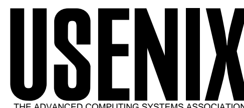
USENIX Association

Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '01)

San Antonio, Texas, USA
January 29 - February 2, 2001



© 2001 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

HBench:JGC – An Application-Specific Benchmark Suite for Evaluating JVM Garbage Collector Performance

Xiaolan Zhang and Margo Seltzer
Division of Engineering and Applied Sciences
Harvard University

Abstract

As Java becomes a viable platform for server applications, performance becomes a greater concern. An important aspect of Java Virtual Machine performance is its dynamic memory management system (*garbage collection* or *GC*). Traditional GC benchmarking often focuses on a set of fixed applications. As a result, when an actual application's memory behavior differs from that of the standard benchmarks, the benchmark results do not help the user judge which GC implementation suits her application the best. In this paper, we present HBench:JGC, an application-specific benchmarking suite, based on the idea that a system's performance be measured in the context of a specific application. HBench:JGC employs a methodology that characterizes the application memory usage and the GC implementation independently and carefully combines both characterizations to form a single metric that reflects a particular application's performance in the presence of a particular GC implementation. We evaluate our approach on Sun Microsystems's JDK1.2.2 classic JVM with a sequential mark-sweep GC. Our results demonstrate HBench:JGC's unique predictive power and its ability to provide meaningful metrics that lead to a better understanding of GC performance.

1. Introduction

In recent years, there has been a rapid increase in the adoption of Java technology in a variety of environments, ranging from JVMs embedded in web-browsers to high-performance server products. As Java becomes a viable platform for server applications, performance becomes a greater concern. An important piece of Java Virtual Machine performance is its dynamic memory management system (*garbage collection* or *GC*). Historic data show that it is quite common for garbage collection to account for 20% or more of an application's total running time [9]. Sometimes garbage collection is the performance bottleneck. Understanding GC performance and selecting the right GC implementation, therefore, can lead to significant savings in the total running time of the application.

The traditional GC benchmarking approach is to pick a set of programs, run them with different GC algorithms, and compare the total elapsed times. This approach has been used by Smith and Dorisett, as well as by Zorn [12][15]. This approach is inadequate, since the optimal GC algorithm varies with the application [15], and the set of benchmark programs may not represent the actual memory behavior of the application of interest.

Another approach to benchmarking and selecting GC algorithms for a given application is to manually construct a small program that models the memory behavior of the application in question and when run, produces the same memory footprint. This approach requires a high level of skill and is error-prone, especially when the application's memory behavior is complicated.

HBench:JGC is a benchmark suite that allows one to measure GC performance in the context of the applications in which users are interested without having to model the applications manually. The underlying principle is to separate the characterization of application memory usage from that of the GC implementation. HBench:JGC includes a GC-independent profiler that traces an application's memory behavior. It uses a set of microbenchmarks to measure the performance of a GC implementation in an application-independent way. The two characterizations are then fed to an analyzer, which calculates the predicted GC time. Figure 1 depicts the schema of HBench:JGC. HBench:JGC has the added advantage that one can use it to predict an application's garbage collector performance on a target GC implementation without actually running the application with the particular collector, as long as its performance characteristics are available. The GC-independence of the application characterization facilitates this unique flexibility. We

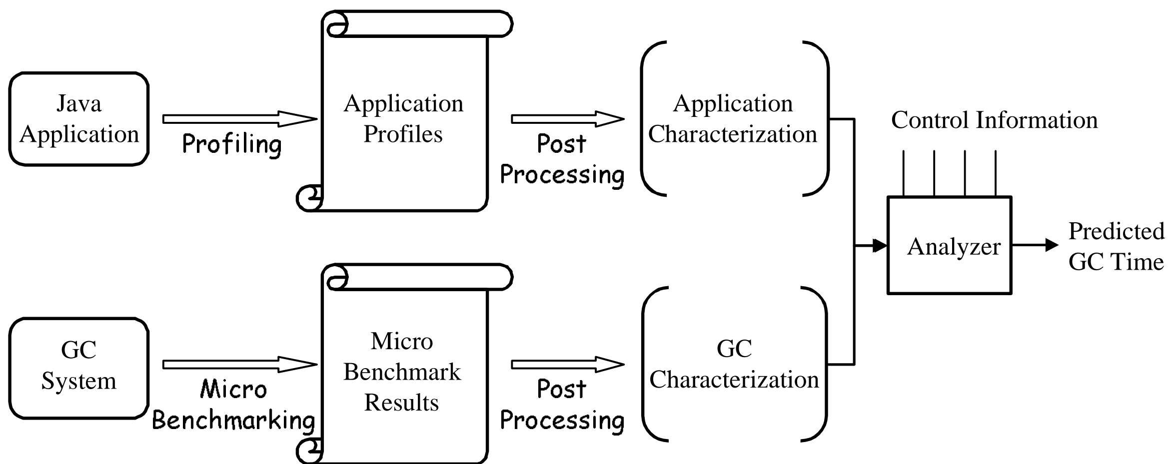


Figure 1. Schematic View of HBench:JGC Process

can predict the application's performance on different GC implementations by feeding performance characteristics of different GC implementations to the analyzer.

Section 2 describes the design of HBench:JGC in detail. Section 3 describes its prototype implementation. Section 4 presents experimental results on applying HBench:JGC to evaluating GC performance. Section 5 discusses open issues and future work. Section 6 describes related work and Section 7 concludes.

2. HBench:JGC Design

HBench:JGC is part of HBench, an application-specific benchmarking framework designed to address the problem that standard benchmark results do not reflect a particular application's performance on a particular system [11]. HBench:JGC is based on HBench's vector-based methodology. The principle behind the vector-based methodology is that a system's performance is determined by the performance of the individual primitive operations that it supports and that an application's performance is determined by how much it utilizes the primitive operations of the underlying system. The running time of a given application can be estimated by carefully combining the two characterizations. A simple form of this combination process would be to add up the costs of all primitive operations executed by the application. By separating characterizations of the application from that of the underlying system and by incorporating application characteristics into the benchmarking process, HBench can provide performance metrics that reflect the expected behavior of a particular application on a particular platform, as well as allow meaningful comparisons between different platforms.

