by setting the signal mask by *sigaction* and *sigprocmask*). Otherwise, DSS handles the signal.

3. Handling signals: When the user program uses the *sigaction* system call to set up a signal handler, DSS associates the signal number with the function address of the signal handler. This system call also causes DSS to block some signals while executing the signal handler. When DSS detects a signal and calls the signal handler, it performs the following operations on the simulated processor state:

   (a) Save the current contents of the registers.

   (b) Set the next PC to the start address of the signal handler.

   (c) Set the signal mask of this signal to prevent recursive triggering of the same signal, as required by *sigaction*.

   (d) Set the registers and other global variables appropriate to the signal and pass these values to the signal handler by storing them in simulated memory. For example, on an invalid memory access that triggers `SIGSEGV`, DSS passes the address for which access failed.

   (e) Set the return address of the handler function to `SIGRET`, a special constant value that will never appear as a return address in a normal program.

   Then DSS returns to the main simulation loop and continues the simulation.

4. Return from exceptions: When the signal handler finishes its execution, it will try to jump to an instruction with the PC value of `SIGRET`. DSS detects this jump, restores the saved register contents, and continues simulating the user program. Note that handlers may update the saved register state so that when the program returns from handling the signal it resumes execution somewhere else. This update is necessary in order

to support the technique that the Jikes RVM uses to turn signals into throws of Java exceptions.

Precise Interrupts

SimpleScalar's out-of-order execution did not implement precise interrupts for exceptions because it did not handle exceptions. We thus implemented precise interrupts in DSS for exceptions, to attain correct timing and program behavior in DSS. There are several methods we could have used to implement precise interrupts, such as a reorder buffer, a history buffer, or a future file [21]. As do many current microarchitectures, we use a reorder buffer to simulate the timing effects of precise interrupts.

As we described previously, DSS checks for exceptions after each instruction, and if one is found, it flushes all entries in the reorder buffer after the faulting instruction. When we simulate branch prediction for the executing the program, DSS speculatively executes instructions on the mispredicted path, but does not check or receive signals on the mispredicted path, waiting until the mispredicting instruction reaches the commit stage before servicing a signal.

Thread Scheduling and Synchronization

To support thread scheduling in Jikes, DSS needs support for `SIGALRM` as described above. In addition, DSS must support locks. Therefore, we implement the `lwarx` and `stwcx` instructions in DSS.

In our implementation, `lwarx` executes as follows:

1. Set the reservation bit to 1.

2. Set the reservation address to the effective address computed for the `lwarx` instruction (the address being loaded).

3. Load the data from memory and put it in the destination register.

An `stwcx` causes the following actions:

1. DSS checks if the reservation is 1 and the reservation address equals the effective address of the `stwcx`. If either is false, instruction execution fails and the store is not performed.

2. Otherwise, DSS stores the value to memory and resets the reservation bit to 0.

In DSS, we support only one processor, so the only instructions that will change the reservation address are `lwarx` and `stwcx`. Further, the Jikes RVM never uses an ordinary store to an address that might be reserved, so we did not need to add reservation clearing code to ordinary stores. If we need to support multiple processors or there may be some other instructions that affect reservations, we will need to change the implementation of store instructions to clear reservations as appropriate. This change is easy to implement but may significantly increase simulation time. Additional investigation may find a more efficient solution and extensions for simulating multiprocessors.

Dynamic Code Generation

Here we first discuss how DSS handles the special case of system calls, and then the more general technique that handles dynamically generated, moved, or patched code.

System Call Instruction Decoding

SimpleScalar predecodes the text segment of the program after the program is loaded into simulated memory and before simulation starts. The predecoding process patches calls to system routines and decodes the opcode of instructions to speed up simulation. DSS builds on this structure for system calls and dynamically generated code.

