Analyzing Memory Reference Traces of Java Programs

(Extended Abstract)

Jin-Soo Kim*

Dept. of Computer Engineering
Seoul National University
Seoul 151-742, KOREA
jinsoo@comp.snu.ac.kr

Yarsun Hsu

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598
yarsun@us.ibm.com

1 Introduction

The Java[TM1] programming language [1] is rapidly gaining in popularity and importance for the development of serious applications mainly due to its platform independence. There is no question that a significant number of future systems will be running Java applications. Nonetheless, very little is known about the execution characteristics and the architectural requirements of the Java programs.

Several studies have been performed to evaluate the performance of Java programs [2, 3]. However, their studies are limited to the interpreted Java programs and/or static executable images generated by a bytecode to native code translator. On the contrary, we will consider Java programs executed with a Just-In-Time (JIT) compiler for the following reasons. First, the interpreter turns an application's instruction reference stream into data reference stream, which makes it hard to study the application's original data reference behavior. It is also noted in [2] that application-specific behavior is overwhelmed by the performance of the interpreter itself if the application is interpreted. Second, a JIT compiler promises some significant speedup and there is no argument that JIT compilers help [4]. For the benchmark programs studied in this paper, running them with a JIT compiler was faster than with an interpreter by 1.7 to 14.3 times. Finally, using a JIT compiler is a more general way to run the Java programs, because the JIT compiler is already becoming an integral part of the Java Virtual Machine (JVM).

In spite of the wide acceptance of JIT compilers as a mechanism to run the Java applications, the characteristics of JIT-compiled Java programs have not been investigated thoroughly. We think the main reason is due to the lack of suitable instrumentation methodology for JIT-compiled programs. The existing trace-driven simulators such as ATOM [5], EEL [6] or Etch [7], rely on the static code annotations inserted to the target program at the object or binary level [8]. However, when a JIT compiler is present, it takes the Java bytecodes and compiles them into native codes *at run time*. Because the actual executable

*This work was done while the author was visiting the IBM T. J. Watson Research Center.

[1] Java is a trademark of Sun Microsystems, Inc.

Table 1: The general statistics of SPECjvm98 benchmarks

	JESS	DB	MTRT	JACK
classes loaded	219	81	102	131
methods called ($\times 10^3$)	101,884	114,282	280,340	43,795
bytecodes executed ($\times 10^3$)	1,820,852	3,700,062	2,122,522	2,996,618
Number of objects allocated	8,131,609	3,262,899	6,695,116	6,955,528
Average object size (Byte)	40	31	25	31
Heap_{min} (MB)	2	12	9	2
Heap_{max} (MB)	308	98	161	203

codes are generated dynamically, the approaches based on static instrumentation fail to handle them. In this paper, we use an exception-based approach that can trace virtually every instruction without any instrumentation to Java programs or the JIT compiler. The complete machine simulators such as SimICS [9] or SimOS [10] could be an alternative, but they are too heavy for our purposes.

The rest of the paper is organized as follows. Section 2 presents our evaluation methodology including a description of the benchmark programs and an exception-based tracing tool, JTRACE. In section 3, we analyze the various memory system behavior of Java programs using the trace-driven simulation. First, we study the cache performance of Java programs under the different heap configurations. The cache misses during the garbage collection are measured separately. Second, we investigate the behavior of objects and their lifetime characteristics. We conclude in section 4.

2 Methodology

2.1 Benchmarks

We use SPECjvm98 [11] as our benchmark programs. SPECjvm98 is the first attempt to define an industry standard benchmark suite for Java programs. Among eight applications in SPECjvm98, we only consider the following four applications here due to space limitations; JESS (_202_jess), DB (_209_db), MTRT (_227_mtrt), and JACK (_228_jack).

Table 1 shows the general statistics of benchmark programs, including the number of objects allocated, dynamic bytecode counts, and the average object size. We ran the benchmarks as stand-alone applications rather than as applets, using JDK 1.1.6 with a JIT compiler on an IBM AIX platform. In table 1, Heap_{min} denotes the minimum heap size that should be provided to run the application, while Heap_{max} means the heap size that the application begins to have no garbage collection. Note that the average object size includes 12 bytes of object header information. The size of each benchmark can be controlled by a run-time parameter, *speed*, but we always used full scale benchmarks, i.e., *speed* = 100.