Although originally designed as part of the HBench:Java benchmark suite [14], the methodology of HBench:JGC described in this paper is applicable to GC implementations for other languages such as Lisp, Scheme, Smalltalk, and C++.

2.1. GC Characterization

2.1.1. Basic GC Concepts

Like all memory management systems, a garbage collector implementation supports two primitive operations, namely, object allocation and reclamation.

The garbage collector manages the collection of free space from which new objects are allocated. The free space can be represented as a list of free blocks, a single chunk of contiguous space, or a combination of the two.

When the allocator fails to satisfy an allocation request, it initiates a garbage collection run. A garbage collection run typically starts with a *marking* phase, when live objects are identified and marked. This phase may be followed by one or more phases (typically called the *sweep* phases) that free the space occupied by the dead objects, making it available for allocation. *A non-copying* collector does not move the live objects, whereas a *copying* collector typically compacts the live objects to one end of the heap in order to create a large contiguous free space at the other end of the heap. Examples of non-copying col-

lectors include the most widely adopted mark-sweep garbage collector [1] and its variants. Examples of copying collectors include the Lisp 2 collector [8], which is a mark-compact collector, and Cheney's two-space copying collector [3]. For a complete treatment of this topic, readers are encouraged to refer to the book by Jones et al. [7].

2.1.2. A GC Implementation Taxonomy

Independent of the GC algorithms (e.g., copying vs. non-copying), we can classify GC implementations according to the four attributes described in Table 1. The first attribute represents the axis between stopping all execution for garbage collection and running the collector completely in parallel with program execution [2]. The second attribute describes the internal architecture of the collector itself, whether it is sequential (single-threaded) or parallel (multi-threaded). The third attribute describes the granularity of collection, whether collection occurs in a single, complete pass (batch-oriented) or whether just some of the available memory is reclaimed during each iteration (incremental). The fourth and last attribute distinguishes generational garbage collectors [10] from non-generational collectors. Generational collectors implement a set of heaps that are cleaned with varying frequency depending on the age of the objects stored in the heap. Each heap corresponds to a different age group.

Attributes of GC Implementations
Stop-the-world ↔ Concurrent
Sequential ↔ Parallel
Batch ↔ Incremental
Non-generational ↔ Generational

Table 1. GC Implementation Techniques

The four attributes in the taxonomy are largely orthogonal, with a few exceptions. For example, a GC algorithm can be both stop-the-world and parallel, but it cannot be both concurrent and batch mode.

In this paper we consider only sequential, stop-the-world, batch-mode and non-generational garbage collectors. We chose to start with this type of collector because it involves the fewest variables and thus allows faster prototyping of the analytical models and more controllable experimentation. Furthermore, this type of collector is still in wide use. For example, Sun's standard JDK1.1 and JDK1.2 Java Virtual Machines use this type of collector. Section 5 discusses how we envision enhancing our approach to cope with concurrent, parallel, incremental and generational garbage collectors.

2.1.3. Object Allocation

For a given memory management algorithm, the cost of object allocation is typically determined by the following two factors:

1. the size of the allocation,
2. the state of the heap, such as the number of free blocks and their sizes.

We can represent this cost with a function C_{alloc}(*heap_state, allocation_size*). Depending on the memory management algorithm, C_{alloc} carries different forms. In the case of copying garbage collectors, the free space is a contiguous area, and allocation can be implemented by a simple pointer advancement. Therefore, in the case of a copying collector, C_{alloc} is a constant function. In the case of non-copying collectors, such as a non-copying mark and sweep collector, the allocation time depends on the state of the free-block lists maintained by the collector. If we characterize the heap state with simple statistical measures, such as a normal distribution with a given mean and standard deviation, or a uniform distribution with a given range, we can represent C_{alloc} in a concise way. Furthermore, we can measure C_{alloc} using microbenchmarks that initialize the heap according to the statistical measures.

2.1.4 Object Reclamation

An interesting aspect of garbage collection performance is that the cost of dead object reclamation depends on the amount of live data on the heap, since the way a garbage collector identifies live objects is to traverse the connected object graph from a set of root objects.

We divide the cost of object reclamation into three parts: the fixed cost (C_{fixed}), the per-live-object cost (C_{live}), and the per-dead-object cost (C_{dead}). C_{fixed} corresponds to the fixed cost associated with a garbage collection run, such as the initialization of data structures. C_{fixed} normally depends only on the heap size. C_{live} is the overhead measured per live object (objects that survive

the collection). For non-copying collectors, C_{live} is typically constant. For copying collectors, C_{live} is a function of the size of live objects, as live objects are compacted (copied) at the end of a collection run. C_{dead} corresponds to the per-object cost of releasing the space of a dead object. In most cases, this involves updating bookkeeping information for the freed object, and thus C_{dead} is usually constant for a given collector algorithm. In summary, the cost of object reclamation can be represented by three functions, C_{fixed}(*heap_size*), C_{live}(*object_size*), and C_{dead}. Let N_l be the distribution function of the sizes of live objects, i.e. $N_l(s)$ is the number of surviving objects with size *s*. Let N_d be the distribution function of dead object sizes. The total cost of garbage collecting a heap of size *h* can then be calculated using the following formula (1):

$$T_{GC} = C_{fixed}(h) + \sum_s C_{live}(s) \cdot N_l(s) + C_{dead} \sum_s N_d(s)$$

The above reasoning makes the simplifying assumption that every live object is traversed exactly once during marking. For cases where an object is referenced by several live objects, the object will be visited multiple times by the collector. We characterize this additional cost by adding a second variable, d_i, the *fan-in degree* of an object, in the per-live-object overhead function C_{live}. The middle term of the formula thus becomes:

$$\sum_s \sum_{d_i} C_{live}(s, d_i) \cdot N_l(s, d_i)$$

The situation is further complicated by the fact that certain copying collectors need to update an object's references, if the objects it points to are copied to a different place. We characterize this additional cost by adding yet another variable, d_o, the *fan-out degree* of an object, in the per-live-object overhead function C_{live}. The middle term now becomes:

$$\sum_s \sum_{d_i} \sum_{d_o} C_{live}(s, d_i, d_o) \cdot N_l(s, d_i, d_o)$$

The difficulty of characterizing object reclamation costs lies in deriving the three cost functions C_{fixed}, C_{live}, and C_{dead} using results from microbenchmarks. Our experience indicates that the simplified formula (1) for estimating GC time works well in practice for a mark-sweep GC algorithm. In the future, we will include the refinements discussed above if necessary.

