A Sampling Microarchitecture Simulator for Java Workloads

Rao, Pradeep
Institute of Systems & Information Technology/KYUSHU | Department of Informatics, ISEE, Kyushu University

Murakami, Kazuaki
Institute of Systems & Information Technology/KYUSHU | Department of Informatics, ISEE, Kyushu University

出版情報：Workshop on Tools, Infrastructures and Methodologies for the Evaluation of Research Systems. 2008, pp.45-52, 2008-04-20
バージョン：
権利関係：

A Sampling Microarchitecture Simulator for Java Workloads

Pradeep Rao and Kazuaki Murakami
Department of Informatics, ISEE, Kyushu University, Japan.
Institute of Systems & Information Technology/KYUSHU, Japan.
pradeep.rao@isit.or.jp

***Abstract*—Java™ has found widespread adoption across a variety of architectures. Understanding Java application behavior and further design and development of Java systems can be facilitated by software based microarchitecture simulators. However, the use of cycle-accurate, user-mode, software microarchitecture simulators in Java characterization studies are scarce and can be attributed to the following reasons: (1) simulating Java applications require the simulator to implement additional features necessary to support the Java runtime which allows dynamic compilation, thread scheduling and garbage collection, (2) the lack of such a simulator *validated* against actual hardware and its inability to support contemporary Java applications, (3) the complexity of Java applications and the intricate hardware that needs to be modelled result in impractically long simulation time for a single full run of the application, in turn adversely affecting the design and development time for Java-based systems.**

This paper seeks to address the impediments highlighted above. We enhance the dynamic simplescalar (DSS) simulator to support contemporary Java benchmark workloads. DSS is an out of order superscalar simulator for the PowerPC instruction set architecture and implements features required to support the Java runtime. In order to mitigate simulation time with minimal loss of accuracy, we implement statistical simulation sampling in the DSS simulator. We employ systematic sampling to measure in detail, only a small portion of the entire application being simulated. The application of established statistical sampling techniques allows us to evaluate performance parameters to the desired accuracy and allows us to attribute confidence levels to our estimates of performance. Finally, we validate our enhanced simulator against actual PowerPC hardware using its on-chip performance monitoring unit.

Results show that our implementation of statistical sampling in DSS is able to track actual machine performance and achieves an average speedup of over 12x when simulating Java applications. Our validated simulator should help system designers accelerate microarchitecture design space exploration of Java applications.

I. Introduction

Java™ is widely used across a variety of hardware platforms ranging from embedded systems and desktop computers to high end enterprise servers. Designing and optimizing systems for Java execution across different system architectures requires an understanding of program execution characteristics and the effect of system parameters on different aspects of performance. Software-based microarchitecture simulators[1] frequently help computer architects quickly evaluate and explore this design space and obtain quantitative estimates of expected performance. Microarchitecture simulators also help model innovative architectures and optimize processors in the design phase – a distinct advantage over characterization studies based on direct measurements on current machines. Despite these benefits, Java characterization studies and published research that explore the design space for Java applications have rarely employed cycle accurate microarchitecture simulators. We attribute the scarcity of such design data to the following impediments:

Simulating Java applications requires the typical user-mode microarchitecture simulator to implement additional features to support the Java runtime that allows for dynamic compilation, thread scheduling and garbage collection. The simulator also needs to be validated against actual hardware to document modelling errors thereby increasing the rigor of simulation studies that use the simulator. Furthermore, the complexity of contemporary Java applications coupled with the need to model complex processor microarchitectures result in extremely long simulation time for a detailed run of the benchmark application. In several cases the simulation takes days, if not weeks, for a detailed run of the complete benchmark program for a specified microarchitecture configuration. This makes effective design space exploration cumbersome and time-consuming and thus adversely impacts the design and development of Java based systems.

This paper describes our work that addresses the impediments mentioned above. We enhance the dynamic simplescalar (DSS) simulator to evaluate contemporary Java benchmarks. DSS[2] is a configurable out of order superscalar simulator for the PowerPC instruction set architecture and is based on the popular simplescalar[3] toolset. In addition to modeling microarchitecture components such as branch predictors, caches, *etc.*, the simulator also implements features required to support our chosen Java runtime – the IBM Jikes research virtual machine.

In order to accelerate simulation speeds, we employ systematic sampling to measure in detail, only a small portion of the entire application being simulated. The application of established statistical sampling techniques allow us to evaluate performance parameters to the desired accuracy and also to attribute confidence levels to our estimates of performance. We implement statistical sampling in the DSS simulator and validate our implementation against actual hardware using the performance counters available on the PowerPC processor. We hope that these measures will assist designers and architects drive design space exploration of Java applications.

The rest of the paper is organized as follows: In *Section II* we briefly describe the core features of, and our additions to, the DSS simulator to enable execution of contemporary Java applications. We then introduce statistical sampling as applied to microarchitecture simulation and detail our implementation. The framework for our experiments is discussed in *Section III* and the results of our study are contained in *Section IV*. We discuss our work in relation to earlier research in *Section V* and conclude this paper with our plans for further work in *Section VI*.

II. Implementation Details

A. Supporting the Java Runtime

We first highlight the features and requirements of our Java runtime and describe alongside, how DSS implements features that support the requirements of the Java runtime. In Section II-B we briefly introduce statistical sampling theory and describe its implementation in our microarchitecture simulator.

