Memory System Behavior of Java Programs: Methodology and Analysis

Jin-Soo Kim*
Electronics and Telecommunications Research Institute (ETRI)
Taejon 305-350, Korea
jinsoo@computer.org

Yarsun Hsu†
Dept. of Electrical Engineering
National Tsing Hua University
Hsinchu, Taiwan
yshsu@ee.nthu.edu.tw

ABSTRACT

This paper studies the memory system behavior of Java programs by analyzing memory reference traces of several SPECjvm98 applications running with a Just-In-Time (JIT) compiler. Trace information is collected by an exception-based tracing tool called JTRACE, without any instrumentation to the Java programs or the JIT compiler.

First, we find that the overall cache miss ratio is increased due to garbage collection, which suffers from higher cache misses compared to the application. We also note that going beyond 2-way cache associativity improves the cache miss ratio marginally. Second, we observe that Java programs generate a substantial amount of short-lived objects. However, the size of frequently-referenced long-lived objects is more important to the cache performance, because it tends to determine the application's working set size. Finally, we note that the default heap configuration which starts from a small initial heap size is very inefficient since it invokes a garbage collector frequently. Although the direct costs of garbage collection decrease as we increase the available heap size, there exists an optimal heap size which minimizes the total execution time due to the interaction with the virtual memory performance.

1. INTRODUCTION

Although the Java programming language [7] is rapidly gaining in popularity and importance for the development of serious applications, very little is known about the execution characteristics and the architectural requirements of the Java programs. Most of the architectural evaluations have been performed using scientific or commercial workloads [14; 1], which are written in C, C++, or Fortran languages. Unlike those languages, Java has several distinctive features such as automatic memory management, a support for multithreading, the existence of architecture-neutral intermediate codes (*bytecodes*), etc. Thus, it is very interesting to see whether Java applications share similar characteristics with the traditional applications or not.

With the increasing gap between the speeds of CPU and memory, memory system has become a major performance bottleneck in modern computer systems. Java has a potential to stress the underlying memory system further because of its automatic memory management and the accompanying garbage collection (GC). Typically, Java applications allocate a substantial amount of objects in a heap memory. Since the size of heap memory is limited, it is quickly exhausted and a garbage collector is invoked to reclaim memory allocated to objects that will not be used again. Depending on the characteristics of the application and the available heap size, Java applications can spend considerable time on garbage collection. The application's cache and virtual memory performance will be also affected by the heap size. This paper examines the memory system behavior of Java programs focusing on the various factors that affect the memory system performance under the different heap memory configurations.

Java programs can be executed either by a virtual machine interpreter or by a Just-In-Time (JIT) compiler. Alternatively, they can be translated into native codes by traditional ahead-of-time compilers. In this paper, however, we only consider Java programs executed with a 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 [17] 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 [21]. 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 has not been investigated thoroughly. We think the main reason is due to the lack of suitable instrumentation methodology for JIT-compiled

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

†On leave from IBM T. J. Watson Research Center.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advant -age and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMETRICS 2000 6/00 Santa Clara, California, USA
© 2000 ACM 1-58113-194-1/00/0006...$5.00

Table 1: The general characteristics of SPECjvm98 benchmarks.

	COMPRESS	JESS	DB	JAVAC	MTRT	JACK
classes loaded (user/total)	27 / 87	154 / 219	18 / 81	156 / 220	40 / 102	67 / 131
methods called ($\times 10^3$)	225,961	101,884	114,282	89,887	280,340	43,795
bytecodes executed ($\times 10^3$)	12,474,021	1,820,852	3,700,062	1,953,961	2,122,522	2,996,618
Number of objects allocated	7,446	8,131,609	3,262,899	6,244,896	6,695,116	6,955,528
Average object size (Byte)	14,823	40	31	36	25	31
Heap_{min} (MB)	15	2	12	12	9	2
Heap_{max} (MB)	106	308	98	217	161	203

programs. The existing trace-driven simulators, such as ATOM [18] or EEL [12], rely on static code annotations inserted to the target program at the object or the binary level. 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 codes are generated dynamically, the approaches based on static instrumentation fail to handle them. Instead, we use an exception-based approach that can trace virtually every instruction without any instrumentation to Java programs or the JIT compiler.

The rest of the paper is organized as follows. Section 2 summarizes the related work and Section 3 presents our evaluation methodology including descriptions of the benchmark programs and the exception-based tracing tool called JTRACE. In section 4, we analyze the cache performance, the behavior of objects, and the performance impact of garbage collection and heap size. We conclude in section 5.

2. RELATED WORK

Romer et al. [17] and Hsieh et al. [8] evaluate the performance of Java programs, but their studies are limited to the interpreted Java programs and/or static executable images generated by a bytecode to native code translator. Radhakrishnan et al. [15] compare the characteristics of both the JIT and the interpreter and their interaction with the architectural features such as the cache and branch prediction hardware. None of the previous studies, however, examined the behavior of Java objects and its implication on the cache performance, nor did they analyze the performance impact of garbage collection and heap size.

The impact of garbage collection has been studied mainly for functional languages such as Lisp [22], Scheme [16] or Standard ML [6; 3]. Some observations from functional programs can be applicable to Java programs. However, all the run-time systems of functional languages employ a sort of *generational* garbage collectors, while the Sun's standard implementation of JDK 1.1 uses a simple *mark-and-sweep* garbage collector. In addition, there is no notion of *native methods* in functional languages. Due to these differences, it is hard to draw any conclusion for Java programs from the results of functional programs.

3. METHODOLOGY

3.1 Benchmarks

We use SPECjvm98 [19] as our benchmark programs. The SPECjvm98 benchmark suite is the result of an effort to define an industry standard benchmark for Java programs. Most applications are derived from real applications and designed to evaluate the performance of both hardware and software aspects of the JVM. Among eight applications in SPECjvm98, we exclude **_200_check**, which is a synthetic benchmark to check various features of the JVM and the Java language. **_222_mpegaudio** is found to use a very small amount of heap memory (less than 1MB), and thus it is not studied either. We now briefly describe the structures and input data of benchmark programs used in this paper[1].

_201_compress COMPRESS is a Java port of the **129.compress** benchmark from SPEC CPU95. The benchmark compresses and decompresses each input file in memory using two buffers whose sizes are equal to the input file size. It makes five loops over five input files, hence handles 25 files.