2.2. Application Characterization

The following metrics describe an application's memory usage behavior:

1. Object allocation rate (both in terms of the number of objects and the number of bytes);

2. Object death rate (both in terms of the number of objects and the number of bytes);

3. Object age (the time an object remains alive);

4. Connectivity of the live object graph, i.e., the number of references to an object (*fan-in degree*) and the number of references it contains (*fan-out degree*).

Some of the metrics, such as object allocation rate, can be obtained quite easily. Some other metrics, such as object age, are difficult to measure and can only be estimated using profiling tools.

One significant challenge in characterizing an application's memory behavior is that of GC (and JVM) independence. For example, if we use the number of objects per second as the unit for object allocation speed, it is not portable to other JVM or GC implementations, as this unit is system dependent. To solve this problem, we use objects per bytecode as our basic unit for both object allocation rate and object death rate.

2.3. Predicting GC Time

Object allocation cost is an important part of the performance metric of GC systems. It is, however, not directly measurable for a given application. As a first step, this paper focuses on predicting the time the application spends on garbage collection, or the time between the start and finish of a garbage collection run. Unless otherwise specified, GC time refers to the cost of object reclamation, and does not include allocation costs.

The total GC time of an application can be determined by two factors: the number of GC runs and the time for each GC run.

With the knowledge of object allocation rate and object death rate, one can estimate the amount of live data at a given execution point, from which one can then calculate the number of GCs deterministically, assuming a heap that is fixed-size or one whose growth policy is known a priori.

The time for each GC run can be estimated using formula (1) described in section 2.1.3. The total GC time is the sum of times of all individual GC runs.

3. HBench:JGC Implementation

As depicted in Figure 1, the major components of HBench:JGC are: the profiler that traces an application's memory behavior, the set of microbenchmarks whose measurement results form the characterization of the given garbage collection implementation, and finally, the analyzer that estimates the GC time given both application and GC characterizations. The following three subsections describe each component in more detail.

3.1 Profiler

Sun Microsystems's JDK 1.2.2 provides an interface called the Java Virtual Machine Profiling Interface (JVMPI) [6] that allows one to attach a profiling agent to the JVM at startup time. The agent can register for events in which it is interested through callback functions and intercept the events as they occur.

We are interested in the following events: GC start and finish, object allocation, free and move, heap dump and object dump. Object allocation and free events can be used to estimate object lifetimes and the number of free/live objects at a given execution point. Heap dumps help determine the object connectivity such as fan-in and fan-out degrees. Our current implementation includes all the events except heap and object dump.

3.2. Microbenchmarks

The goal of microbenchmarking is to measure the fixed and per-object costs of memory reclamation. Our first microbenchmark deals with singular linked list data structures. We are in the process of creating microbenchmarks that model more complicated object types with different fan-in and fan-out degrees.

The microbenchmark first populates the heap with an array of linked lists of objects. The size of array, the length of the list, and the object size can all be dynamically configured with command-line options. The microbenchmark then explicitly invokes garbage collection at three different times:

1. When all objects on the heap are alive;

2. When all objects on the heap are reclaimable, i.e., after the microbenchmark sets the pointers to the heads of the linked lists to null;

3. When the heap is entirely empty, i.e., after the GC following step 2.

To measure C_{fixed}, we run the microbenchmark with different heap sizes, fixing the other two parameters. We then plot the GC times measured in step 3 above against the heap sizes. The resulting regression formula is the approximate function for C_{fixed}.

Similarly, to measure C_{live}, we run the microbenchmark with a varying numbers of objects, fixing the other two parameters. The GC times measured in step 1 above are then plotted against the number of objects for a given object size s and the resulting regression function defines $C_{live}(s)$. Since C_{live} might also depend on object sizes, we again repeat the microbenchmark for different object sizes.

The same process is performed to measure C_{dead}, except that in this case the GC times of step 2 are used.

3.3. Analyzer

Given both the application and GC characterizations, the analyzer tries to estimate the time the application spends on garbage collection. The analyzer also needs certain configuration information, such as the heap size, in order to determine the total GC time. Note that heap sizes may change dynamically. For example, if the memory system cannot satisfy the allocation request even after a GC, or if the percentage of free space is below a certain threshold, the heap is expanded. The policies as to when and how much to expand the heap should be specified to the analyzer.

CPU	Memory (MB)	Operating System	JVM Version	GC Algorithm
Pentium Pro 200MHz	128	Windows NT 4.0	1.2.2 Classic	Mostly mark-sweep
Pentium III 550 MHz	256	Windows 2000		
UltraSPARC IIi 333 MHz	128	Solaris 7		

Table 2. Test Configurations

4. Experimental Results

4.1. Experimental Setup

We ran our experiments on Sun Microsystems's JDK1.2.2 classic version on three different machine configurations. Table 2 shows the hardware properties.

Sun Microsystems' JDK1.2.2 classic JVM uses a mark-sweep (with compaction) collector. Mark-sweep collection is one of the classical garbage collection algorithms that remains in wide usage today. Due to its conservative nature, it is popular for type-unsafe languages such as C/C++. The collector of the JDK1.2.2 classic JVM is a variation of the classical mark-sweep collector — it occasionally moves live objects around the heap. Although compaction does not occur often for the applications we tested, it does generate some uncertainties that make it harder to predict the GC time.