Our evaluations use the Jikes research virtual machine RVM [4], an open source Java virtual machine from IBM implemented mostly using the Java programming language. The RVM provides a family of compilers, infrastructure for adaptive optimization and implements several memory management policies. The Java runtime can be broadly divided into the following major components:

1) Core runtime: consisting of the thread scheduler, class loader, verifier, *etc*. This component manages the underlying data structures required for application execution and interfaces with the libraries. Several of these features rely on support for exception handling (*e.g.*, array bounds checking) and Unix *signals* (*e.g.,* `SIGSEGV`, `SIGALRM`, `SIGTRAP`). The DSS simulator implements a virtual memory model and signal handling mechanisms to handle these requirements. The flow chart for the implementation of signal handling in the DSS simulator is shown in Figure 1.

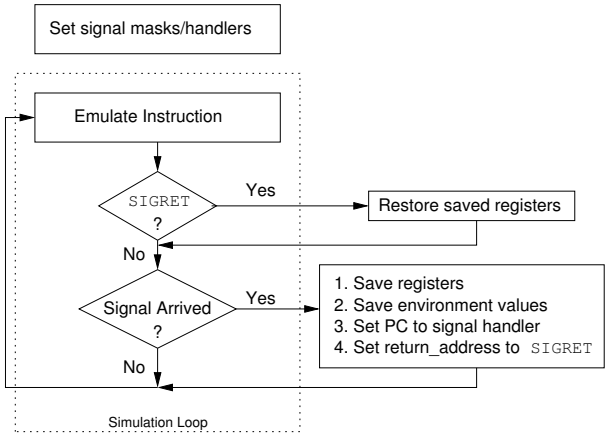

Fig. 1. Signal handling as implemented in the DSS simulator

Supporting Java threads: Java threads (both VM and application) in the JikesRVM are multiplexed onto one (or more) virtual processor(s), corresponding to each physical processor in a multiprocessor system. However, since DSS is a uniprocessor simulator, we build the JikesRVM to use only one virtual processor by setting the preprocessor directive `RVM_FOR_SINGLE_VIRTUAL_PROCESSOR`. This ensures that all threads are scheduled onto one processor. DSS also supports *locks* by implementing the `lwarx` and `stwcx` instructions. The JikesRVM uses simple time slicing to achieve load balancing and relies on timer signals to switch Java threads. DSS implements a timer mechanism and keeps it updated with the simulation time in proportion to the time elapsed since program start. Expiry of the timer generates a `SIGALRM` which is delivered to the virtual machine using the signal mechanism implemented (see Figure 1).

Instruction	Description
`dcbst`	store cache block to memory
`sync`	wait for update
`icbi`	invalidate I-cache copy
`isync`	instruction fetch synchronization
`mfspr`	move from special purpose register
`mftb`	move from time base
`lwarx`	load word & reserve indexed
`stwcx`	store conditional
`eieio`	enforce in-order IO execution
`twi/twi`	trap word

TABLE I
ADDITIONAL POWERPC INSTRUCTIONS IMPLEMENTED IN DSS

2) Compilers and Adaptive Optimization system: The compiler dynamically translates Java bytecode into executable native machine code. The Jikes RVM implements three kinds of compilers: *baseline, quick* and *optimizing*. The baseline and quick compilers seek to reduce compilation time using a simple single pass compiler. The quick compiler improves over the baseline version by employing optimizations that enhance performance without significant overheads in compilation time. When using the *optimizing* compiler, each method to be optimized goes through a series of intermediate stages with various optimizations being applied at each stage. Thus, the optimizing compiler aims to enhance performance at the expense of the time required for compilation.

The adaptive optimization system seeks to improve performance by profiling the executing application and using the optimizing compiler only when appropriate. In order to support dynamic compilation, DSS does away with the instruction predecode mechanism of simplescalar and decodes instructions as they are fetched from the simulated memory. Jikes ensures cache coherence during dynamic compilation by first storing updated D-cache contents into memory (`dcbst`), as there isn't a direct path from the code installed in the D-cache to the I-cache. These locations in the I-cache are then invalidated (`icbi`) and the pipeline flushed (`isync`) to prevent stale instructions from committing.

3) Memory Managers: are responsible for the allocation and collection of objects during runtime and is implemented in the memory management toolkit - MMTk [5]. The toolkit implements different allocation-collection plans and are detailed in [6], [5].

The additional instructions and system calls implemented

System Call	Description
`mmap`	map memory pages
`sigprocmask`	change list of currently blocked signals
`sigaction`	specify action to take on signal
`kill`	send a specified signal to a process/process group
`getitimer/setitimer`	read/write interval timer
`mkdir`	create directory

TABLE II
ADDITIONAL SYSTEM CALLS IMPLEMENTED IN DSS

in the DSS simulator are shown in Table I and Table II respectively.

B. Statistical Sampling for Microarchitecture Simulation

We first outline the theory used and then detail our implementation of sampled microarchitecture simulation.