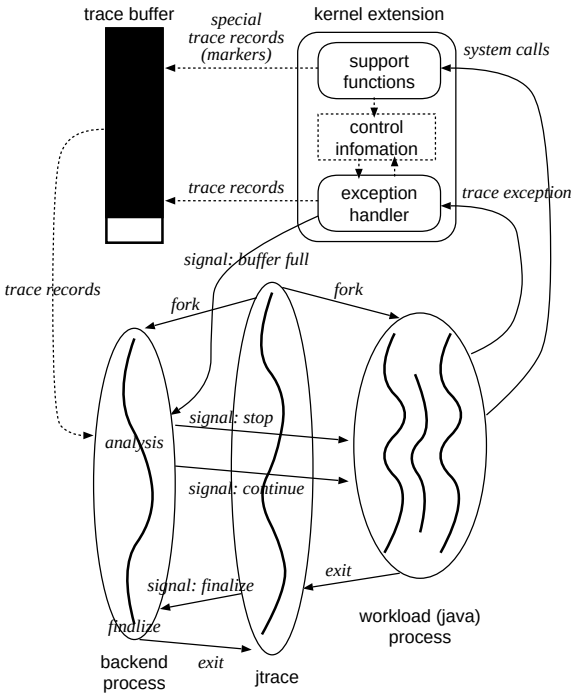

Figure 1: The organization of JTRACE

2.2 Tracing tool

To collect memory reference traces of Java programs, we have built an exception-based tracing tool called JTRACE. Some implementations of PowerPC architecture have a built-in capability to generate a *trace exception* whenever a single instruction is successfully completed, by turning on a special flag in MSR (Machine Status Register) [12]. The main advantage of the exception-based approach is that it is possible to trace virtually every instruction including system activity without any modification to executable images. No instrumentation to Java programs or a JIT compiler is required to support dynamic codes generated by the JIT compiler.

There is, however, one serious restriction when we use trace exceptions; normally operating system requires that the exception handler should not cause any page fault [13]. In other words, all the codes and data structures used by the exception handler should be *pinned*. This means that the amount of traces we can capture is essentially limited by the amount of physical memory of the machine on which traces are collected. To overcome this problem, JTRACE dynamically controls the execution of the workload process. JTRACE consists of three components as shown in figure 1; a kernel extension, JTRACE process, and a backend process.

The kernel extension contains the exception handler, system call routines and control information shared by all the components. Initially, JTRACE reserves a small portion of physical memory as a trace buffer and forks two processes, a target workload (Java) process and a backend process. And then JTRACE itself becomes idle waiting for the completion of the workload process. As soon as the workload process starts its execution, it generates trace exception for every instruction. If the current instruction is either

a load or a store, the exception handler places the corresponding data address on the trace buffer. When the exception handler detects the trace buffer is near full, it sends a signal to the backend process, which was sleeping till then. To avoid the buffer overrun, the backend process first suspends the execution of the workload using a standard UNIX signal (SIGSTOP). It is a responsibility of the backend to consume the trace records in the buffer, either by running a trace-driven simulator on-the-fly or by storing them into a file. Eventually, the state of the trace buffer is reset to be empty and the execution of the workload is resumed for another bunch of tracing records. This repeats until the workload finishes its execution, in which case JTRACE wakes up and asks the backend to finalize the current tracing.

The hardware platform used for tracing is an IBM RS/6000 7043-140 running IBM AIX 4.3.2, with 332MHz PowerPC 604e microprocessor and 768MB of main memory. The slowdown factor of tracing is found to be about 100 to 200, with the stack simulation described in the next section. The size of the trace buffer was generally independent of the tracing speed and a small trace buffer (64MB) worked quite well.

2.3 Evaluation methodology

Several different backends are used to study various aspects of Java programs. For cache performance, we use a stack simulation algorithm [14] that can generate miss ratios for different sizes of fully-associative caches in one pass. Our implementation of stack simulation algorithm is based on [15], where the algorithm is accelerated by considering cache sizes that are powers of 2. Additionally, we instrument the JVM so that it makes system calls before and after the garbage collection, which notifies the exception handler whether the Java program is in the garbage collection routine or not. This enables us to count the exact number of instructions, data references and cache misses that took place during the garbage collection.