We use Java applications included in the SPECJVM98 benchmark suite [13] to evaluate the predictive power of our approach. Most SPECJVM98 applications induce extensive GC activities, except _222_mpegaudio, which is excluded from our set of test applications. Table 3 shows the number of bytes allocated by each test application quoted from the benchmark's documentation. The actual numbers appear to differ but the magnitude is the same.

SPEC Application	Allocation (MB)
_201_compress	334
_202_jess	748
_209_db	224
_213_javac	518
_227_mtrt	355
_228_jack	481

Table 3. GC Activity of Test Applications

4.2. Microbenchmark Results

We report the GC times of the three steps described in Section 3.2. Unless otherwise specified, all data points reported in this section are means of 10 runs of the microbenchmark. In most cases, the standard deviation is within 1%.

4.2.1. GC on Empty Heap

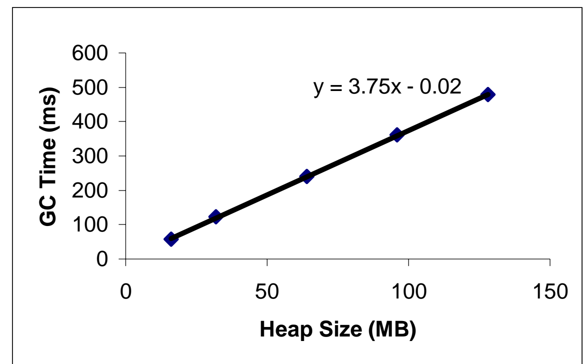


Figure 2. GC Time of Empty Heap on Sun SPARC. We use an object size of 28 bytes, and 512 lists each with 512 objects. The number of objects and the size of objects remain fixed as the total heap size varies.

Figure 2 shows the garbage collection times of an empty heap (see step 3 in section 3.2) on the Sun SPARC workstation. The regression formula indicates that GC times of empty heaps are linearly dependent on the size of the heap and that the per-megabyte cost of an empty heap GC for this particular GC implementation is 3.75ms. The y-intercept (0.02) is negligible. We therefore derive the following formula for $C_{fixed}(h)$ (described in Section 2.2) for this GC algorithm:

$$C_{fixed}(h) = 3.75 \cdot h \ ,$$

where h is the size of the heap in megabytes. The value of the slope (3.75) remains the same (variations

within 5%) for different object sizes and numbers of objects.

Similar results were obtained for the other machine configurations, albeit with a different slope value.

4.2.2. GC on Fully Reclaimable Heap

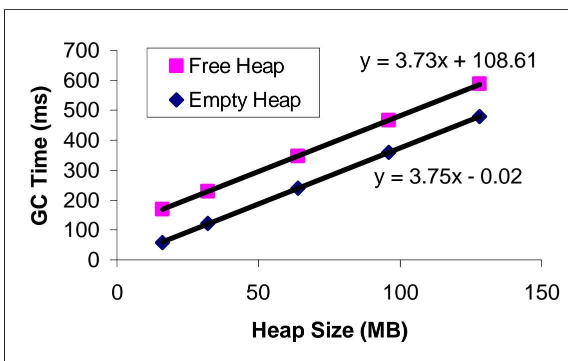
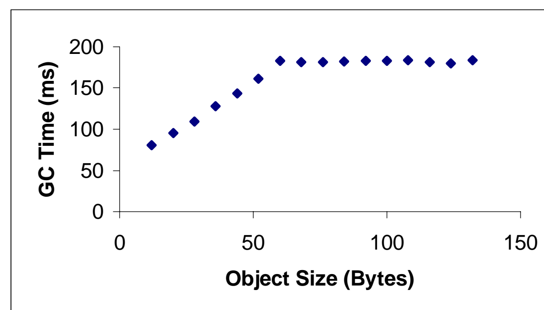


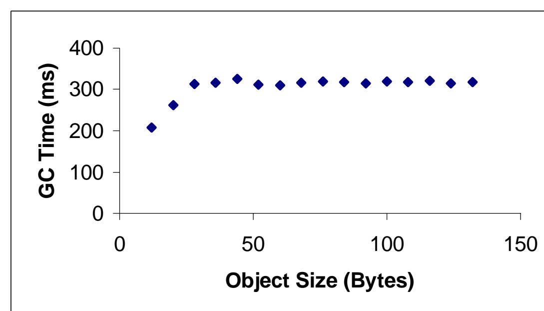
Figure 3. GC Time of Fully Reclaimable Heap With Respect to Heap Size on Sun SPARC. We use an object size of 28 bytes, and 512 lists each with 512 objects. The number of objects and the size of objects remain fixed as the total heap size varies.

Figure 3 shows the garbage collection times of a fully reclaimable heap (see step 2 in section 3.2). The GC time again shows a linear dependence on the size of the heap, and the slope value (3.73) is close to the slope value of C_{fixed} (3.75). If we remove the fixed cost C_{fixed} (empty heap), the remaining time is essentially independent of heap size. Since all objects on the heap are free and are reclaimed by the collector, this remaining time, when divided by the number of dead objects, represents the per-dead-object cost C_{dead}. In this particular case, C_{dead} takes on a value of 108.6/(512*512), or 0.4 ns/object. Again, similar results are observed from runs on the other machine configurations.

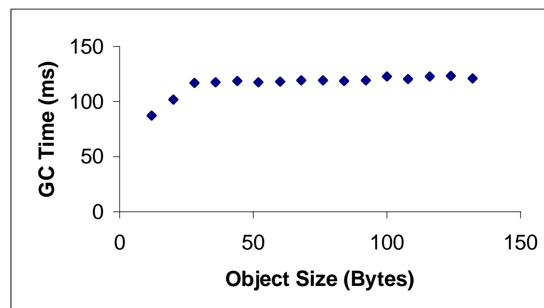
Theoretically, C_{dead} is independent of object size, since dead objects are neither scanned nor copied. However, to our surprise, our measurements suggest that C_{dead} is indeed dependent on object size. Figure 4(a) shows the results on the Sun SPARC workstation. The GC time seems to grow as the object size increases, until the object size hits 60 bytes, and stays at around 180ms thereafter. We do not have a conclusive explanation for this behavior but we hypothesize that the dependence on the object size is due to memory cache effects.