The Jikes RVM is designed to run as a user-level process. As such, it accesses the underlying file system, network, and processor resources via operating system calls. In the Jikes RVM, a small portion of the code is written in C and only this code can make system calls to the kernel. This code is in the text segment, and is never modified or moved by the Jikes RVM. So in DSS, we predecode system calls in the text segment before simulating the program. After simulation starts, DSS does no further checking for, or translation of, system calls, because there is no need. It would not be very difficult to add dynamic rewriting of system calls, should one desire to simulate programs needing that functionality.

SimpleScalar simulates operating system calls at the level of C library calls. The library routines generally (but not always) boil down to a corresponding `sc` instruction. However, the `sc` level interface to the operating is not published, and further, we believe that the AIX libraries rely on special support from the operating system, which we cannot replicate in a user-mode simulator. Hence, we call a corresponding host system *library* routine at the point where the simulated program would make a *library* call. In current work porting DSS to Linux we believe we can support simulation at the `sc` level, which eliminates the need for locating and rewriting library calls.

Dynamic Instruction Decoding

The original SimpleScalar predecodes all the instructions before simulation to save repeatedly decoding opcodes for the same instruction during simulation. We had to change this scheme because the Jikes RVM generates, moves, and changes machine code during the execution of a Java program. We implemented two methods for dynamic decoding:

1. The simpler scheme decodes each instruction every time it is executed, by fetching it from simulated memory. This mechanism is correct because we propagate updates to simulated memory immediately.

2. The more complex scheme predecodes instructions, managing a "cache" of pages that have been predecoded. If DSS attempts to execute an instruction from a non-predecoded page, it predecodes the page. The simulator invalidates pages in the predecoded cache.

We compared the results of both schemes and found that the first scheme is about 30% faster than the second one. Because the first scheme requires less code in the critical simulation loop, it performs better and we use it in DSS.

There are other alternatives for implementing decoding. For example, we can predecode code when it is created and detect when new code is generated or the old code is changed in the simulating program.

The cache coherence of the instruction cache is another issue in dynamic code generation. Since the Jikes RVM generates, moves, and modifies machine instructions during execution, it uses special instructions to manipulate the caches to make sure the proper instructions are fetched and executed. DSS must implement these special instructions, which were not supported by SimpleScalar, to ensure correct semantics and correct timing results. (We obtain correct semantics because there is in fact just one copy of memory data, in the simulated memory locations, but timing is possibly an issue.) The cache behavior the Jikes RVM expects on the IBM PowerPC 604 architecture is as follows:

1. After the compiler writes code (be it new code, modified code, or copied code), the program must first force the updated data cache lines to be stored into memory. The data must go through memory because there is no direct path from the data cache to the instruction cache (i-cache). The Jikes RVM uses the PowerPC instruction `dcbst` to force dirty cache lines to memory. The program must then perform a `sync`, to insure the memory writes have finished before it proceeds.

   To simulate the timing correctly, the `dcbst` instruction for dirty cache lines must take the same number of cycles as a write to memory. The `sync` instruction also stalls all subsequent instructions until `dcbst` completes.

2. The user program must then insure that those memory locations are not in the i-cache. It is possible (though perhaps unlikely) for code in location X to be (1) loaded in the i-cache, (2) copied somewhere else by the GC, and then (3) some other code written to location X later. Thus it is possible for i-cache contents to be stale. The Jikes RVM use the `icbi` instruction to invalidate updated code locations. In DSS, we implement it by invalidating cache lines, causing misses on new accesses to those instructions.

3. The Jikes RVM then does an `isync`, to insure that the `icbi` instruction completes before proceeding to the following instructions. We implemented `isync` as `sync`, which would stop subsequent instructions from executing before previous instructions finish their execution. The `isync` instruction will also flush any instruction that is already in the pipeline because it could be stale.

5. Evaluation

In this section, we describe our experimental setup, including our simulator environment, the architecture against which we validated, and our benchmark programs. We then present results for executions on both the DSS simulator and a PowerPC machine.

