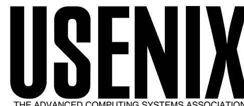
USENIX Association

Proceedings of the Java™ Virtual Machine Research and Technology Symposium (JVM '01)

Monterey, California, USA
April 23–24, 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.

Hot-Swapping between a Mark&Sweep and a Mark&Compact Garbage Collector in a Generational Environment

Tony Printezis

Department of Computing Science, University of Glasgow,
17 Lilybank Gardens, Glasgow G12 8RZ, Scotland.

tony@dcs.gla.ac.uk

Abstract

This paper describes a novel method for dynamically switching between a Mark&Compact (M&C) and a Mark&Sweep (M&S) garbage collector in the generational memory system of a high performance Java virtual machine. A M&C collector reclaims space by sliding all live objects towards the beginning of the heap. A M&S collector de-allocates garbage objects in-place. In this paper, both algorithms are assumed to operate over the old generation of a generational memory system and on their own provide different trade-offs to the application that uses them: faster old collections but with slower young collections and the possibility of fragmentation (M&S) or slower old collections but with faster young collections and the guarantee to eliminate fragmentation (M&C). We propose *Hot-Swapping*, a technique for dynamically switching between these two algorithms, to attempt to achieve the "best of both worlds". Its introduction to the memory system of the virtual machine imposed minimal changes to the existing implementations of M&C and M&S and virtually no extra performance overhead. Experimental results, presented in the paper, show that this hybrid scheme can either outperform both algorithms, or is very close to the faster of the two (whether this is M&S or M&C), while never being the slowest.

1 Introduction

Programming languages that rely on garbage collection [14] for their memory management have existed since the late 1950s. Even though the advantages of garbage collection (ease-of-development, program robustness, etc.) are well-known and accepted, most software developers have continued to rely on the traditional explicit memory management (`malloc`/`free` in C, `new`/`delete` in C++, etc.), largely because of performance concerns. Recently, however, the wide acceptance of the Java™ programming language [12] has encouraged developers to take advantage of the benefits of garbage collection and has stirred further interest in the area.

Generational garbage collection techniques [17, 25] can address the performance concerns usually associated with automatic memory management. They divide the heap into spaces, referred to as *generations*, according to object age. Since, for most programs, young objects are more likely to be garbage than older ones, concentrating collection activity on the young space increases throughput, as more free space is reclaimed per collection cycle. The young space is kept small to allow fast non-disruptive collection times.

Objects that survive a given number of young collections are considered long-lived and are *promoted* into an older generation. Even though older generations are large, they are not of infinite size, therefore they will eventually run out of space and require collection. Two well-known and widely-used algorithms that can perform the collection are Mark&Sweep and Mark&Compact [14, 27]. The former reclaims space by de-allocating it in-place and keeping track of it in order to re-use it later. The latter reclaims space by sliding (compacting) all live objects towards the beginning of the space, thus creating a single contiguous free chunk.

The two garbage collection algorithms described above have different performance characteristics and each one will perform best for specific loads. Mark&Compact provides fast allocation to old space, hence providing faster young collection times, and eliminates *fragmentation* [13]. Mark&Sweep provides faster old collection times, but can suffer from fragmentation and slower young collection times.

This paper proposes a technique to dynamically switch (*hot-swap*) between Mark&Compact and Mark&Sweep in order to achieve the "best of both worlds". This technique is simple to implement and imposes virtually no performance overhead on the memory system. Experimental results, presented in the paper, show that it can

either outperform both garbage collection algorithms on their own, or be very close to the faster one (whichever that is), while never being the slowest.

The idea of dynamically switching between different garbage collection algorithms has been successfully adopted before [7, 21, 16]. However, the novelty of the technique proposed here lies in the fact that it uses two collection algorithms that provide, in turn, faster old collections/slower young collections and slower old collections/faster young collections and tries to find a winning balance between them. As far as the author is aware, this aspect of the work has not been explored before.

It must also be noted that this paper does not address incremental garbage collection. Instead, it presents a technique that can improve the performance of batch jobs (compilation, raytracing, etc.), whose main requirement is throughput and not non-disruptive garbage collection pauses.

A longer version of this paper, which contains more graphs generated from the experimental results, is also available as a technical report [18].

1.1 The Implementation Platform

The Sun Microsystems Laboratories Virtual Machine for Research, henceforth *ResearchVM*, which was used as the implementation platform for all experiments presented in this paper, is a high performance Java virtual machine developed by Sun Microsystems. This virtual machine has been previously known as the "Exact VM", and has been incorporated into products; for example, the Java 2 SDK (1.2.1_05) Production Release, for the Solaris™ operating system. It employs an optimising just-in-time compiler [10] and a fast object-synchronisation mechanism [3].

More relevantly, it features high-performance *exact* (i.e., non-*conservative* [8], also called *precise*) memory management [2]. The memory system is separated from the rest of the virtual machine by a well-defined *GC Interface* [26]. This interface allows different garbage collectors to be "plugged in" without requiring changes to the rest of the system. A variety of collectors implementing this interface have been built. Above the GC Interface, a second layer called the *generational framework* facilitates the implementation of generational garbage collectors.

1.2 Terminology

Some terminology and abbreviations that will be used throughout the paper are enumerated below.

- *Mark&Sweep* (M&S): a stop-the-world GC that reclaims free space by de-allocating garbage objects in-place [14].
- *Mark&Compact* (M&C): a stop-the-world GC that reclaims space by sliding (compacting) live objects towards the beginning of the heap, providing a single contiguous area of free space [14].
- *Hot-Swapping* (H-S): the mechanism to dynamically switch between a M&S and a M&C GC, described in this paper.
- *Young GC*: a young generation GC in a generational memory system [25, 14].
- *Old GC*: an old generation GC in a generational memory system [25, 14].