(a). Results on Sun SPARC



(b). Results on Pentium Pro



(c). Results on Pentium III

Figure 4. GC Time (Excluding Fixed Overhead) of Fully Reclaimable Heap with Respect to Object Size. The GC time is calculated from the regression formula as shown in Figure 3.

Our experiments on the other two machine configurations seem to confirm our hypothesis. Figures 4(b) and 4(c) show the results on the Pentium Pro and Pentium III machines respectively. In both cases, C_{dead} shows similar dependence patterns. C_{dead} is independent of object size, except when the object size is less than 28 bytes. The memory effects seem to be smaller for these two configurations than for the Sun SPARC workstation.

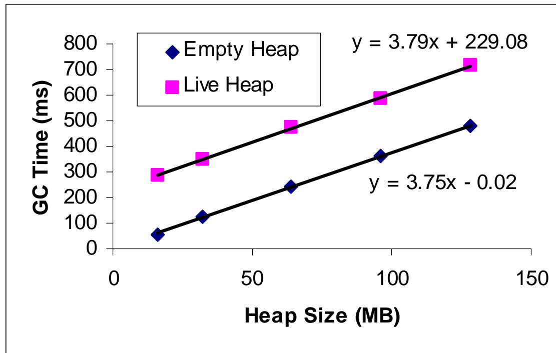
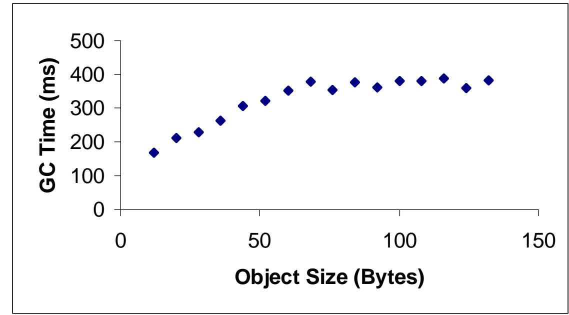


Figure 5. GC Time of Fully Live Heap with Respect to Heap Size on Sun SPARC. We use an object size of 28 bytes, and 512 lists each with 512 objects. The number of objects and the size of objects remain fixed as the total heap size varies.

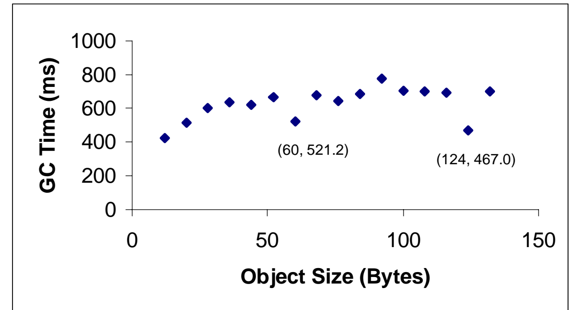
4.2.3. GC on Fully Live Heap

Figure 5 shows the garbage collection times of a fully live heap (see step 1 in section 3.2). In this case, all objects on the heap are live and survive the garbage collection. Similar to the case of a fully reclaimable heap, the GC time shows a linear dependence on the size of the heap. If we exclude the fixed cost C_{fixed}, the remaining time is independent of heap size. The GC time, when divided by the number of total objects on the heap, yields the per-live-object cost C_{live}. In this particular case, C_{live} takes on a value of 229.1/(512×512), or about 0.9ns/object. Again similar results are observed from runs on other machine configurations.

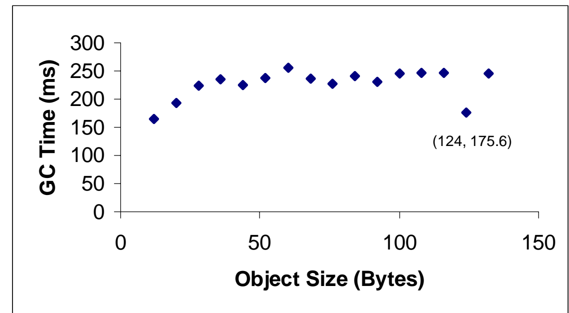
Figures 6 (a), (b) and (c) show C_{live} as a function of object size on the Sun SPARC workstation, the Pentium Pro machine, and the Pentium III machine, respectively. We observe patterns similar to those of the fully reclaimable heap case, albeit with different threshold values. For the Sun SPARC workstation case, the value of C_{live} seems to grow as the object size increases, until the object size hits 60 bytes and stays at approximately 380ms thereafter. For the Pentium Pro machine case, the value of C_{live} seems to oscillate between 600ms and 700ms after the object size hits 28 bytes. Similarly, for the Pentium III machine case, the value of C_{live} oscillates between 220ms and 250ms after the object size hits 28 bytes. Since no objects are copied, C_{live} should be independent of object size. We therefore attribute this observed dependence on object size to memory cache effects. This effect is also detected with two anomalous data points for the Pentium Pro configuration: at object sizes of 60 bytes and 124 bytes. There is also a similar anomalous data point for the Pentium III at object size of 124 bytes. We are still investigating what exact memory effect causes the anomalies.



(a). Results on Sun SPARC



(b). Results on Pentium Pro



(c). Results on Pentium III

Figure 6. GC Time (Excluding Fixed Overhead) of Fully Reclaimable Heap with Respect to Object Size. The GC time is calculated from the regression formula as shown in Figure 5.