The JVM has two run-time flags, "`-ms`" and "`-mx`", which specify the initial and the maximum sizes of heap memory. On JDK 1.1.6 for AIX, default values for these flags are 1MB and 32MB, respectively. We consider six different heap configurations in this paper; default (`-ms1M -mx32M`), 16MB (`-ms16M -mx16M`), 32MB (`-32M -mx32M`), 48MB (`-ms48M -mx48M`), 64MB (`-ms64M -mx64M`), and infinite (`-ms320M -mx320M`). With infinite heap configuration, no benchmark has any garbage collection.

We are also interested in the memory system behavior of Java programs at the object level. Memory reference traces alone, however, do not carry any information on the objects. To collect object-level statistics, we modify the JVM to produce at the end of the execution a file which contains the start address and the size of each object by traversing the entire heap memory. During tracing, the backend simply keeps track of the number of references and the first and the last reference for each 8-byte heap memory block[2]. When the backend receives a finalization signal from JTRACE, it can identify which memory blocks belong to the same object by reading through the file generated by the JVM. Then, the backend merges the statistics for those memory blocks and converts it into that of the object. In this measurement, we provide enough heap space (320MB) to prevent objects from being moved or reclaimed by the garbage collector.

[2]This is an allocation unit used for heap memory in JDK 1.1.6.

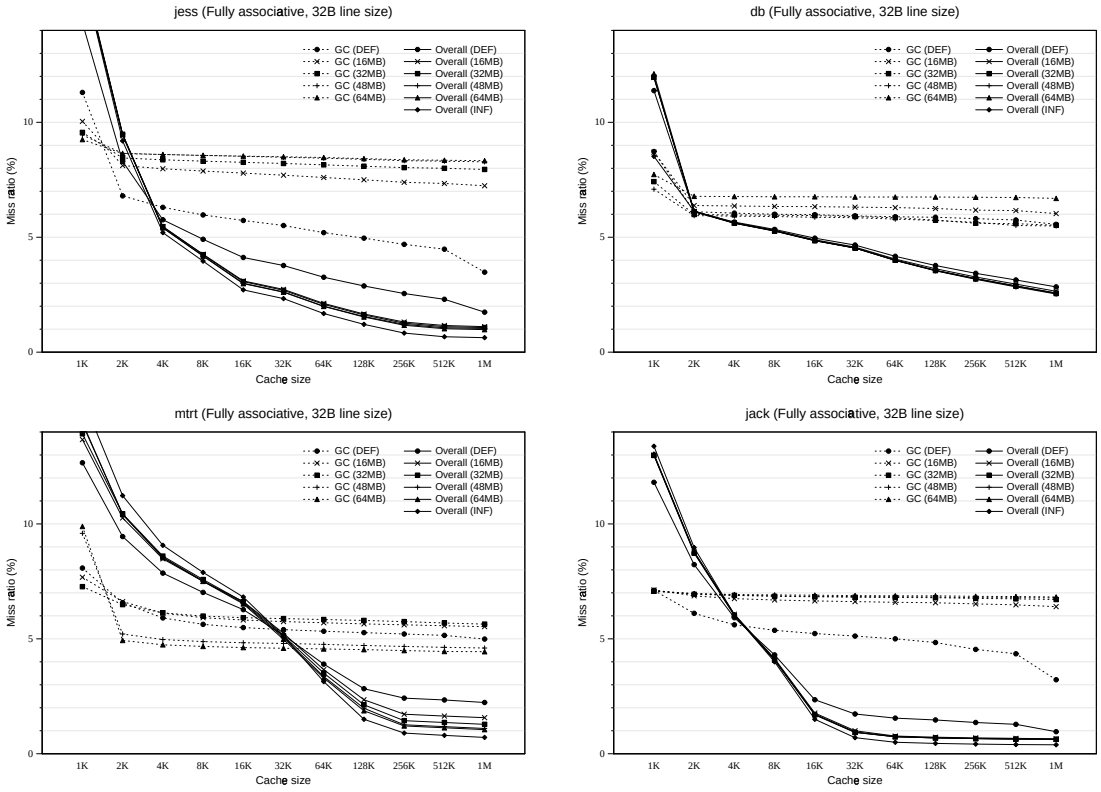

Figure 2: Data cache miss ratios with different heap configurations

3 Results

3.1 Cache performance

Figure 2 displays the fully-associative data cache miss ratios of SPECjvm98 applications under different heap configurations, where miss ratios during garbage collection are separately plotted in dotted lines. First, we note that the miss ratios under infinite heap configuration are not worse than those of integer SPEC92 applications written in C or Fortran. In [16], the average data miss ratio for integer SPEC92 benchmarks is reported to be 0.91% at a 256KB, 8-way set-associative cache with the same 32-byte line size. In our Java applications, only DB shows the higher miss ratio at this cache size, which does not have a clear working set after 2KB.