_202_jess JESS is the Java Expert Shell System based on NASA's CLIPS system. The benchmark solves a word puzzle and 14 number puzzles. For the number puzzles, each time it asserts a new set of facts representing the same puzzle but with different literals. Thus, the inference engine must search through progressively larger rule sets as execution proceeds.

_209_db DB emulates multiple database operations on memory resident database. The database is normally accessed via an `index` structure, where the references to database records are sorted based on a certain field. Each database record consists of 8 fields.

_213_javac JAVAC is a Java compiler from the JDK 1.0.2. The same Java source file is compiled four times.

_227_mtrt MTRT is a multithreaded raytracer that works on a scene depicting a dinosaur. The main thread forks two threads and each thread is responsible for generating rendering results to the left or the right half of the output canvas.

_228_jack JACK is a Java parser generator that is based on the Purdue Compiler Construction Tool Set (PCCTS). The input file contains instructions for the generation of JACK itself, and is fed to JACK 17 times.

In the experiments, we ran the above benchmarks as stand-alone applications rather than as applets, using IBM JDK 1.1.6 with a JIT compiler on an AIX platform. To minimize nondeterministic behavior of Java programs across multiple runs, we turn off the asynchronous garbage collector. Also, all the synchronous garbage collection calls in the benchmarks are disabled so that the garbage collector is invoked only if there is not enough space in the heap memory.

[1] Three different data set sizes are available for each benchmark, but we always used full scale benchmarks by specifying `-s100` at the command line.

Table 1 shows the general characteristics of benchmark programs including the number of objects allocated, dynamic bytecode counts, and the average object size. 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. In contrast to the Sun's standard implementation, IBM JDK uses an enhanced object layout to improve the performance of the JVM. In the new object layout, handles are removed, and each object has two-word header instead of a handle. Therefore, the average object size in table 1 includes 12 bytes of the space overhead; two words for a header and one word for heap maintenance [20]. We can see that most objects are very small except for COMPRESS, where a small number of large buffers increase the average object size.

3.2 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) [10]. 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. Moreover, no instrumentation to Java programs or a JIT compiler is required to trace dynamic codes generated by the JIT compiler.

The type of instruction and the effective address of an operand for a load or a store instruction are available to the exception handler. The exception handler can save this information into a trace buffer for future analysis. 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 [9]. 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; (1) a kernel extension, (2) JTRACE process, and (3) 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.

Only the tracing flag of the workload process is enabled when it is created by JTRACE. As soon as the workload process starts its execution, it generates a trace exception for each instruction. If the current instruction is either a load or a store, the exception handler places the corresponding data address on the trace buffer. JTRACE also provides the workload process with several system calls. These system calls can be used to turn on or off the tracing selectively, or to insert special tracing records into the trace buffer, in case the source code of the workload is available.

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 set 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. By reusing the small trace buffer in this way, we are able to trace long-running applications regardless of the physical memory size.

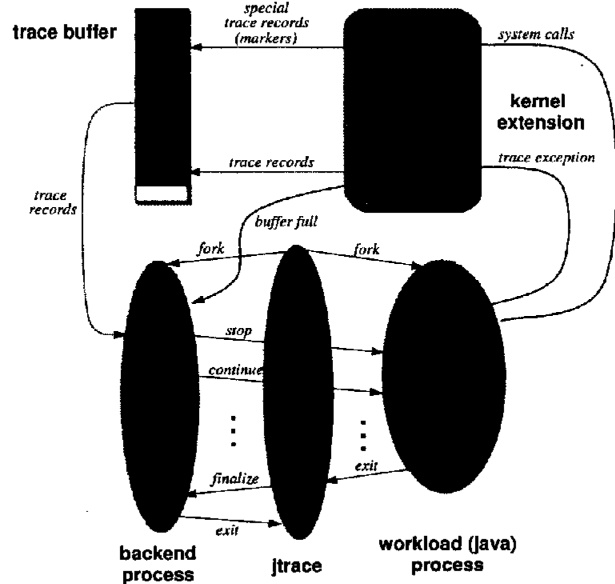


Figure 1: The organization of JTRACE.

Because the tracing significantly slows down the execution of the workload, a special care has been taken so that the workload receives roughly the same number of clock interrupts during its scheduling quantum. This is done by compensating timing related registers according to the actual number of instructions traced. Otherwise, the tracing results will exaggerate the kernel activity due to excessive context switches.

The hardware platform used for tracing is an RS/6000 model 7043-140 running IBM AIX 4.3.2, with 332MHz PowerPC 604e microprocessor and 768MB of main memory. The slowdown factor of tracing is about 100 to 200, with the backend simulating a stack algorithm (described in the next section). The size of the trace buffer was generally independent of the tracing speed and a small trace buffer (64MB – 128MB) worked quite well.

3.3 Evaluation methodology

Several different backends are used to study various aspects of Java programs. For cache performance, we have constructed a backend simulating the stack algorithm [5] that can generate miss ratios for different sizes of fully-associative caches in one pass. Our implementation of the stack simulation algorithm is based on [11], where the algorithm is