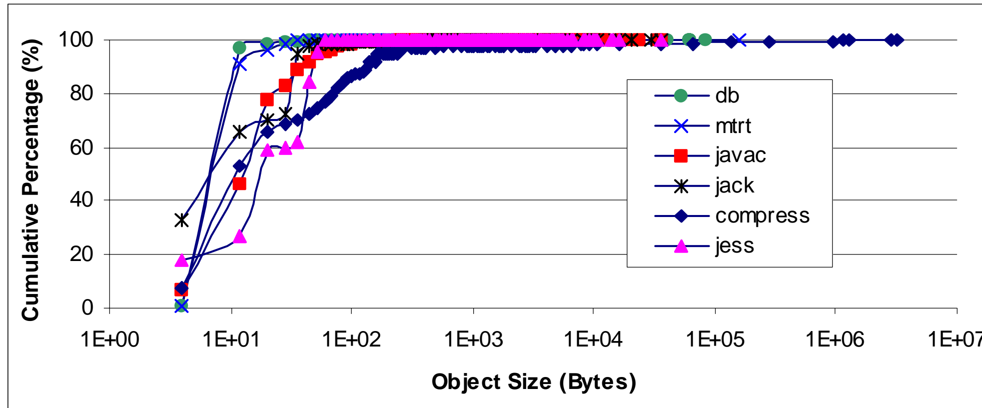


Figure 7. Cumulative Object Size Distribution in Number of Objects

4.3. Predicting GC Time

In this section we demonstrate how the microbenchmark results can be used to predict garbage collection time for a given Java application.

First, we calculate the values of the three functions that characterize a GC algorithm, namely, C_{fixed}, C_{live}, and C_{dead}. Table 4 shows the coefficient values of the three functions for the JVM on the Sun SPARC workstation. For objects with size larger than 132 bytes, the values for 132 bytes are used.

Next we obtain characterizations of the applications' memory behavior. Our current profiler implementation generates information such as the number of live objects, the number of dead objects, and the object size distribution. Assuming that live and dead objects have the same size distribution, we can approximate the GC time function T_{GC} (section 2.1.4) with the following formula

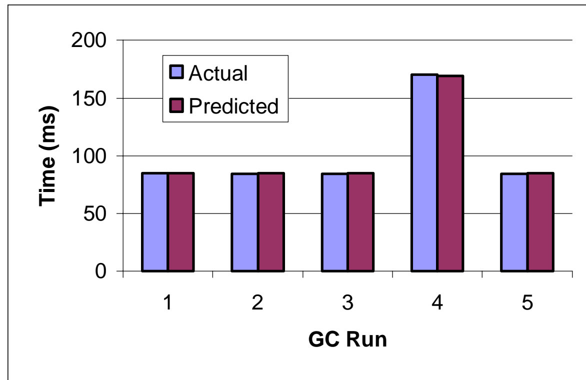
$$T_{GC} = C_{fixed}(h) + L \cdot \sum_{s} C_{live}(s) \cdot n(s) + D \cdot \sum_{s} C_{dead}(s) \cdot n(s)$$

where n(s) is the normalized object size distribution function, i.e. n(12) is the percentage of objects with size equal to 12 bytes, L is the number of live objects and D is the number of dead objects. Figure 7 shows the accumulative object size distribution function for the test applications. Applications such as `db` and `mtrt` are dominated by one object size, whereas other applications use multiple object sizes. In general, the majority (more than 90%) of objects are small, i.e., less than 100 bytes, except for `compress`.

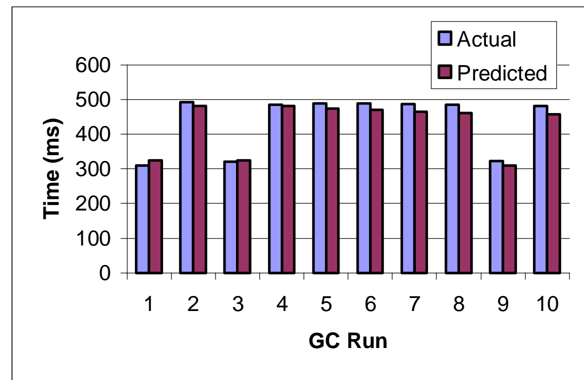
So far our formula has not taken into consideration the cost of the occasional copying performed by the collector. For our test cases, copying only occurred in two applications in four GC invocations (out of a total of

Object Size	C_{fixed} Per MB	C_{live} Per Object	C_{dead} Per Object
12	3.75	7.04E-04	3.02E-04
20	3.75	7.51E-04	3.49E-04
28	3.75	8.67E-04	4.03E-04
36	3.75	9.55E-04	4.71E-04
44	3.75	1.07E-03	5.49E-04
52	3.75	1.24E-03	6.15E-04
60	3.75	1.30E-03	6.83E-04
68	3.75	1.38E-03	6.85E-04
76	3.75	1.44E-03	6.83E-04
84	3.75	1.41E-03	6.85E-04
92	3.75	1.59E-03	6.85E-04
100	3.75	1.40E-03	6.82E-04
108	3.75	1.33E-03	6.87E-04
116	3.75	1.40E-03	6.91E-04
124	3.75	1.33E-03	6.92E-04
132	3.75	1.46E-03	7.17E-04

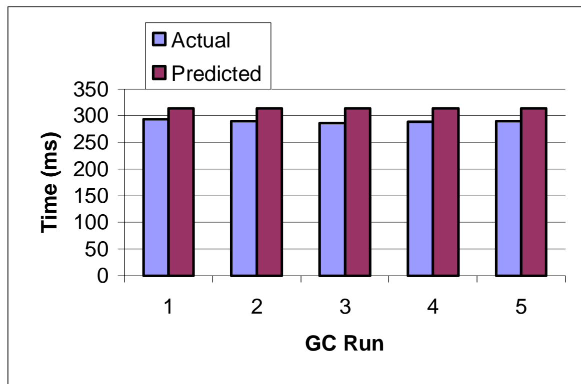
Table 4. GC Characteristics on 333MHz UltraSPARC IIi