5.1 Experimental Setup

Jikes RVM and GCTk: We used Jikes RVM 2.0.3 in these experiments. We use the *FastTiming* configuration in which the Jikes RVM compiles and executes all methods for these experiments. This configuration exaggerates the effect of the compiler on runtime compared to the adaptive configuration, which compiles only the frequently executed methods. The adaptive configuration is not deterministic, which prevents repeatable accurate simulations and precludes its use here.

Our research group recently developed a new garbage collector (GC) toolkit for Jikes RVM, called GCTk [5, 6]. We have written a number of GC algorithms in GCTk; in these experiments, we use our GCTk implementation of Appel's generational collector [4], and run our benchmarks with various heap sizes. This collector is a very good two generational copying GC [6]. It has a flexibly sized nursery, which is initially the entire heap. Each collection reduces the nursery size by the survivors until the heap is full, which triggers a full heap collection. It performs much better than a fixed-size nursery collector in previous experiments [6]. We compare Appel-style, fixed-size nursery, and semi-space garbage collectors in the next section.

Benchmarks: We use benchmarks from the SPECjvm98 suite in this experiment. SPECjvm98 programs are designed to measure the performance of entire Java platforms, including Java virtual machines, operating systems, and underlying hardware. A detailed analysis of SPECjvm98 is given by Dieckman and Hölzle [11]. The eight benchmark programs we use are:

- _201_compress, a Java port of the 129.compress benchmark from SPEC CPU95
- _202_jess, an expert system shell
- _205_raytrace, a ray tracing program
- _209_db, which simulates a database system
- _213_javac, the Sun JDK 1.02 Java compiler, compiling jess
- _222_mpegaudio, a decoder to decompress MPEG-3 audio files
- _227_mtrt, a multithreaded raytracer
- _228_jack, which generates a parser repeatedly

PowerPC Architecture and PMAPI: We use a PowerPC machine running AIX 4.3.3 as the target machine for validation. We collected data using the PMAPI library[1] installed on this machine. PMAPI is an application programming interface for accessing the performance counters inside certain PowerPC machines.

We run each configuration (a specific heap size for a specific benchmark) three times in single-user mode, and used the run with the smallest number of cycles (i.e., the one least disturbed by other effects in the system) in our validation.

The memory hierarchy configuration of the PowerPC machine we use is as follows:

- L1 data cache: 64KB, 128 byte line, 128 way, 1 cycle latency
- L1 instruction cache: 16KB, 32 byte line, 8 way, 1 cycle latency
- L2 unified cache: 4M, 128 byte line, direct mapped, 6-7 cycle latency
- Instruction TLB: 128 entry, 2 way
- Data TLB: 256 entries, 2 way
- Memory: latency approximately 35 cycles

DSS: DSS uses the same memory hierarchy as the PowerPC machine. DSS uses the five-stage pipeline model of SimpleScalar. The details of the simulated micro-processor are as follows:

- Five-stage pipeline based on a Register Update Unit, which combines the physical register file, reorder buffer, and issue window into a single data structure

[1]http://www.alphaworks.ibm.com/tech/pmapi

- Out-of-order issue, including speculative execution
- Issue width, decode width, and commit width are 4, The size of the RUU is 16
- Bimodal branch predictor with table of 2048 entries
- Load-store queue has 8 entries

The host machine for the simulator is a SPARC running Solaris 5.8.

5.2 Validation

Although we configure DSS to have the same memory hierarchy as the PowerPC machine, the real machine has a more complicated memory system than the simulator. For example, DSS does not simulate the effects of the memory controller and memory bridge in the real machine. Likewise, DSS does not model performance effects of operating system code. Therefore, executions on DSS and the real machine produce different cycle counts (and other measures). However the performance curves of both executions on DSS and PowerPC machine should have the same trends because they are very similar.

Table 2 presents statistics for each benchmark with a heap size of 2 times the minimal heap size for that benchmark. The table contains execution results for the number of cycles, instruction numbers, L1 instruction cache misses, L1 data cache misses, TLB misses, and GCs. It also contains the comparison with the results from the native PowerPC machine.