Statistical sampling techniques attempt to estimate parameters of a *population* using identical, independent *samples* drawn from the population. The sample *mean* $\overline{Y}$ and *standard deviation* S for a random sample of *n observations* $Y_1, Y_2 ... Y_n$ from a normal population with mean μ and standard deviation σ are given by Equations 1 and 2.

$$\overline{Y} = \frac{\sum_{i=1}^{n} Y_i}{n} \tag{1}$$

$$S = \left[\frac{\sum_{i=1}^{n} (Y_i - \overline{Y})^2}{n-1} \right]^{1/2} \tag{2}$$

$$T = \frac{(\overline{Y} - \mu)}{\hat{s}} \tag{3}$$

The estimated standard deviation ($\hat{s}$) of the sample mean $\overline{Y}$ is given by $\hat{s} = S/\sqrt{n}$ and the random variable T (Equation 3) has a Student's-t distribution with $(n-1)$ *degrees of freedom* [7]. It follows that the probability that the interval given by Equation 4 contains the population mean μ is $(1-\alpha)$. The quantile value $t_{(1-\frac{\alpha}{2}, n-1)}$ of T can be determined, given α and n.

$$\overline{Y} \pm t_{(1-\frac{\alpha}{2}, n-1)} \frac{S}{\sqrt{n}} \tag{4}$$

Thus, given the desired confidence interval, we can estimate the number of samples n required to bound the error ϵ to

$$\epsilon = t_{(1-\frac{\alpha}{2}, n-1)} \frac{S}{\sqrt{n}} \tag{5}$$

Alternately, the equation above can be used to determine error in a sampling experiment at a given confidence specified using α. We use these two properties of equation 5 to quantify the error and confidence in our simulation experiments and to tune our simulation to achieve the desired error bound at predetermined confidence. We note that, for practicality, when the degrees of freedom are relatively large ($n \geq 30$) the t-distribution can be approximated by the *standard normal* [7].

Application to Microarchitecture Simulation:

Complete simulation of a Java benchmark is a time consuming process. To accelerate simulation, we simulate and measure only a small fraction of the entire dynamic instruction stream. We chose to implement systematic sampling as an approximation to random sampling due to its ease of implementation in an execution driven simulator such as the DSS.

The simulator primarily operates in three distinct modes as illustrated in Figure 2:

Measuring Mode (M): In this mode the simulator performs detailed simulation of the specified microarchitecture configuration. *M* consecutive instructions in the dynamic instruction stream constitute a sampling unit. Systematic sampling of M instructions begins at an offset of j instructions and repeats every $k \cdot M$ instructions until program termination. The cycles per instruction (CPI) is estimated (Equation 1) based on the measurements obtained from n samples of M instructions that are simulated in detail.

Warming Mode (W): Detailed simulation is initiated W instructions before starting measurement (M-mode). This is done to warmup large structures (*e.g.,* caches, branch predictors, pipeline state *etc.*) and avoid introducing significant *bias* which are otherwise introduced in the measurements [8].

Functional Mode (FF): In this mode the simulator performs fast functional simulation while maintaining only the architected state (registers and memory). None of the microarchitecture components are simulated or updated. Before starting functional simulation, the simulator completes instructions pending in the pipeline and load-store buffers and handles all queued signals.

Since detailed warming adds to the simulation time, this interval between two successive detailed simulation modes ($M(k-1) - W$ instructions) can also maintain selective microarchitecture state (caches and branch predictors) at low overheads in addition to carrying out functional simulation. This is referred to as *functional warming* and is effective in reducing the number of instructions that are otherwise required to be simulated in detail in W-mode [8].

III. EXPERIMENTAL FRAMEWORK

This section describes our experimental setup and methodology in detail.

A. Processor Simulator

We use a modified version of the DSS processor simulator for our evaluations. DSS is a detailed, cycle accurate superscalar processor simulator for the PowerPC instruction set architecture. DSS supports the Java runtime with additional features already detailed in Section II-A. Our version of the DSS simulator implements simulation sampling and models the memory system in detail. Specifically it models miss status handling registers (MSHR), store buffers and interconnect bottlenecks.

The simulator is compiled with the *gcc v3.3* compiler using *-O2* level optimizations. The simulator is run on a Sun Fire (Two Dual-core Opteron™290 processors, 8GB RAM) host

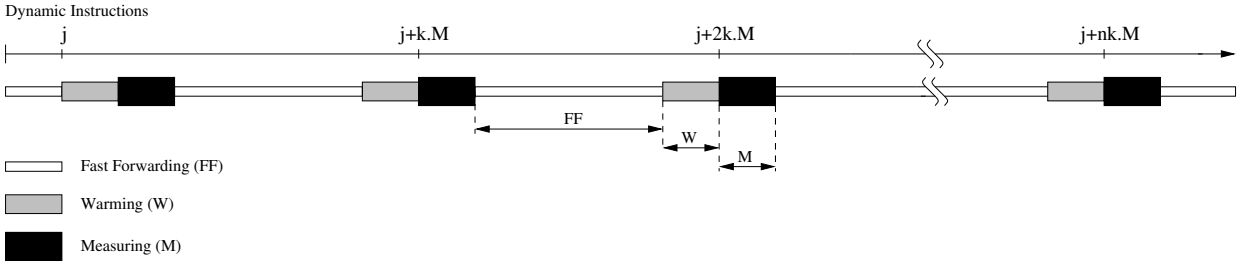

Fig. 2. Sampling Microarchitecture Simulation

running the Linux kernel 2.6.9. While measuring simulation time on this host we use *processor affinity* to bind each simulation process to a specific core and thus reduce overheads due to process migration.

The reference hardware platform (PowerMac G5) used to validate our simulator fidelity is based on the 2.0GHz PowerPC 970FX processor. The processor parameters used in our simulations are drawn from PowerPC manuals[9] and the memory and cache latencies are extracted using low level benchmarks[10], [11]. The resulting parameters are documented in Table III.