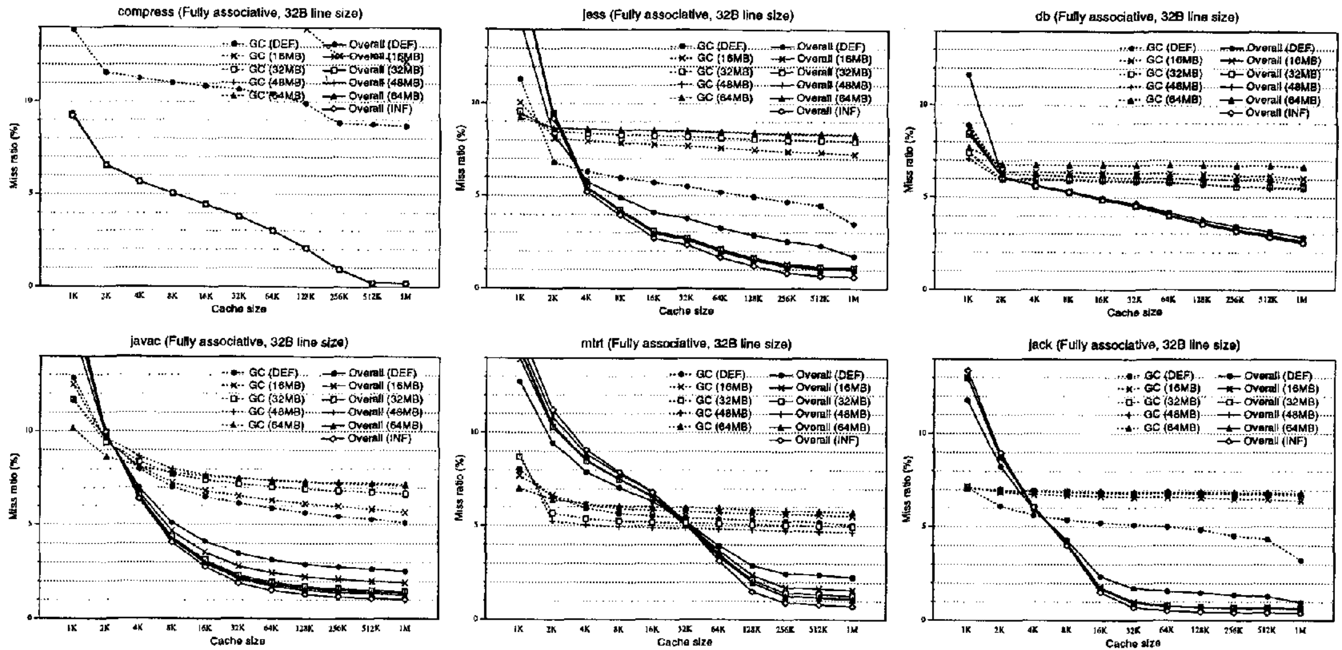


Figure 2: Data cache miss ratios with different heap configurations. DEF and INF correspond to the default and the infinite heap configuration, respectively.

accelerated by considering cache sizes that are powers of 2 only. For comparison, direct-mapped and set-associative caches are simulated using another backend built from a conventional cache simulator, dinero IV [4]. We use a fixed line size of 32 bytes for the cache simulation.

Additionally, we instrument the JVM so that it makes system calls before and after garbage collection, which notifies the exception handler if the Java program is in the middle of garbage collection. This enables us to count the exact number of instructions, data references, and cache misses that took place during garbage collection. We find out that the percentage of system activity is small in most applications, and only user data references are used in this paper.

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 a file which contains the start address and the size of each object by traversing the entire heap memory at the end of the execution. During tracing, a 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 identifies 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. The first and the last reference are timestamped with the number of total memory references (including non-heap references) instead of the actual time. Also, for 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.

Finally, we quantitatively analyze the impact of garbage collection on the performance of Java programs. We take into account the instruction overhead of garbage collection, cache performance, and virtual memory performance caused by TLB misses and page faults. TLB misses and the number of page faults are obtained in the same way as cache misses by setting the line size to 4KB. An LRU policy is assumed for page replacements.

The JVM has two run-time flags, "`-ms`" and "`-mx`", which specify the initial and the maximum size of heap memory. On JDK 1.1.6 for AIX, default values for these flags are 1MB and 32MB, respectively. We consider the following six heap configurations in this paper; default (`-ms1M -mx32M`), 16MB (`-ms16M -mx16M`), 32MB (`-ms32M -mx32M`), 48MB (`-ms48M -mx48M`), 64MB (`-ms64M -mx64M`), and infinite (`-ms320M -mx320M`). With the infinite heap configuration, no benchmark has any garbage collection, because we provide 320MB of heap memory, which is larger than $Heap_{max}$ values in table 1.

4. ANALYSIS

We analyze the various memory system behavior of Java programs using memory reference traces collected by JTRACE. First, we study the cache performance of Java programs including temporal locality and the effects of cache associativity. Second, we investigate the behavior of objects and their lifetime characteristics. Finally, the instruction overhead of garbage collection and the performance impact of heap size are studied.

4.1 Cache performance

Temporal locality. Figure 2 displays the fully-associative data cache miss ratios of SPECjvm98 applications under different heap configurations, where the miss ratios during

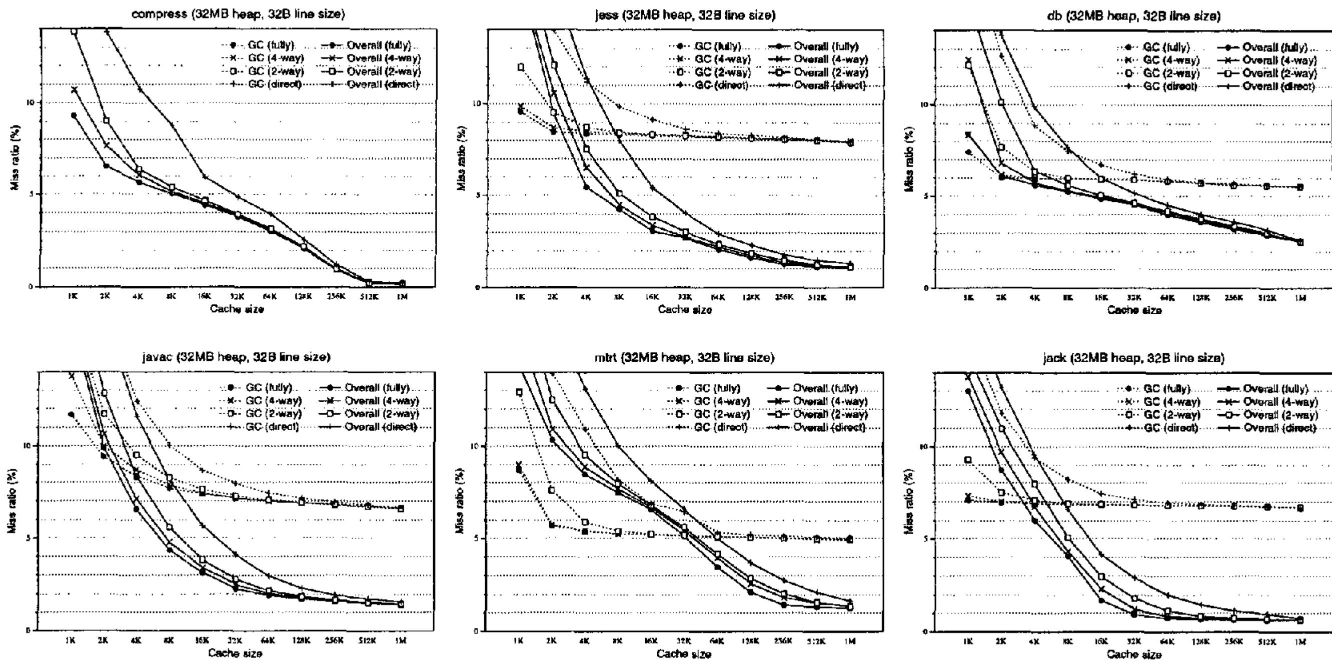


Figure 3: Effects of the cache associativity (with 32MB of heap memory).

garbage collection are separately plotted in dotted lines (labeled as GC).