From figure 2, we can see that the miss ratios during garbage collection are generally higher than the overall miss ratios and appear to be insensitive to the cache sizes. This is likely because the garbage collector should sweep through the entire heap memory and its working set size is connected to the total heap size. As observed in [17], the garbage collector imposes both direct and indirect costs on the overall performance; directly, the garbage collector itself executes its own instructions and causes cache misses. Indirectly, it interferes with the application's temporal locality, making the application suffer from cache misses right after the garbage collection. Also, the garbage collector can move objects, which may improve (or degrade) the application's locality. Indirect costs of garbage collection are, however, found

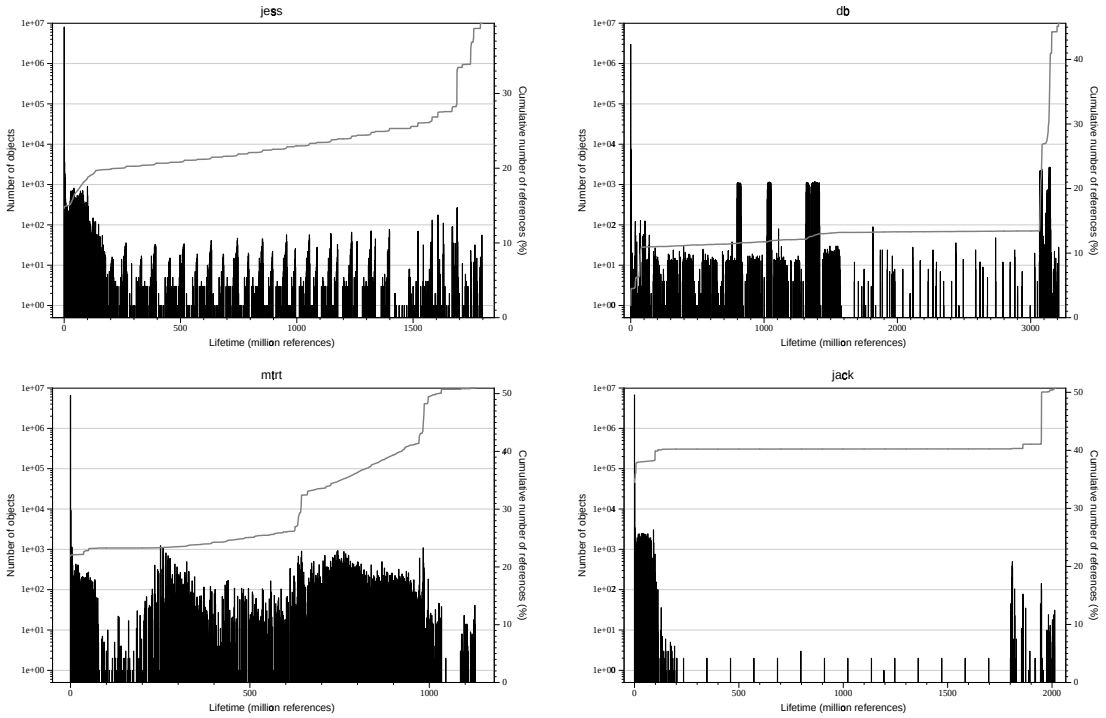

Figure 3: The distribution of lifetimes and the number of references

to be negligible; if we exclude data references occurred in garbage collection, all the applications show the same curves as with infinite heap, regardless of the heap size used. This means that the differences in overall cache miss ratios mostly result from the cache misses occurred in garbage collection. Since the variations in the miss ratios during garbage collection are small across different heap sizes, usually it is the percentage of the garbage collector's data references that determines how much the overall miss ratios are affected.

The results presented in figure 2 are the miss ratios for fully-associative caches obtained via the stack simulation. We have also simulated conventional set-associative caches and found 2-way set-associative caches closely resemble the performance of the fully-associative caches when the cache size is moderate.