Parameter	Description
Pipeline	4 stage
Issue Width	8 way
L1 I-Cache (I$)	64KB, direct mapped, 128B line-size, VIPT, 1cycle read/write, LRU replacement
L1 D-Cache (D$)	32KB, 2-way set associative, 128B line-size, VIPT, 1cycle read/write, LRU replacement
L2 Cache	512KB, 8way set associative, LRU replacement
ITLB/DTLB	4way, 128 entries/ 4way
L1/L2/Mem. Latency	1/5/125 cycles
Functional Units	2 I-ALU, 1 I-MUL/DIV, 2 FP-ALU, 1 FP-MUL/DIV
Branch predictor	combined, 16K table, 3 cycle misprediction latency, 1 prediction/cycle, 16entry RAS, 256 entry BTB

TABLE III
MACHINE CONFIGURATION USED IN THIS STUDY

B. Virtual Machine

We use the JikesRVM *v2.4.4* runtime, an open source JVM written in Java. For accurate comparisons, the exact same compiled binary, memory images and libraries are used on both the hardware we model as well as on the simulator. All Java classes are precompiled with assertions disabled. We use the adaptive compiler and the semispace garbage collector for our studies. Some experiments employ the *replay* compiler [6] to eliminate mutator variations due to the adaptive compiler and are documented in [12].

All our experiments consider a per benchmark heap size as in [13]. We vary heap size starting from the minimum heap size and increase the heap size in steps of 0.5 times this minimum heap size upto a maximum heap size of 6 times the minimum heap size. We observe from [13] that the varying effect of garbage collection on performance can be captured within this range. The minimum heap size is determined empirically to be the least heapsize at which the Java application can execute without failure due to *out-of-memory* errors. The minimum heap size per benchmark thus obtained is shown in Table IV

C. Benchmarks

Our workload is drawn from three different benchmark suites: SPECjvm98 [14], DaCapo [15] *v2006-10-MR2* and the Java Grande [16] suites, representative of contemporary *real* Java applications.

Table IV lists the benchmark applications, a brief description, the input set used and the minimum heap size required to run the benchmark on our platform. We also list the average number of instructions and its standard deviation as measured over five runs of the application on our hardware platform. These measurements were obtained using on-chip performance monitors described next.

D. Performance Counters

We use the IBM `pmcount` [17] tool for the *Linux* operating system to access the PowerPC performance counters. The 970FX processor performance monitoring unit (PMU) has eight 32bit counters that can be configured to monitor several dynamic events such as cache misses, cache references, instructions completed, cycles elapsed, *etc*. Additional kernel modules provide 64bit virtual counters and the ability to collect per-process data.

We take the following steps to minimize measurement noise: (1) we enable only one processor during kernel boot by setting the kernel parameter `max_cpus` (PowerMac G5 is a dual processor machine). This prevents overheads due to process migration and excludes counts from the other processor, (2) all experiments are performed in single user mode by turning off all redundant applications, *daemons*, *cron* jobs running on the system. We have observed that measurements thus obtained tend to reduce the variability in measurements.

IV. VALIDATION RESULTS

Our experiments using sampled simulation demonstrate that the parameters described next, result in $\epsilon < 2\%$ (Eq. 5) for almost all benchmark applications: Each sampling unit measures in detail, 1000 instructions (M=1000) and we begin sampling at an offset of 1000 instructions (j=1000). The detailed simulation for processor (simulated) state warmup lasts

Benchmark	Description	Input	Min.heap(MB)	Instructions,in M	(σ)
compress	file compression	-s100	16	13,093	(261)
jess	expert shell system	-s100	10	120,161	(1856)
raytrace	raytracing	-s100	20	12,101	(155)
db	database	-s100	24	17,241	(281.5)
mpegaudio	MPEG-3 audio decompression	-s100	8	19,930	(715)
mtrt	multithreaded raytracer	-s100	24	17,240	(463)
jack	java parser generator	-s100	10	51,049	(3331)
hsqldb	SQL relational database	medium	176	20,451	(1588)
luindex	document indexing	medium	28	51,684	(725)
euler	computational fluid dynamics	size A	20	37,378	(133)
search	$\alpha - \beta$ pruned search	size A	16	27,018	(299)
md	molecular dynamics simulation	size A	10	9,487	(63)
mc	Monte-Carlo simulation	size A	176	21,117	(157)
rtracer	3D raytracer	size A	8	35,584	(157)

TABLE IV
BENCHMARKS CONSIDERED IN THIS PAPER: SPECJVM98, DACAPO AND JAVAGRANDE

2000 instructions (W=2000) as shown in Figure 2. We collect samples at every 1000 M-intervals (k=1000), *i.e.*, one sample of 1000 instructions is measured every 1000,000 instructions.

We first quantify the accuracy of sampled simulation against detailed simulation. Due to excessively long simulation time for detailed simulation, we perform this comparison only for the SPECjvm98 benchmark applications at two heap sizes – minimum heap size and 1.5 times the minimum heapsize. The measured CPI for detailed simulation and the estimated CPI for sampled simulation are shown in Figure 3. The thin bands over the estimated (sampled simulation) CPI bars in the figure provide the confidence interval at $\alpha = 0.997$. We observe that on an average, the CPI estimated using sampled simulation is within 6.5% (heap = min.heap) and 12.5% (heap = 1.5·min.heap) of the CPI measured using detailed simulation. These differences are significantly higher than that obtained when using sampled simulation for SPEC-CPU2000 benchmarks in [8]. However, the average simulation speedup exceeds 12x as shown in Table V. This translates to a reduction in simulation time from an order of days to a few hours at most.