First, we observe 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 will be connected to the total heap size. However, the garbage collector shows the first working set at near 2KB in most applications which is independent of the heap size. Since the garbage collector suffers from higher cache miss ratios than the application itself, all the SPECjvm98 applications studied in this paper show the lowest cache miss ratios with the infinite heap configuration, where the applications do not experience any garbage collection (marked as diamonds with a solid line in figure 2).

As observed in [16], 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. In addition, the garbage collector can move objects, which may improve (or degrade) the application's locality. Indirect costs of garbage collection are, however, found to be negligible; if we exclude data references issued by the collector, all the applications show the same curves as with the 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. For example, in the default heap configuration, 37.3% of data references are from the garbage collector in JAVAC, while 10.8% in DB. Hence, the overall miss ratio of JAVAC is more affected compared to DB. We will see in section 4.3 that the overhead of garbage collection is decreasing as we increase the heap size. Therefore, the curves of overall miss ratios approach toward that of the infinite heap with the increased heap size.

Note that JESS and JACK exhibit noticeably lower miss ratios during garbage collection with the default heap configuration. This is because, under the default heap configuration, the JVM uses less than 2MB of heap memory for these applications, which improves the garbage collector's locality. In spite of the improvement, the actual miss ratios are still worst with the default heap configuration for JESS and JACK, since the garbage collector is more frequently invoked due to the small heap size (cf. section 4.3). The large miss ratios in garbage collection for COMPRESS are not meaningful, because the absolute number of cache misses is very small. In COMPRESS, the garbage collector only accounts for 0.3% (with the default heap size) or less than 0.04% (otherwise) of total data references. The impact of garbage collection will be discussed in more detail in section 4.3.

Performance on direct-mapped and set-associative caches. The results presented in figure 2 are the miss ratios for fully-associative caches obtained via the stack simulation. We now consider more realistic caches such as direct-mapped and set-associative (2-way and 4-way) caches.

Figure 3 shows that the miss ratios during garbage collection are finally converged to the same level, no matter which cache associativity is used. In fully-associative caches, the garbage collector has about 2KB (64 cache entries) of the working set size in most applications (cf. figure 2). It appears that this small number of memory blocks causes many conflicts when the cache size is small, especially in the direct-mapped caches. As the cache size increases, these blocks are more evenly distributed over the entire cache, and the conflict misses become insignificant.

Looking at the overall miss ratios in figure 3, 2-way set-

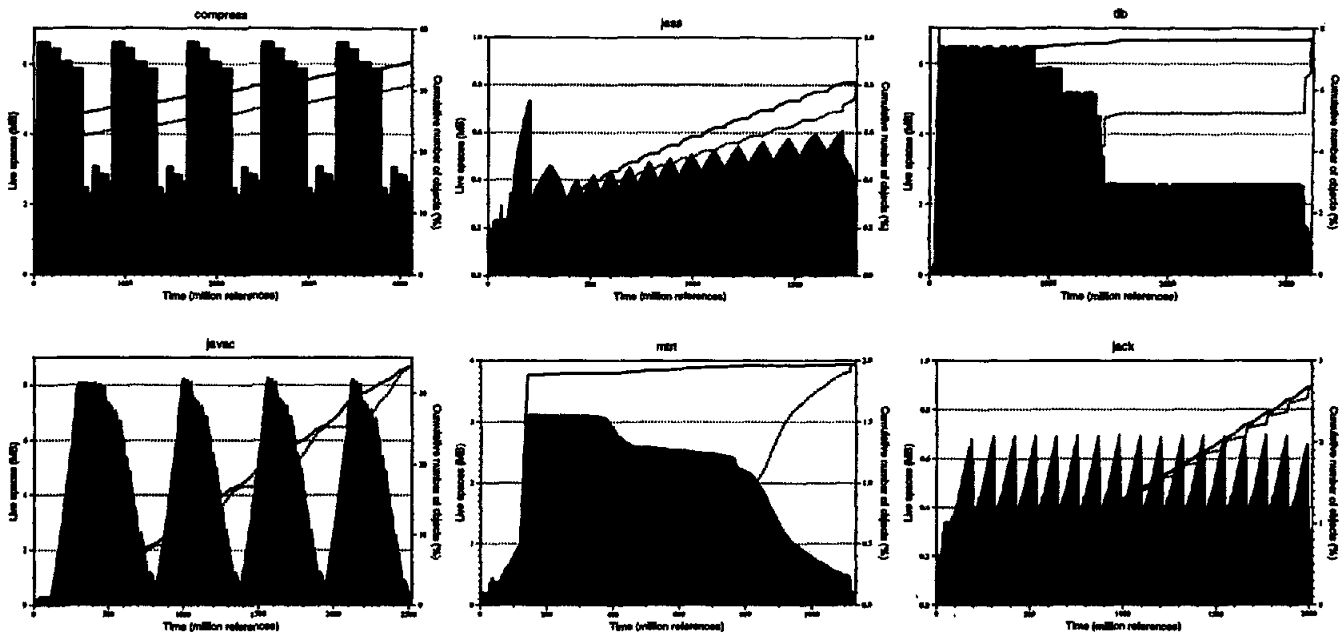


Figure 4: Changes in the number and the space of objects which last more than 1 million references.

associative caches perform very well under practical cache sizes; 2-way set-associative caches lead to a large improvement over direct-mapped caches, while the additional advantages of 4-way set-associative caches are small. The similar observation has been reported for several SPEC95 benchmarks [2] and for standard ML programs [6].

4.2 The behavior of objects

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 object links from the *root set*. The root set usually includes the local variables in run-time stacks and static variables defined in loaded classes. We will call an object *long-resident*, if it is reachable for a relatively long time, to distinguish it from a long-lived object. In this section, we analyze the characteristics of live objects and its implication on the cache performance. The reachable objects are more important to understand and model the cost of garbage collection, and will be discussed in section 4.3.

The behavior of live objects. As mentioned in section 3.3, 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 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 occupying a heap space.

Figure 4 plots the changes of live objects and their sizes, as each program executes. To reduce the number of data points, every 1 million references are grouped together and we exclude the objects whose lifetimes are less than 1 million references. The upper solid line in figure 4 represents the cumulative percentage of objects which come into existence, while the lower dotted line denotes the cumulative percentage of dead objects. These lines are associated with the right-hand scale. The difference between two lines is the percentage of live objects and the gray region indicates the size of heap memory occupied by the live objects.