3.2 The behavior of objects

The lifetime characteristics. We define an object is *live* with respect to the memory reference, if it is being actively referenced by the application. We call an object *long-lived* (*short-lived*), if it is live for a relatively long (short) time. On the other hand, we define an object is *reachable*, if it is accessible by following references from the *root set*. We will call an object *long-resident*, if it is reachable for a relatively long time, to distinguish it from long-lived objects.

We timestamp the first and the last reference to an object with the number of total data references made up to that point. Assume that the first reference to an object is made at the t_1-th data reference and the last one at the t_2-th data reference. According to our definition, the object is live from t_1 to t_2

Table 2: The dynamic statistics of SPECjvm98 benchmarks (without garbage collection)

	JESS	DB	MTRT	JACK
Instructions executed ($\times 10^3$)	5,327,589	9,167,372	3,916,519	6,552,677
Data references ($\times 10^3$)	1,797,615	3,211,373	1,129,016	2,014,066
% heap references	39.40%	45.61%	50.97%	50.74%
Heap memory allocated (KB)	314,533	99,927	164,444	207,550
Average allocation rate (bytes/1000 instructions)	60	11	43	32
Average lifetime (million refs.)	1.3	129.5	10.8	1.6
Average number of references				
per each object	87	449	86	147
per each word	9	57	14	19
Lifetime $<$ 1 million refs.				
% objects	99.19%	92.33%	98.03%	97.33%
% heap memory	98.88%	71.15%	97.60%	95.48%
% heap references	37.15%	4.14%	42.91%	67.97%
% total data references	14.64%	1.89%	21.87%	34.48%

and the lifetime of the object is the duration that it remains live, i.e., $t_2 - t_1$. If an object is not referenced anymore, it is not visible to the memory system and we consider the object is dead even though it is still reachable.

Figure 3 illustrates the distribution of the objects which have the same lifetime. The lifetime is calculated in the unit of 1 million references and is arranged in ascending order; short-lived objects are on the left, while long-lived objects are on the right. Note that the number of objects, on the left y-axis, is displayed in log scale. We can observe that the large percentage of objects has a lifetime less than 1 million references and is located at the leftmost of the graph ($x = 0$).

The dotted curve in figure 3, associated with the right y-axis, shows the cumulative distribution of data references. A point on the curve indicates, in its y value, the fraction of references to the objects whose lifetimes are less than or equal to x. We note that although most objects are short-lived, a substantial amount of references is still going to the long-lived objects in some applications. This is plausible because long-lived objects tend to hold essential information such as database records (DB) or scene data (MTRT), and have more chances to get referenced. DB is one of the extreme cases; 92% of objects have the lifetime less than 1 million references, but only account for 4% of total heap references.

Table 2 summarizes the lifetime characteristics of SPECjvm98 applications including the number of (native) instructions and data references. Also, the table shows the percentage of short-lived objects in terms of the number of objects, heap spaces, and data references, whose lifetimes are less than 1 million references.

The implication to the cache performance. The results in figure 3 are obtained under infinite heap configuration, therefore, show the inherent behavior of objects which is not disturbed by the garbage collection. We now explore the implication of this behavior on the cache performance. The cache performance under the infinite heap configuration is also important because it is functioning as an upper bound of what can be achieved with smaller heap sizes, due to the presence of garbage collection (cf.

figure 2).

First, we consider JESS and JACK. As we can see in table 2, most of objects in JESS and JACK die young. We can expect that such short-lived objects have good temporal locality, and do not cause capacity misses. However, they touch more than 95% of the total heap memory and the average number of references per cache line is relatively small. Consequently, the short-lived objects produce a substantial amount of cold misses. It is possible to reduce the number of cold misses by decreasing the available heap size, and reusing the same line for different objects. But in this case, the cache misses during the garbage collection quickly offset the benefit of reduced cold misses, as we can see in figure 2.