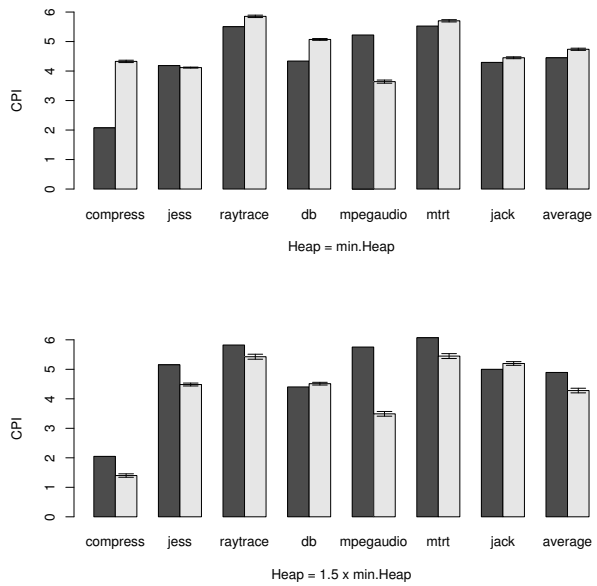

Fig. 3. Sampling simulation accuracy for SPECjvm98. Dark bars: Detailed simulation, Light bars: Sampled simulation

Validating simulator fidelity:

The microarchitecture of the PowerPC 970FX processor has several nuances (*e.g.*, deep pipelines, hardware prefetch, *etc.*) that we do not model in DSS due to its complexity and to avoid loosing the flexibility with which we can model different instances of the PowerPC architecture without binding ourselves to one implementation. Hence, we do not compare exact performance metrics between the simulator and hardware. Instead, we quantify the extent to which our simulator is able to track hardware performance, so as to enable the use of our accelerated simulator in comparative studies which will reflect equivalent hardware performance.

Figure 4 shows a scatter-plot of the machine cycles reported by the PMU against cycles estimated by our sampling simulator for each of the benchmark applications at six different heap sizes. We observe that the association is linear in most cases. A similar scatter plots for another metric – D-cache misses is shown in Figures 5.

We quantify the degree of linear association using statistical tests that measure correlation between the performance estimated by our simulator and the PMU reported performance. The correlation between two variables reflects the degree of association between them. Specifically, we use the following two statistical tests:

Pearson's product moment correlation coefficient r, is a measure of *linear relation* between two variables and is given by Equation 6. The coefficient r ranges from $+1$, indicating a perfect positive linear relationship to -1, indicating a perfect negative linear relationship between the variables. A score of zero implies that there is no linear relation between the two variables.

Sim.Time(minutes)	compress	jess	raytrace	db	mpegaudio	mtrt	jack	Average
Detailed	421	4780	613	756	1416	806	1970	1537.4
Sampling	88	361	61	69	64	70	136	121.1
speedup	4.7x	13.2x	10.1x	11x	22x	11.5x	14.5x	12.7x

TABLE V
REDUCTION IN SIMULATION TIME FOR SPECJVM98 WHEN USING SAMPLED SIMULATION

$$r = \frac{\sum XY - \frac{\sum X \sum Y}{N}}{\sqrt{\left[\sum X^2 - \frac{\left(\sum X\right)^2}{N}\right]\left[\sum Y^2 - \frac{\left(\sum Y\right)^2}{N}\right]}} \quad (6)$$

Spearman's rank correlation coefficient ρ, also measures the degree of linear association between two variables and is similar to Pearson's correlation, except that the samples are converted into ranks before computing the coefficient. Additionally, Spearman's statistic does not assume that the variables are normally distributed.

Table VI shows the values of the two correlation coefficients (ρ, r) computed for the performance metrics – total cycles, L1 D-cache misses and all L1 D-cache references. The correlation coefficients express the degree of linear association between the measured (PMU) performance metric and the estimated (simulator) performance metric at the minimum heap size for each benchmark application. We observe that there is a strong linear relationship between the simulator and PMU measurements for both total cycles as well as L1 D-cache references across most of the benchmarks.

However, the correlation on L1 D-cache misses are not easy to interpret for some of the benchmarks. We hypothesize that this is due to features on the 970FX processors that we do not model – especially the prefetcher and the perfetch and victim caches. This hypothesis is strengthened by the fact that there is a good correlation when the number of references to the L1 D-cache is considered (Table VI), but not with the misses tracked for the L1 D-cache.

We have also observed high correlation between the PMU and simulator when data TLB misses are considered as the performance metric. This indicates that our accelerated microarchitecture simulator is very effective at tracking real hardware performance and makes it extremely useful in comparative studies that evaluate microarchitecture enhancements to processor designs.