The repeated patterns of COMPRESS, JAVAC, and JACK clearly demonstrate the benchmarks' loop structures over the same input data. In COMPRESS, most of live spaces come from two buffers allocated for each input file, while the compression phase uses some additional tables. The number of live objects is roughly constant and only 12 – 15 objects are newly created and destroyed for each (de)compression phase. For JESS, the first peak is for the word puzzle, and the remaining 14 peaks for the number puzzles. Because the older sets are not retracted during the number puzzles, the number of live objects are slightly increasing and it takes longer to solve the same puzzle. However, the amount of objects kept across the iterations is quite small. In DB, there are several points that many objects suddenly die, which is due to the ceased access to a certain database field. For example, the size of live objects drops quickly around the 870 millionth reference when the benchmark sorts database records on the second field. All the objects that store the second field of database records become dead, since the field is not used anymore after this sort operation. The live spaces are gradually decreasing in MTRT, probably because some of scene data are not referenced as the rendering proceeds. We note that, unlike other benchmarks, most of live objects are allocated at the very beginning in DB and MTRT.

Our measurements show that the size of live objects remains very small compared to the total heap memory allocated. For instance, JACK demands about 14MB of heap memory up to the 200 millionth reference, but only less than 0.7MB are used for the objects which survive more than 1 million references. This suggests that Java programs gener-

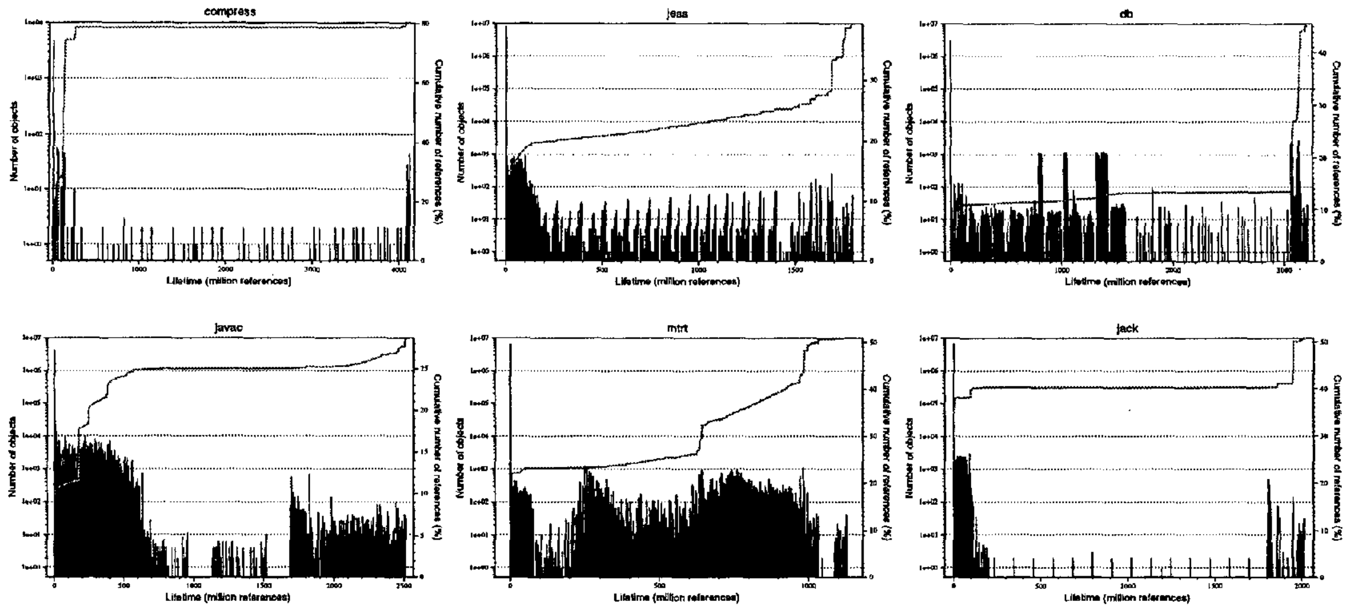


Figure 5: The distribution of lifetimes and the number of references.

ate a significant number of short-lived objects during their execution which last less than 1 million references. From the cumulative percentages shown in figure 4, we can estimate that 65% to 99% of the total objects are such short-lived objects. With the exception of COMPRESS, 71% to 99% of the heap memory is used for them. In the case of COMPRESS, although 65% of the total objects are short-lived, they only occupy 0.2% of the heap memory because some of other objects (buffers) are relatively huge in their sizes.

The lifetime characteristics. Figure 5 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.

As we observed in figure 4, a large percentage of objects have lifetimes less than 1 million references and are located at the leftmost of the graph ($x = 0$). For the programs which have a loop structure (COMPRESS, JAVAC, and JACK), the lifetimes of most objects are less than the duration of each loop. For example, figure 4 shows that each iteration of JACK makes about 110 million references. Nearly all objects are created, used, and dead inside of each iteration, which makes them located in the region less than 110 million references in figure 5. In DB and MTRT, more objects are distributed toward the right-hand side of the graph.

The dotted curves in figure 5, associated with the right y-axis, show 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 of objects are short-lived, a substantial number of references are 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 lifetimes less than 1 million references, but only account for 4% of total data references. The final value of the dotted curve in figure 5 gives us the percentage of heap references to the total data references, which ranges from 29% to 80%.

Table 2 summarizes the lifetime characteristics of studied 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. We note that the studied applications allocate 10 – 60 bytes in every 1,000 instructions. This allocation rate is much slower than that of Standard ML, which is known to allocate about one word for each six user instructions [6; 3], but quite similar to Scheme's rate (10 – 54 bytes per 1,000 instructions [16]). The reason of higher allocation rate in Standard ML is that it allocates function activation records (*closures*) on the heap, instead of using a stack as do most other languages including Java and Scheme.

In most SPECjvm98 applications, only less than half of the total data references are going to the heap memory, as shown in table 2. The rest of data references will go either to *Java stacks* or to *C stacks*. A Java stack is used for local variables, operand stacks, and method invocation, while a C stack supports native methods written in languages other than Java [13]. The JIT compiler itself also makes data references when it compiles bytecodes to native instructions. Table 2 shows 2.4% to 35.5% of (user) instructions are spent on the shared library routines which include the standard Java native methods, the JVM run-time system, and the JIT compiler. The highest percentage of JAVAC is, in part, due to the translation overhead of the JIT compiler, since JAVAC has the largest number of (static) bytecodes to compile among SPECjvm98 applications.