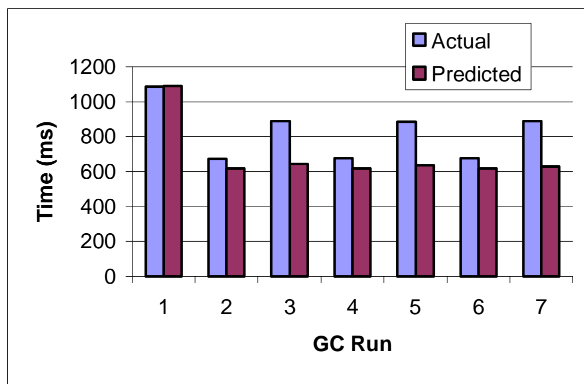
(a). _201_compress



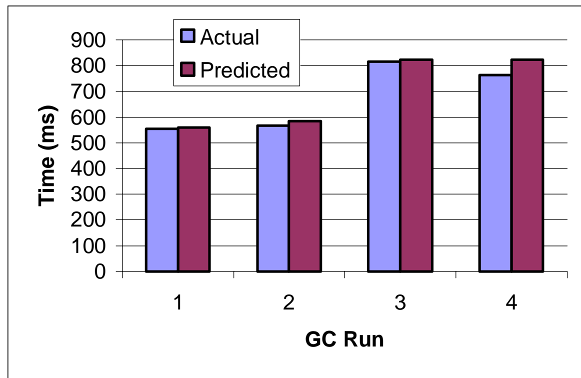
(b). _202_jess



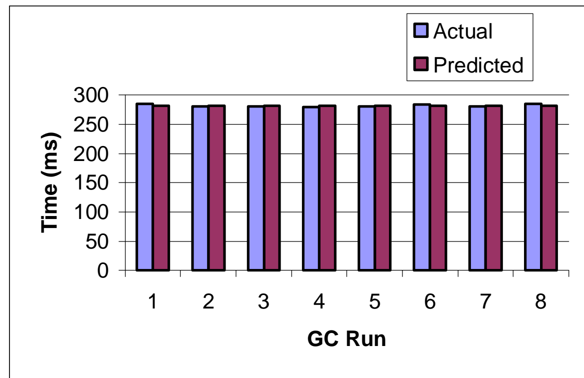
(c). _209_db



(d). _213_javac



(e). _227_mtrt



(f). _228_jack

Figure 8. Predicted versus Actual GC Times. All tests were run on the Sun SPARC workstation using a heap size of 32MB, except for javac and mtrt, which were run on a heap size of 64MB to eliminate the variation on the number of GCs from different runs.

thirty-five GC invocations). Three of those four GC invocations were explicit garbage collections made by the application, which trigger unnecessary copying. Currently we approximate this copying overhead by dividing the number of bytes copied over the memory bandwidth, and we use the actual number of bytes copied. In the future, we will enhance our analyzer to estimate this information from the application memory

characterization, assuming that the algorithm that decides when to perform a copy is known. We will also explore techniques to design microbenchmarks that would trigger a copy and measure the cost directly.

Figure 8 shows the predicted versus actual GC running times for the six SPEC applications on the Sun SPARC workstation. A summary of the percentage time difference between the predicted and the actual GC times is presented in Table 5.

For compress (Figure 8(a)), there are five garbage collections during the execution of the compress application. The predicted GC times match the actual times quite closely (with 0.2% error rate), showing that our prediction model works well in this case. In the fourth GC run, the collector copied certain live objects to the beginning of the heap, which accounts for the boost in the GC time. The result shows that our approximation on the copying time works well in this case also.

Figures 8(b), 8(c), 8(e) and 8(f) show the results for jess, db, mtrt and jack, respectively. The predicted times track the actual times quite closely. No copying occurred in these cases.

Figure 8(d) shows the results for javac. The predicted times track the actual times nicely except for the 3rd, 5th, and 7th GC runs. It turns out that these three GCs were invoked explicitly by the application at times when the heap space had not been exhausted and most objects on the heap were live objects. The explicit GCs also trigger unnecessary copying of live objects. In this case, our approximation on the copying cost does not work well. This might be due to the fact that the approximation does not include the overhead for initiating a copy, therefore it underestimates the cost in cases when many small objects are copied.

In summary, HBench:JGC is able to predict the actual GC times within 10% for five out of the six applications (Table 5). In the case of javac, the error rate is –6.4% if we disregard the three explicit GCs. The results demonstrate that the vector-based methodology used by Hbench:JGC is a promising technique for predicting application performance. In addition, we believe that when equipped with a better profiler and analyzer, the prediction accuracy of HBench:JGC can be improved further.

SPEC Application	Stdev (%)	Time Difference (%)
_201_compress	0.5	0.2
_202_jess	0.4	-2.2
_209_db	0.8	8.3
_213_javac	0.5	-15.8(-6.4*)
_227_mtrt	9.5	3.1
_228_jack	0.5	-0.2

* Results if we discard 3 explicit GCs.

Table 5. Summary of Predicted vs. Actual GC Times

5. Discussion and Future Work

In this section we discuss issues that might arise when using HBench:JGC on more sophisticated GC implementations such as those presented in Section 2.1.2, and how we plan to address these issues.

Concurrent garbage collection presents some technical challenges. With concurrent garbage collection, the application can continue to allocate new objects and access objects on the heap while a garbage collection is in process. Measuring the GC time is difficult because the GC time is dispersed in application execution time. We plan to approach this problem in the following way. We run a standard Java application without garbage collection, and then we run the same application with an additional thread that continuously allocates objects and invokes garbage collection. The performance degradation observed when the application is run with the additional GC intensive thread should be a good approximation of the GC time.

Many concurrent collectors are also incremental. Therefore, we will need to estimate the percentage of the heap that is scanned by the collector. In most cases, an incremental collector sets an upper bound on the number of root objects to be processed, from which one can estimate the number of objects on the heap to be scanned.

Predicting the performance of parallel garbage collectors can be potentially difficult because the speed-up of a parallel GC run over its sequential counterpart depends not only on the degree of parallelism, but also on how balanced each thread's load is and the interactions between the threads such as lock contention. Analyzing performance of multi-threaded applications in general is still an active area of research.