Application	Total Cycles		D$ misses		D$ references	
	r	ρ	r	ρ	r	ρ
compress	0.94	0.95	0.89	0.45	0.91	0.79
db	0.97	0.93	0.3	-0.08	0.85	0.80
euler	0.99	0.97	0.99	0.93	0.99	0.89
hsqldb	0.98	0.29	0.69	-0.082	0.99	0.51
jack	0.99	1	0.99	1	0.99	1
jess	0.99	1	0.99	1	0.99	1
luindex	0.94	0.87	0.76	0.7	0.97	0.9
mc	0.86	0.55	0.98	0.67	0.97	0.74
md	0.99	0.92	0.07	-0.21	0.99	0.9
mpegaudio	1	0.83	0.88	0.3	0.99	0.7
mtrt	0.69	0.76	0.99	0.96	0.76	0.81
raytrace	0.99	0.83	1	0.98	1	0.94
rtracer	0.99	0.9	0.99	0.96	0.98	0.96
search	1	1	1	0.95	1	0.99

TABLE VI
CORRELATION BETWEEN SIMULATOR AND PERFORMANCE COUNTER READINGS (PMU) FOR THREE METRICS

V. RELATED WORK

Simulating Java applications: Our goal in enhancing the DSS simulator was to have a platform that allows Java characterization studies at the microarchitecture level. We briefly review current microarchitecture simulators and their support for the Java runtime in general and the JikesRVM in particular. Java characterization studies that employ these simulators are indicated where appropriate. Note that JikesRVM currently supports only the PowerPC and the x86 (32bit only) processor architectures.

Li et.al. [18] characterize the SPECjvm98 [14] benchmarks using SIMOS [19], a complete system simulator that can also simulate operating system effects. However, the PowerPC variant from IBM neither supports a detailed processor model nor does it feature a timer model. SimICS [20], another full system simulator is also inadequate for our purpose due to the lack of a detailed processor model. The general execution driven simulator (GEMS [21]) builds on SimICS by adding a multiprocessor memory system simulator and an interconnect model to it. It also provides a timing model for the SPARC architecture, but does not implement the full ISA or any devices.

Radhakrishnan et.al. [22] characterize Java runtime systems at the bytecode level using the Kaffe [23] virtual machine. They characterize Java cache performance using trace based simulation tools. A similar approach is also reported in [24] and [25].

Other tools such as PSIM [26], RSIM [27] and simplescalar [3] are unable to simulate the Java runtime as they lack support to handle signals. PTLsim [28] is a high performance cycle-accurate simulator and virtual machine for the x86 and x86-64 instruction sets. However, at the time of writing this paper, signal handling support for 32bit x86 applications on PTLsim seems to be broken [29].

Simulator validation is critical to lend credibility to the studies that use it and is emphasized in the study by Desikan et.al. [30] as well as in the considerable time and effort processor manufacturers expend in ensuring that their simulators track hardware as closely as possible.