The implication on the cache performance. We now explore the implication of the behavior of live objects on the cache performance. 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 such short-lived objects have good

Table 2: The dynamic statistics of SPECjvm98 benchmarks (with the infinite heap configuration).

	COMPRESS	JESS	DB	JAVAC	MTRT	JACK
Instructions executed ($\times 10^3$)	11,100,907	5,327,589	9,167,372	7,716,669	3,916,519	6,552,677
Data references ($\times 10^3$)	4,121,794	1,797,615	3,211,373	2,515,695	1,129,016	2,014,066
% instructions from shared lib.	2.41%	21.99%	13.25%	35.53%	14.92%	31.04%
% heap references	80.27%	39.40%	45.61%	28.71%	50.97%	50.74%
Heap memory allocated (KB)	107,787	314,533	99,927	221,206	164,444	207,550
Average allocation rate (bytes/1000 instructions)	10	60	11	29	43	32
Average lifetime (million refs.)	155.6	1.3	129.5	55.2	10.8	1.6
Average number of references						
per each object	444,366	87	449	116	86	147
per cache line (32B)	959	70	458	102	109	154
Lifetime < 1 million references						
% objects	64.40%	99.19%	92.33%	66.20%	98.03%	97.33%
% heap memory	0.21%	98.88%	71.15%	68.64%	97.60%	95.48%
% heap references	0.0061%	37.15%	4.14%	26.47%	42.91%	67.97%
% total data references	0.0049%	14.64%	1.89%	7.60%	21.87%	34.48%

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 (70 in JESS and 154 in JACK for 32-byte line size). Consequently, the short-lived objects produce a large 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 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 almost disappear. For instance, 244 objects (33KB in size) account for about 9% of data references in JACK, with the lifetime of around 1950 million references (cf. figure 5). 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, JAVAC 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 JAVAC and MTRT, the most-frequently-referenced long-lived objects appear to fit into about 64KB and 256KB respectively, where the applications exhibit the first working set sizes (cf. figure 2). Still, a significant number of references are made to other long-lived objects, thereby resulting in 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, 28% and 35% of misses are due to capacity misses for JAVAC and MTRT respectively, while the percentage for DB is 96%.

When the cache is not fully-associative, the application experiences conflict misses, which explain the differences between the miss ratios of fully-associative cache and set-associative caches in figure 3. Reinhold [16] showed that the sequential allocation scheme is naturally suited to direct-mapped caches, because it tends to spread the objects spatially throughout the cache. He also noted, when most objects are short-lived, they are dead by the time their cache lines are reused for newer dynamic objects, based on his observation of garbage-collected Scheme programs. Among our benchmark programs, JESS and JACK have similar characteristics as Scheme programs in the sense that they have few frequently-referenced long-lived objects and that most objects are allocated rather linearly and short-lived. However, when comparing the miss ratio curves of JESS and JACK to other Java applications in figure 3, we do not see any significant benefit of direct-mapped caches in these applications. One possible explanation is that, unlike Scheme programs, many conflict misses are produced by the interferences from non-heap references, which are mostly to Java stacks and C stacks. As we move to 2-way set-associative caches, such conflict misses are reduced significantly.

Finally, we mention that the characteristics of COMPRESS is very different from other benchmark programs; the computation is done with the small number of large objects without creating many temporary objects. The garbage collection can be done quickly, hence the performance of the application is not affected by the heap size. In the perspective of the memory usage patterns, we believe COMPRESS is not much different from the corresponding C version in SPEC95.

Table 3: The number of garbage collections and the instruction overhead (shown in parentheses).

	JESS		DB		JAVAC		MTRT		JACK	
Default	687	(116.7%)	38	(16.0%)	90	(72.2%)	71	(62.6%)	301	(36.9%)
16MB	21	(10.1%)	11	(5.8%)	29	(28.4%)	17	(25.7%)	13	(5.2%)
32MB	10	(7.9%)	3	(2.9%)	9	(11.1%)	6	(15.7%)	6	(4.3%)
48MB	6	(6.6%)	2	(2.7%)	5	(7.3%)	3	(11.6%)	4	(4.3%)
64MB	4	(5.6%)	1	(1.1%)	3	(4.9%)	2	(10.4%)	3	(4.2%)

4.3 Impact of garbage collection and heap size

The instruction overhead of garbage collection. Before we analyze the performance impact of garbage collection and heap size, we examine one of the direct costs of garbage collection, the instruction overhead. Note that our analysis is highly dependent on the garbage collection algorithm employed in the JVM. In this paper, we only deal with the "mark-and-sweep" collection algorithm used in IBM JDK 1.1.6.

The mark-and-sweep algorithm collects garbages in three phases. The *mark phase* visits and marks all reachable objects starting from a special set of object links called the root set. After the reachable objects have been identified, the remaining objects are known to be garbages and can be reused. In the *sweep phase*, the collector sweeps sequentially through the heap memory, adding unmarked objects to the free list of objects. Finally, the *compaction phase* relocates the reachable objects to eliminate dead space between them and to improve the spatial locality of objects. In the current implementation, the compaction phase may not be called if the heap memory is not fully fragmented.

Table 3 compares the number of garbage collections occurred with different heap configurations. The numbers in parentheses denote the ratio of the number of instructions executed for garbage collection to the number of instructions for the application itself. We exclude COMPRESS, because the overhead of garbage collection is too small. One notable thing in table 3 is that the garbage collector is invoked very frequently in the default heap configuration, especially for JESS and JACK. With the default heap configuration, the JVM initially sets the available heap size to 1MB. The heap is increased incrementally up to 32MB if the collector fails to satisfy a new allocation request after garbage collection. For JESS and JACK, the collector does not increase the heap beyond 2MB, because the size of reachable objects is very small and the collector can still find some space after garbage collection. However, the available space after garbage collection is usually less than 1MB and this small space is quickly exhausted, which leads to another garbage collection. Due to this problem, the default heap configuration results in great inefficiency, especially when the size of reachable objects is small. As we can see in table 3, setting the initial heap size to at least 16MB significantly reduces the number of garbage collections. The default heap configuration may improve the cache (cf. figure 2) and the virtual memory performance because the application has smaller data footprints. The benefit is, however, easily overwhelmed by the additional costs of garbage collection.