To apply HBench:JGC to generational garbage collectors, we model the collector performance for each generation, and then combine them together to form the total GC time. To achieve that, our profiler needs to be enhanced with the capability to estimate the object life expectancy. Furthermore, our analyzer should be able to predict when objects are promoted to older generations, i.e., it needs to know the age threshold for promotion. Some GC implementations make this knowledge public. For implementations that do not, we need to design our microbenchmark suite such that it can deduce the age threshold by creating and deleting objects at different rates.

Currently, the memory cache effect is included in our cost functions as a function of object size. Our results indicate that in some cases, this simple model might be insufficient. We are investigating ways to model the memory cache hierarchy explicitly.

Our short-term goal is to experiment on more garbage collector implementations and include more applications in our experiments. In the long run, we expect to refine our model to cope with more sophisticated GC implementations and incorporate HBench:JGC into the HBench:Java suite, in order to more accurately predict a Java application's total running time.

6. Related Work

Many researchers have studied the performance of dynamic memory management [5][15]. This literature provides a good foundation for understanding the inherent cost of dynamic storage allocation. Our approach differs in the goals we try to achieve. We emphasize predictability — the ability to predict application performance on different GC implementations without running the application on target implementations. In contrast, past research has focused on comparing the cost of memory management by running a set of popular applications on target memory management implementations.

Knuth [8] presents a comprehensive analysis and comparison of the time complexity of several dynamic storage management algorithms. This systematic approach to benchmarking memory management algorithms offers insight into the efficiency of these algorithms and helps explain the performance differences. However, the analysis assumes certain statistical properties for both memory allocation and liberation patterns and only applies when the system reaches equilibrium.

In [4], Cohen et al. compare performance of four compacting algorithms using analytical models. The analytical models are parameterized by the amount of work to be done, such as the number of cells (objects), number of pointers (links) and related information, and the time to perform the basic operations common to all compactors, such as the time to test a conditional expression. Their goal is similar to ours in that they also try to estimate GC execution times "without resorting to empirical tests". The main difference lies in the level of abstraction used for the primitive (elementary) operations. Their primitive operations are low-level machine instructions, whereas we conglomerate all machine instructions performed on an object into a single per-object operation (e.g., per-live-object overhead). Because their primitives are at such a low-level, their models are more elaborate and require intimate knowledge of the algorithms (i.e., the complete source code). Furthermore, as computer architectures become more advanced, machine-level optimizations and the memory cache hierarchy could introduce significant side effects such that the analytical model will no longer be applicable. In our case, the cost of primitives is measured explicitly by the microbenchmark and therefore includes these side effects.

7. Conclusion

HBench:JGC is a vector-based, application-specific benchmarking framework for evaluating garbage collector performance. Our results demonstrate HBench:JGC's unique predictive power. By taking the nature of target applications into account and offering fine-grained performance characterizations of garbage collectors, HBench:JGC can provide meaningful metrics that help better understand and compare GC performance.

8. Acknowledgement

Many of the ideas presented in this paper were inspired by discussions with Dave Detlefs from Sun Microsystems Labs. We would also like to thank the anonymous COOTS reviewers and our shepherd Chris Small for their valuable comments and suggestions in improving the paper.

9. Bibliography

[1] Boehm, H., and Weiser, M., "Garbage Collection in an Uncooperative Environment." *Software–Practice and Experience*, pages 807-820, September 1988.

[2] Baker, H. G. Jr., “List Processing in Real Time on a Serial Computer.” *Communications of the ACM*, 21(4), pages 280-294, April 1978.

[3] Cheney, C. J., “A Non-Recursive List Compacting Algorithm.” *Communications of the ACM*, 13(11), pages 677-678, November 1970.

[4] Cohen, J., and Nicolau, A., “Comparison of Compacting Algorithms for Garbage Collection.” *ACM Transactions on Programming Languages and Systems*, 5(4), pages 532-553, 1983.

[5] Detlefs, D., Dosser, A., and Zorn, B., “Memory Allocation Costs in Large C and C++ Programs.” *Software–Practice and Experience*, 24(6), pages 527-542, June 1994.

[6] JVMPI, Java Virtual Machine Profiling Interface. http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/index.html.

[7] Jones, R., and Lins, R. D., *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Son Ltd, New York, 1996.

[8] Knuth, D. E., *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Second Edition, Addison Wesley, Reading, MA, 1973.

[9] Larose, M., and Feeley, M., “A Compacting Incremental Collector and its Performance in a Production Quality Compiler.” In *Proceedings of the 1998 International Symposium on Memory Management*, pages 1-9, Vancouver Canada, October 17-19, 1998.

[10] Lieberman, H., and Hewitt, C., “A Real-Time Garbage Collector Based on the Lifetimes of Objects.” *Communications of the ACM*, 26(6), pages 419-429, June 1983.

[11] Seltzer, M., Krinsky, D., Smith, K., and Zhang X., “The Case for Application-Specific Benchmarking.” In *Proceedings of the 1999 Workshop on Hot Topics in Operating Systems (HotOS VII)*, pages 102-107, Rio Rico, AZ, March 29-30, 1999.

[12] Smith, F., and Morrisett, G., “Comparing Mostly-Copying and Mark-Sweep Conservative Collection.” In *Proceedings of the 1998 International Symposium on Memory Management*, pages 68-78, Vancouver Canada, October 17-19, 1998.

[13] SPECJVM98 Benchmarks. August 19, 1998 release. http://www.spec.org/osg/jvm98.

[14] Zhang, X., and Seltzer, M., “Hbench:Java – An Application-Specific Benchmarking Framework for Evaluating Java Virtual Machines.” In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 62-70, San Francisco, CA, June 3-4, 2000.

[15] Zorn, B., “The Measured Cost of Conservative Garbage Collection.” *Software–Practice and Experience*, 23(7), pages 733-756, July 1993.