Accelerating microarchitecture simulation: The impracticality

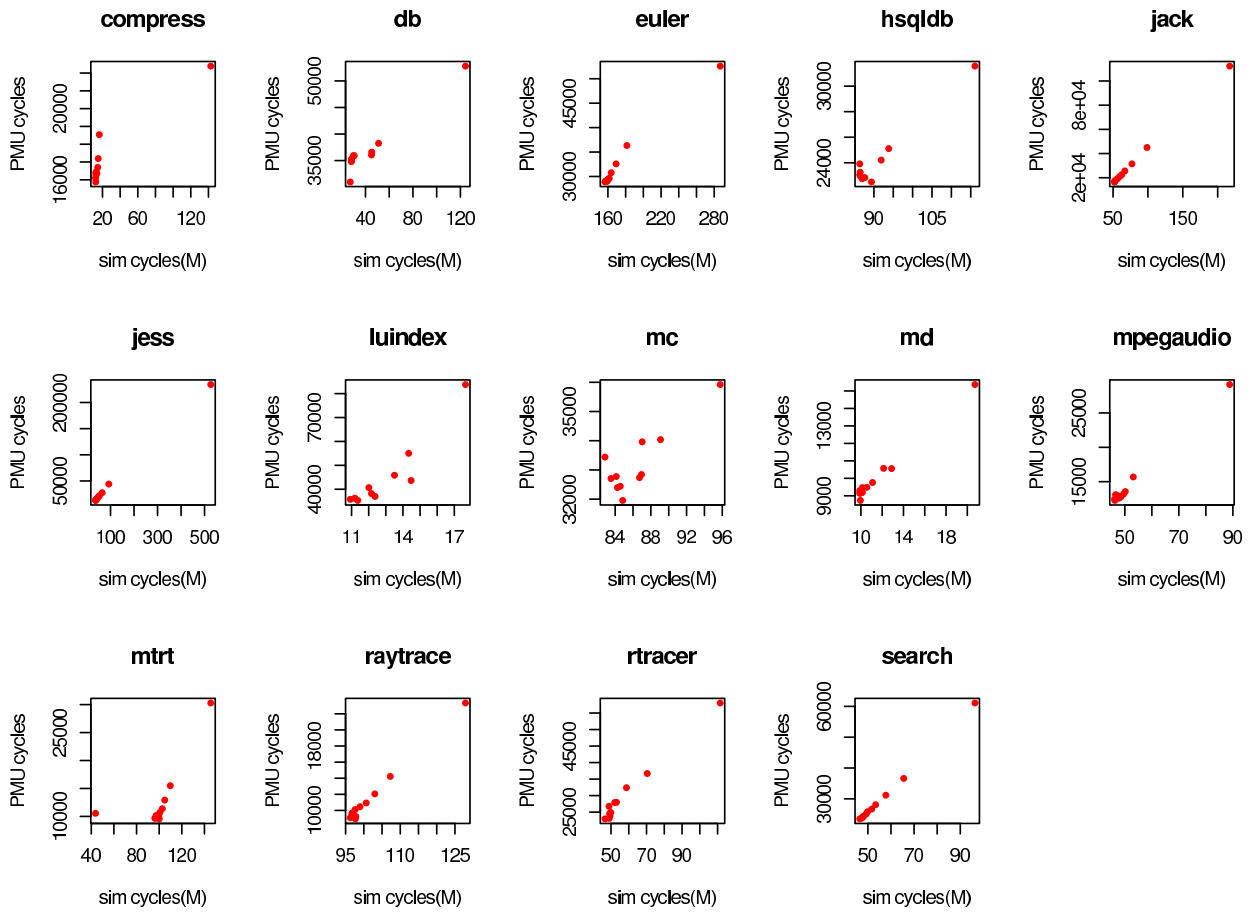

Fig. 4. Scatter-plot of PMU reported cycles versus total simulated cycles (in M-mode)

of simulating complex Java benchmarks to completion and unascertained modelling accuracy of the simulator on contemporary Java workloads has mitigated the usefulness of the DSS simulator. We implement simulation sampling [8] for Java simulation in DSS due to its effectiveness in reducing simulation time.

An earlier study [8] on using simulation sampling when simulating general purpose programs has shown that it lowers simulation time and offers higher accuracy in comparison with other techniques based on simpoint [31]. However, we observe that the average simulation error for sampled simulation of Java workloads are about 6% higher than that for general purpose SPEC-CPU2000 applications.

VI. Conclusion and Future Work

This paper makes the following key contributions: (1) We enhance the DSS simulator to support contemporary Java benchmarks and our performance evaluations are carried out with statistical rigor using the latest Java workloads drawn from three different benchmark suites. In comparison to the earlier study [2] we also employ a relatively recent version of the JikesRVM which implements several bug fixes, improves compiler optimizations and hosts an efficient memory management toolkit (MMTk) [5]. (2) We implement statistical sampling [8] in DSS to accelerate microarchitectural simulation. We describe our implementation and quantify the performance of statistical sampling over full length simulation of the SPECjvm98 benchmarks. (3) We validate and compare the fidelity of our enhanced simulator using performance measures obtained from actual PowerPC hardware using its performance counters. These measures help determine modelling adequacy and document the deviation from actual system behavior. The latest patch against DSS and a detailed technical report on the implementation and evaluation of simulation sampling in DSS can be found at [12]. We aim to keep [12] continuously updated with improved data and methodology, contributed during the course of our research.

Since sampled simulation can reduce simulation time, future efforts can be directed at improving modelling accuracy. Potential candidates for modelling effort are longer pipelines and prefetch instructions.

VII. Acknowledgments

Pradeep Rao is supported by ISIT under the JET Programme. The authors wish to thank the anonymous reviewers, Prof S.K. Nandy, Prof. Matthew Jacob (IISc) and Lovic Gauthier, Victor Goulart (ISIT) for their comments and Norifumi Yoshimatsu for his timely help with resources for simulation.