Figure 4 compares cycle counts for DSS simulations and executions on the PowerPC machine. Because we are interested in relative trends rather than absolute cycle counts, we normalize DSS performance and real machine performance, separately, to their best performance across all heap sizes. Thus DSS performance at heap size *h* is plotted as the DSS cycle count for heap size *h* divided by the count for the best heap size. Likewise, the measured performance at *h* is plotted as the measured cycles at *h* divided by the measured performance at the best heap size. The graphs show that the trends are very similar across heap sizes and benchmarks.

Table 2 offers detailed comparison of the normalized cycle counts plotted in Figure 4. The arithmetic average of the ratios of normalized cycle counts for all benchmarks and heap sizes is 1.33%. The maximum difference across all benchmarks and heap sizes is 5.63% for _222_mpegaudio with heap size 10M. Clearly, executions on DSS and on the PowerPC machine have very similar trends in cycle counts.

Tables 2 shows that these trends are borne out for other event counts from the traces. The one measure that does not validate as well is TLB misses, which are probably strongly affected by interrupts and operating system code.

6. Example Study

This section describes two examples studies using DSS to characterize the performance of Java programs. The first compares the effect of heap size on total time. The second compares a variety of copying collectors and heap sizes and studies aggregate, mutator, and GC behavior. As in the validation section, we use the *FastTiming* configuration in which the Jikes RVM compiles and executes all methods and the same hardware configuration for these experiments.

Program	Heap	Platform	Cycle (10^6)	Inst (10^6)	I-L1 miss (10^3)	D-L1 miss (10^3)	TLB miss (10^3)	# of GCs
oamaru: _209_db	50	PowerPC	15920	9325	7224	161096	74917	12
		DSS	8989	9290	7853	157283	83759	12
		Diff	-43.54%	-0.38%	8.71%	-2.37%	11.80%	0.00%
_213_javac	50	PowerPC	14370	11853	49346	76511	19541	110
		DSS	9796	11876	60101	57068	29959	102
		Diff	-31.83%	0.19%	21.80%	-25.41%	53.31%	-7.27%
_202_jess	30	PowerPC	8199	7094	18944	52714	10200	88
		DSS	5708	7082	18785	43071	11918	88
		Diff	-30.38%	-0.17%	-0.84%	-18.29%	16.84%	0.00%
_228_jack	30	PowerPC	11862	10906	44185	49673	10092	139
		DSS	7549	10479	21765	35161	10540	150
		Diff	-36.36%	-3.92%	-50.74%	-29.22%	4.44%	7.91%
_201_compress	40	PowerPC	9248	10455	6071	164277	7560	21
		DSS	6927	10464	7056	150994	11225	21
		Diff	-25.10%	0.09%	16.22%	-8.09%	48.48%	0.00%
_205_raytrace	30	PowerPC	5913	5330	12743	47456	4696	38
		DSS	3982	5331	11517	39925	4045	38
		Diff	-32.66%	0.02%	-9.62%	-15.87%	-13.86%	0.00%
_222_mpegaudio	20	PowerPC	10326	12247	10986	36014	16269	42
		DSS	9198	12223	13415	25076	23478	46
		Diff	-10.92%	-0.20%	22.11%	-30.37%	44.31%	9.52%
_227_mtrt	50	PowerPC	5687	5086	12869	47014	4324	16
		DSS	3799	5100	11650	40204	3874	16
		Diff	-33.20%	0.28%	-9.47%	-14.49%	-10.41%	0.00%

Figure 2: DSS Simulated Results (for Heap Size = 2*Minimal Heap Size)

Program	Heap (MB)	Cycle (10^6)	Inst (10^6)	I-L1 miss (10^3)	D-L1 miss (10^3)	TLB miss (10^3)	# of GCs
db	25	9879	10207	7979	158773	89319	65
	37.5	9185	9526	7906	158151	80719	35
	50	8989	9290	7853	157283	83759	12
	62.5	8232	9145	7828	151310	52998	7
	75	8714	9126	7830	149148	80140	5

Figure 3: DSS simulated results