The instruction overhead of garbage collection, I_{gc}, can be given by:

$$I_{gc} = N_{gc} \times W_{gc} = N_{gc} \times (W_{mark} + W_{sweep} + W_{compact})$$

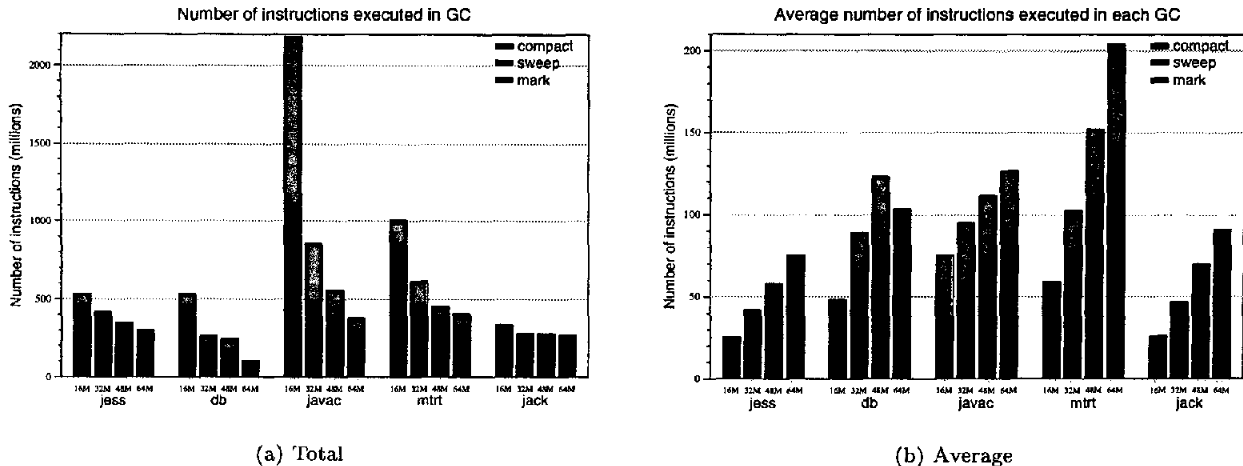
where N_{gc} and W_{gc} denote the number of garbage collections (shown in table 3) and the average number of instructions executed for each garbage collection, respectively. W_{mark}, W_{sweep}, and $W_{compact}$ represent the average number of instructions spent on each mark, sweep, and compaction phase. Figure 6 shows the changes in I_{gc} (figure 6(a)) and W_{gc} (figure 6(b)) with different heap sizes. Even though W_{gc} increases with the larger heap size, the increasing rate of W_{gc} is slower than the decreasing rate of N_{gc}. Thus, I_{gc} is always decreased as we increase the heap size.

Figure 6 also breaks down the cost of each garbage collection phase. The collector has to visit every reachable objects during the mark phase, which makes W_{mark} depend on the number of reachable objects. Our measurements show the size of reachable objects remains constant or changes very regularly for studied applications, and this is the reason W_{mark} is roughly independent of the heap size for a given application in figure 6(b).

On the contrary, the cost of the sweep phase, W_{sweep}, is directly proportional to the heap size. More specifically, it appears that W_{sweep} depends on the total number of objects (the reachable objects plus garbages) in the heap memory, because the collector should check every object to see if it is marked or not. Therefore, MTRT, which has the smallest average object size (cf. table 1), spends longer time on the sweep phase than any other applications.

For the cost of the compaction phase, we have two trends; for DB and MTRT, the total cost of the compaction phase ($N_{gc} \times W_{compact}$) is roughly constant, while the average cost ($W_{compact}$) is constant for JESS, JAVAC, and JACK. We have observed in figure 4 that most of long-resident objects in DB and MTRT are created at the beginning of the execution. Once the long-resident objects are compacted, they do not have to be compacted again in the subsequent garbage collections, which makes the total cost of the compaction phase constant. Other applications, however, have a loop structure, in which new objects are generated and replace the current reachable objects. Therefore, the garbage collector finds uncompacted reachable objects every time it is invoked. Since the size of reachable objects does not change much, the average cost of each compaction phase is constant. Note that in DB with 64MB of heap memory, the garbage collector skipped the compaction phase because it thought the heap was not fragmented yet.

Overall, the decrease in I_{gc} with the larger heap size is mainly due to the decrease in the cost of the mark phase ($N_{gc} \times W_{mark}$) and sometimes in the cost of the compaction phase ($N_{gc} \times W_{compact}$). When the size of reachable objects is very small, the garbage collector spends most of its time on the sweep phase, as can be seen in JESS and JACK. While the results presented in this section are obtained by turning off the asynchronous garbage collector, we have verified that the results with the asynchronous garbage collector are also similar.



(a) Total (b) Average

Figure 6: The breakdown of the number of instructions executed for garbage collection.

The performance impact of garbage collection and heap size. The cache miss ratios shown in section 4.1 are sometimes misleading when we analyze the performance impact of garbage collection and heap size. The lower cache miss ratio does not always guarantee the faster execution time due to the direct and indirect costs of garbage collection. Instead, Reinhold [16] used the CPI (Cycles per instruction) contribution of memory system overheads (O_{mem}) as a performance metric, which can be rephrased as follows:

$$O_{mem} = \frac{(M_{app} + M_{gc}) \times P + I_{gc} + \Delta I_{app}}{I_{app}}$$

where I_{app} and I_{gc} represent the total number of instructions executed by the program and by the garbage collector, respectively. P is the cache miss penalty in processor cycles. M_{app} and M_{gc} denote the number of cache misses for the application itself and for the garbage collector, respectively. ΔI_{app} is the change in I_{app} due to the garbage collector or the different heap size[3].

In Java programs, ΔI_{app} can be caused by initializing such data structures as mark and allocation bits, whose sizes are dependent on the heap size. However, we find ΔI_{app} is negligible compared to the other factors. We have observed in section 4.1 that M_{app} does not change much as we use the different heap sizes. Therefore, the direct costs of garbage collection, M_{gc} and I_{gc}, are the dominant factors that determine the changes in O_{mem} with different heap sizes. Because these direct costs are decreasing as we increase the heap size (cf. figure 6), the larger heap size is always favored to get the smaller O_{mem}.