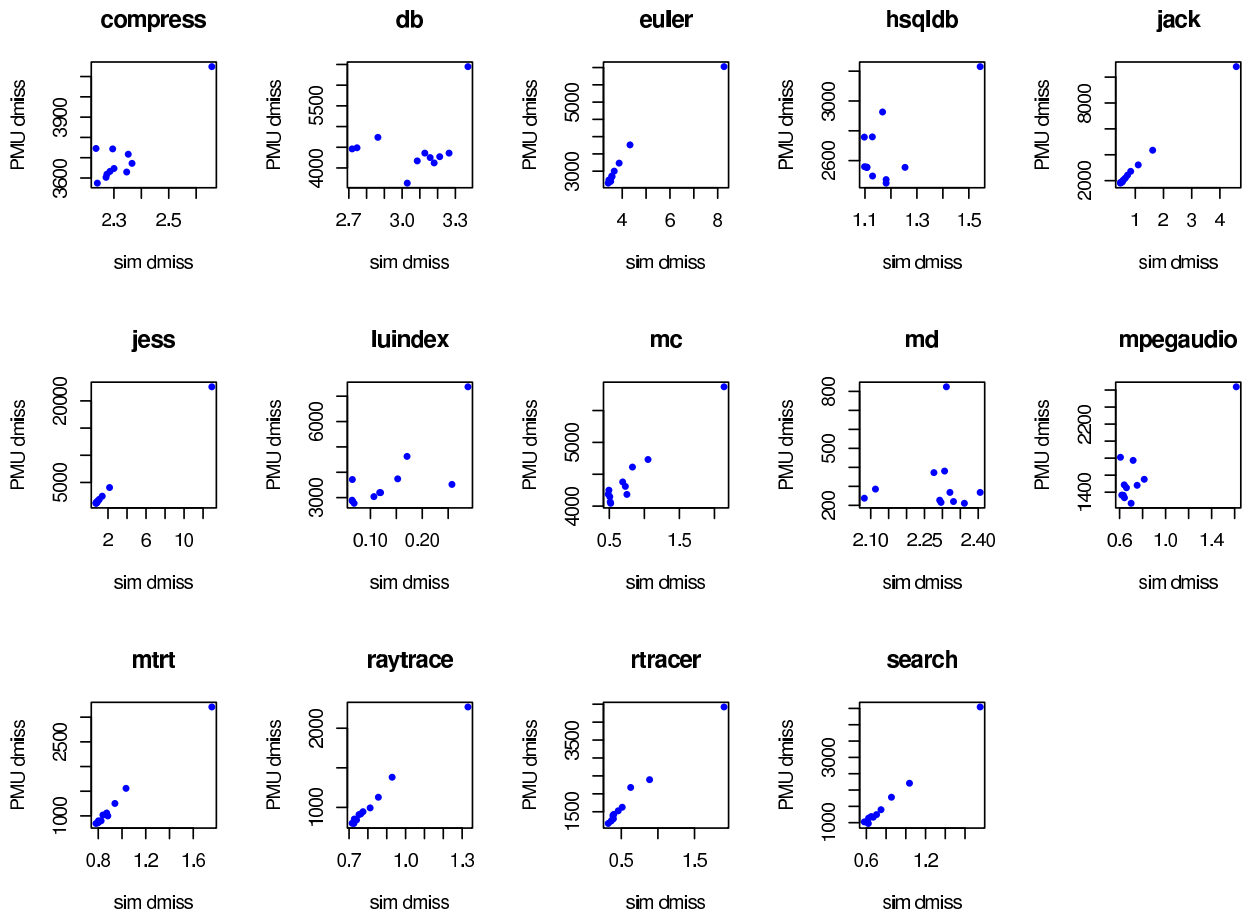

Fig. 5. Scatter-plot of PMU reported D-cache misses versus total D-cache misses reported by simulator (M-mode)

REFERENCES

[1] J. Yi, L. Eckhout, D. Lilja, B. Calder, L. John, and J. E. Smith, "The future of simulation: A field of dreams?" in *IEEE Computer*, 2006.

[2] X. Huang et.al., *Dynamic SimpleScalar: Simulating Java Virtual Machines*. TR-03-03, Department of Computer Science, UT Austin, 2003.

[3] K. Sankaralingam et.al., *SimpleScalar simulation of the PowerPC ISA*. TR-00-04, Department of Computer Science, UT Austin, 2001.

[4] B. Alpern et.al., "The Jalapeño virtual machine," in *IBM Systems Journal*, vol. 39, no. 1, Jan 2000.

[5] S. M. Blackburn et.al., "Oil and water? High performance garbage collection in Java with MMTk," in *ICSE*, 2004.

[6] *The Jikes Research Virtual Machine*. Users Guide v2.4.4, 2006.

[7] G. Canavos, *Applied Probability and Statistical Methods*. Little Brown and Company Limited, 1984.

[8] R.E. Wunderlich et.al., "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling." in *ISCA*, 2003, pp. 84–95.

[9] "IBM PowerPC 970FX RISC microprocessor: User's manual v1.6," *IBM Corporation*, 2005.

[10] L. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *USENIX Annual Technical Conference*, 1996.

[11] R. H. Saavedra and A. J. Smith, "Measuring cache and TLB performance and their effect on benchmark runtimes," *IEEE Transactions on Computers*, vol. 44, no. 10, pp. 1223–1235, 1995.

[12] Pradeep Rao, *Technical Report and DSS patch*, 2008, www.isit.or.jp.

[13] S M Blackburn et.al., "Myths and realities: The performance impact of garbage collection," in *ACM SIGMETRICS*, June 2004.

[14] Standard Performance Evaluation Council, "SPECjvm98 benchmark," http://www.spec.org/jvm98.

[15] S.M. Blackburn et.al., "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06*, Oct. 2006.

[16] Java Grande, "www.epcc.ed.ac.uk/research/activities/java-grande/".

[17] "Pmcount for linux on power architecture," *IBM alphaWorks*. [Online]. Available: www.alphaworks.ibm.com/tech/pmcount

[18] T. Li, L.K. John et.al., "Using complete system simulation to characterize SPECjvm98 benchmarks," in *ICS*, 2000.

[19] M. Rosenblum, "Complete computer system simulation: The SimOS approach," *IEEE Parallel and Distributed Technology: Systems and Applications*, vol. 3, no. 4, pp. 34–43, 1995.

[20] P. S. Magnusson et.al., "SimICS: a full system simulation platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[21] Milo Martin et.al., "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Computer Architecture News*, vol. 33, no. 4, 2005.

[22] R. Radhakrishnan, "Java runtime systems: Characterization and architectural implications," *IEEE Trans. Computers*, vol. 50, no. 2, 2001.

[23] Kaffe, http://www.kaffe.org.

[24] Y. Shuf et.al., "Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations," in *ACM SIGMETRICS*, 2001.

[25] J.-S. Kim and Y. Hsu, "Memory system behavior of Java programs: methodology and analysis," in *ACM SIGMETRICS*, 2000.

[26] A. Cagney, "PSIM - Model of the PowerPC architecture." [Online]. Available: http://sourceware.org/psim/

[27] V. Pai, P. Ranganathan, and S. Adve, "RSIM reference manual, version 1.0. ECE TR 9705, Rice University," 1997.

[28] M. T. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," in *IEEE ISPASS*, 2007, pp. 23–34.

[29] M. Yourst, "PTLsim developers mailing list," Dec 2007.

[30] R. Desikan et.al., "Measuring experimental error in microprocessor simulation," in *ISCA*, 2001.

[31] E. Perelman et.al, "Using simpoint for accurate and efficient simulation." in *ACM SIGMETRICS*, 2003.