JESS and JACK also have some frequently-referenced long-lived objects; although these objects can cause capacity misses by interfering with short-lived objects, their total size is very small. Therefore, if the cache size is big enough to accommodate them, the capacity misses will almost disappear. For instance, 244 objects (33KB in size) account for about 9% of all data references in JACK, with the lifetime of around 1950 million references (cf. figure 3). Once the cache size is larger than 32KB, the miss ratio of JACK does not change, as shown in figure 2. Most of cache misses in these cache sizes result from the cold misses of short-lived objects. Similarly, in JESS, long-lived objects, whose lifetimes are more than 1690 million references, are responsible for 12% of the total references. They can fit into a 16KB cache and constitute the first working set. Another 8% of data references are distributed over the objects with the lifetimes ranging from 140 million to 1690 million references. With the caches larger than 256KB, these objects can stay in the cache and they determine the second working set of JESS, as can be seen in figure 2. In any case, the size of frequently-referenced long-lived objects is closely related to the application's working set size. Long-lived objects which are not referenced frequently do not contribute to the overall cache miss ratio, and can be ignored.

The same observation holds for DB and MTRT. However, these applications have more long-lived objects and keep a larger amount of live objects (3MB – 8MB) compared to JESS and JACK. Thus, even with a 1MB of cache, it is not possible to accommodate all the frequently-referenced long-lived objects and some capacity misses are inevitable. For MTRT, the most-frequently-referenced long-lived objects appear to fit into 256KB, where it exhibits the first working set size. Still, a significant number of references is made to other long-lived objects, thereby resulting in the capacity misses. In DB, 33% of the total references are going to the objects that exist throughout the lifetime of the application. Their total size is larger than 1MB and this is the reason the miss ratio of DB is continuously decreasing in figure 2. Among the cache misses caused by heap references at a 512KB cache, 35% of misses are due to capacity misses for MTRT, while the percentage for DB is 96%.

4 Concluding Remarks

This paper studies the memory system behavior of JIT-compiled Java programs based on the memory reference traces generated by an exception-based tracing tool. Specifically, we examine the cache performance, the behavior of objects and their relationships.

We find that the cache performance of the Java program itself is not significantly worse than the programs written in C or Fortran languages. However, the garbage collector suffers from higher cache

miss ratio and inflates the overall cache miss ratios of Java programs. We also observe that Java programs generate a substantial amount of short-lived objects, but in general, the size of frequently-referenced long-lived objects is closely related to the application's working set size and is critical to the cache performance.

References

[1] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Addison-Wesley, 1996.

[2] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Bear, B. N. Bershad, and H. M. Levy, "The Structure and Performance of Interpreters," in *ASPLOS VII*, pp. 150–159, 1996.

[3] C.-H. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhall, and W. W. Hwu, "A Study of the Cache and Branch Performance Issues with Running Java on Current Hardware Platforms," in *Proc. of IEEE Compcon*, pp. 211–216, 1997.

[4] P. Tyma, "Why are we using Java again?," *Communications ACM*, vol. 41, pp. 38–42, Jun. 1998.

[5] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," in *Proc. of the SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 196–205, 1994.

[6] J. R. Larus and E. Schnarr, "EEL: Machine-independent Executable Editing," in *Proc. of the SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 291–300, 1995.

[7] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen, "Instrumentation and Optimization of Win32/Intel Executables Using Etch," in *The USENIX Windows NT Workshop Proceedings*, 1997.

[8] R. Uhlig and T. Mudge, "Trace-driven Memory Simulation: A Survey," *ACM Computing Surveys*, vol. 29, pp. 128–170, Jun. 1997.

[9] P. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner, "SimICS/sun4m: A Virtual Workstation," in *USENIX Annual Technical Conference*, 1998.

[10] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, "Using the SimOS Machine Simulator to Study Complex Computer Systems," *ACM Trans. on Modeling and Computer Simulation*, vol. 7, pp. 78–103, Jan. 1997.

[11] Standard Performance Evaluation Council, "SPEC JVM98 Benchmarks." `http://www.spec.org/osg/jvm98/`, 1998.

[12] IBM Corp., *PowerPC 604e RISC Microprocessor User's Manual*. 1998.

[13] IBM Corp., *AIX Version 4.3 Kernel Extensions and Device Support Programming Concepts*. 1998.

[14] J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[15] Y. H. Kim, M. D. Hill, and D. A. Wood, "Implementing Stack Simulation for Highly-Associative Memories," in *Proc. of 1991 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pp. 212–213, 1991.

[16] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith, "Cache Performance of the SPEC92 Benchmark Suite," *IEEE Micro*, pp. 17–27, Aug. 1993.

[17] M. B. Reinhold, "Cache Performance of Garbage-Collected Programs," in *Proc. of the ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 206–217, 1994.