When considering the virtual memory performance, this is not true. If the heap size is increased beyond the certain point, the virtual memory performance begins to play an important role in the overall performance. Table 4 shows the number of pages touched by the applications. The numbers in parentheses display the number of page faults when we assume only 32MB (8K pages) of physical memory is available for data. Normally, our Java applications touch another 900 to 3,200 pages for non-heap data. These pages will include the Java stacks, C stacks, run-time data structures used by the JVM and native code pages managed by the JIT compiler. The number of these additional pages is slightly increasing as we increase the heap size.

Even when we have enough physical memory, the first access to each page incurs a *zero-filled page fault*, where the page fault handler simply allocates a free physical page frame and zero-out the page. We find that the cycle penalties due to zero-filled page faults are also comparable to I_{gc} and cache penalties. Therefore, on the real machines, there exists an optimal heap size for a given application which minimizes the total execution time due to the interaction between these factors. The studied applications show 0.43% to 0.90% of miss ratios for 64-entry fully-associative TLBs. However, their impact on the overall CPI is very small for the simulated heap sizes.

From table 4, it is apparent that the heap size should not be increased beyond the available physical memory. An interesting point is that sometimes it is better to continue allocating new objects through virtual memory rather than to invoke the garbage collector. In all applications, the infinite heap configuration (320MB) shows less number of page faults than the case with 64MB. This is a natural consequence of the current simple mark-and-sweep garbage collector, which is not optimized for the virtual memory performance unlike the generational garbage collectors. Because most objects are short-lived, many pages need not be accessed again once they are evicted from the physical memory. However, the mark-and-sweep collector re-touches every page during garbage collection. This will not only increase the number of page faults, but also increase the traffic between the memory and the secondary storage.

5. CONCLUDING REMARKS

This paper studies the memory system behavior of JIT-compiled Java programs based on the memory reference traces obtained by an exception-based tracing tool. Specifically, we examine the cache performance, the lifetime characteristics of Java objects, and the performance impact of garbage collection and heap size.

[3]When we assume a perfect instruction cache and an ideal pipeline which completes one instruction in every cycle, the memory system overhead, O_{mem}, is the only source that increases the CPI. In this case, the overall CPI is given by $CPI = 1 + O_{mem}$. The similar performance metric has been also used for [6; 3].

Table 4: The number of pages touched by the SPECjvm98 applications and the number of page faults when we assume only 32MB of physical memory is available for data (shown in parentheses).

	JESS		DB		JAVAC		MTRT		JACK	
Default	1541	(0)	3804	(0)	5538	(0)	5421	(0)	1626	(0)
16MB	5284	(0)	4995	(0)	6533	(0)	6818	(0)	5381	(0)
32MB	9572	(58403)	9282	(18832)	10823	(69110)	11047	(56939)	9669	(47361)
48MB	13860	(95130)	13570	(39015)	15110	(156946)	15334	(83164)	13956	(87869)
64MB	18148	(118464)	17858	(37664)	19399	(115102)	19622	(82613)	18245	(94305)
320MB	83484	(85103)	29557	(30253)	61400	(79404)	47431	(51128)	56820	(57988)

All the SPECjvm98 applications studied in this paper show the lowest data miss ratios with the infinite heap configuration. This is because the garbage collector suffers from higher cache misses and tends to inflate the overall cache miss ratios. Additionally, we find that going beyond 2-way cache associativity improves the cache miss ratio marginally. 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. Finally, the direct costs of the garbage collection are the dominant factors that affect the overall performance with different heap sizes. These costs are decreasing as we increase the heap size, but the optimal heap size which minimizes the total execution time can be determined by considering cache performance, the instruction overhead of garbage collection, and the virtual memory performance.

Since the SPECjvm98 benchmark suite can not be a single representative of all the Java programs and Java is an evolving language and environment, the characterization of Java programs should be an on-going effort. In this paper, we only studied the mark-and-sweep garbage collector, originally come from the Sun's standard implementation of JDK 1.1. Recently, Sun has announced the HotSpot technology (JDK 1.2), which employs an improved generational copying collector. Therefore, it is necessary to compare and contrast the performance of these garbage collectors. In addition, since most of our benchmark programs are single-threaded, the characteristics of multithreaded Java programs needs to be investigated further to completely understand the behavior of Java programs. We believe the methodology used in this paper is directly applicable to other Java applications and environments, and we plan to continue our study of characterizing Java programs.

6. ACKNOWLEDGMENTS

The authors would like to thank Luc Smolders and Patrick Bohrer in IBM Austin, whose helps were essential to the development of the tracing tool described in this paper.

7. REFERENCES

[1] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *ISCA-25*, pages 3-14, 1998.

[2] S. Dasgupta and E. Servan-Schreiber. Cache behavior of the SPEC95 benchmark suite. University of California at Berkeley, `http://http.cs.berkeley.edu/~dasgupta`, 1996.

[3] A. Diwan, D. Tarditi, and E. Moss. Memory-system performance of programs with intensive heap allocation. *ACM Trans. on Computer Systems*, 13(3):244–273, Aug. 1995.

[4] J. Edler and M. D. Hill. Dinero IV trace-driven uniprocessor cache simulator. `http://www.neci.nj.nec.com/homepages/edler/d4/`, 1998.

[5] J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[6] M. J. R. Gonçalves and A. W. Appel. Cache performance of fast-allocating programs. In *Proc. of the ACM Conf. on Functional Programming Languages and Computer Architecture*, pages 293–305, 1995.

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

[8] 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*, pages 211–216, 1997.

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

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

[11] 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*, pages 212–213, 1991.

[12] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *PLDI*, pages 291–300, 1995.

[13] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[14] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *ASPLOS VI*, pages 145–156, 1994.

[15] R. Radhakrishnan, J. Rubio, L. K. John, and N. Vijaykrishnan. Execution characteristics of just-in-time compilers. Technical Report TR-990717-01, University of Texas at Austin, 1999.

[16] M. B. Reinhold. Cache performance of garbage-collected programs. In *PLDI*, pages 206–217, 1994.

[17] T. H. Romer et al. The structure and performance of interpreters. In *ASPLOS VII*, pages 150–159, 1996.

[18] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *PLDI*, pages 196–205, 1994.

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

[20] T. Suganuma et al. Overview of the IBM Java just-in-time compiler. *IBM Systems Journal*, 39(1):175–193, Jan. 2000.

[21] P. Tyma. Why are we using Java again? *Commun. ACM*, 41(6):38–42, Jun. 1998.

[22] B. G. Zorn. The effect of garbage collection on cache performance. Technical Report CU-CS-528-91, University of Colorado at Boulder, May 1991.