1.3 Paper Overview

Section 2 provides the motivation behind this work by comparing and analysing the M&C and M&S algorithms and presenting the different trade-offs each of them provides. Section 3 outlines the Hot-Swapping mechanism, which we propose in order to achieve the best balance between the two algorithms mentioned previously. Section 4 presents and analyses timing results from the experiments. Section 5 covers the related work and section 6 concludes the paper.

2 Motivation

During our work on a different project (the *Generational Mostly-Concurrent GC*, discussed elsewhere [19, 20]), we needed to implement an in-place de-allocation mechanism to be applied to the old generation of the memory system of ResearchVM. However, we decided to first implement, test, and optimise this mechanism in the context of a non-concurrent stop-the-world M&S GC, for simplicity and predictability reasons.

When we compared the performance of the M&S GC and that of the default M&C GC (both of them were applied to the old generation of the system and shared the same semispace-based young generation), we made the following observations.

- M&S could decrease the maximum old GC pause times by a factor of 2 or 3, compared with M&C.
- Unfortunately, and somehow surprisingly, M&S also imposed a performance penalty of up to a factor of 2 on *young* GC pause times (⇝§2.1 and §2.2).
- M&S sometimes required a larger heap, compared with M&C, in order to operate. This was mainly due to fragmentation [28, 13].

Given the above, it is clear that an application which performs a lot of frequent heavy-duty old GCs could benefit from using M&S, whereas one that mainly performs young GCs could benefit from using M&C, which also has the advantage of eliminating fragmentation. The motivation for the work presented in this paper follows naturally: whether it is possible to dynamically switch between a M&S and a M&C old GC and achieve beneficial performance results by getting the "best of both worlds".

It should be noted that the applications we have observed do not seem to radically change their behaviour during their execution. The hybrid scheme described here is not proposed solely as a solution to detect and deal with application behavioural changes, but also to benefit applications that have uniform behaviour.

The remainder of this section will compare the M&C and M&S algorithms and will demonstrate how they affect the performance of our generational memory system. Section 2.1 covers object-allocation-related issues. Section 2.2 overviews the young GC mechanism and section 2.3 covers old GC issues. Then, section 2.4 presents measurements to illustrate the performance differences between the two algorithms.

2.1 Allocation Issues

As described in section 1.2, the main difference between the M&S and M&C GCs is their allocation and deallocation policies.

During GC, M&S de-allocates garbage objects in-place. It keeps track of where free space is located within the heap and uses this information to satisfy future allocation requests.

The particular technique that we use to keep track of free chunks in our implementation is *free lists, segregated by size*. We maintain separate free lists for free chunk sizes up to 128 4-byte words, segregated free lists for sizes up to 32K words, and a single large-free-chunk list for all larger sizes. During each old GC cycle, the free lists are re-created from scratch, rather than updated. Additionally, all adjacent free chunks are coalesced into single bigger ones.

It is worth pointing out that, according to Johnstone and Wilson, the use of segregated free lists is one of the worst allocation policies for generating fragmentation [13]. Our decision to choose it had originally been based solely on the grounds of simplicity. However, for the purposes of the hybrid mechanism described and analysed in this paper, fragmentation generated by M&S is not a major issue, as M&C will eventually eliminate it. Hence, we did not think that it was necessary to explore alternative allocation policies.

Considering M&C, this GC slides all live objects towards the beginning of the space, thus generating a single contiguous free chunk. Allocation off this free chunk can be very fast, using the well-known "bump a pointer and check" technique. This is clearly faster than any alternative technique (in particular, allocation from free-lists) and gives M&C an allocation performance advantage over M&S.

We consider the two algorithms when applied to the old generation of a generational memory system. In such a system, most allocations to the old generation take place in bursts during young GCs, as objects are evacuated from the young generation[1]. This observation provides an opportunity to optimise the allocation operation of M&S in the following manner: if a very large free chunk can be discovered, then linear allocation can be performed inside it, eliminating the need for look-ups on the free lists. This optimisation is very effective early in an application's execution and allows M&S to allocate objects almost as efficiently as M&C. However, an appropriately large free chunk is not always available, especially after a few old GC cycles when the space starts to become fragmented, and M&S typically has to eventually revert to allocating directly from the free lists.

It is worth pointing out that, no matter how the old generation is managed, the front-line allocator that all applications will use directly is a fast "bump a pointer and check", provided by the semispace-based young generation (see footnote 1 for an exception to this rule). So, an old generation with a slower allocator (e.g. M&S) will not affect the performance of most allocations.

To summarise, the speed of allocation to the old generation directly affects the performance of young GCs. As M&C has an inherently faster allocation operation, it also has the potential to support faster young GCs.

2.2 Young Generation GC Operation

Figure 2 illustrates the operation of young GCs over M&C. Objects from the young generation are promoted to the old generation using a Cheney-style copying technique [9]. Objects reachable from the old generation are promoted first. Forwarding pointers are installed in their old image in order to specify that an object has already been copied, to avoid copying it again if it is referenced multiple times. Then, the reference fields of newly-promoted objects are visited and, if they point to young objects, those objects are promoted too. This process is repeated until a given threshold of young objects has been promoted (this is adjusted dynamically).

The allocation operation of M&C ($\leadsto$§2.1) guarantees

[1] The main exception to this rule is very large objects that do not fit in the young generation and have to be allocated directly in the old one. This, however, is relatively infrequent.

Figure 1: Legend for figures 2, 3, 7, and 9.

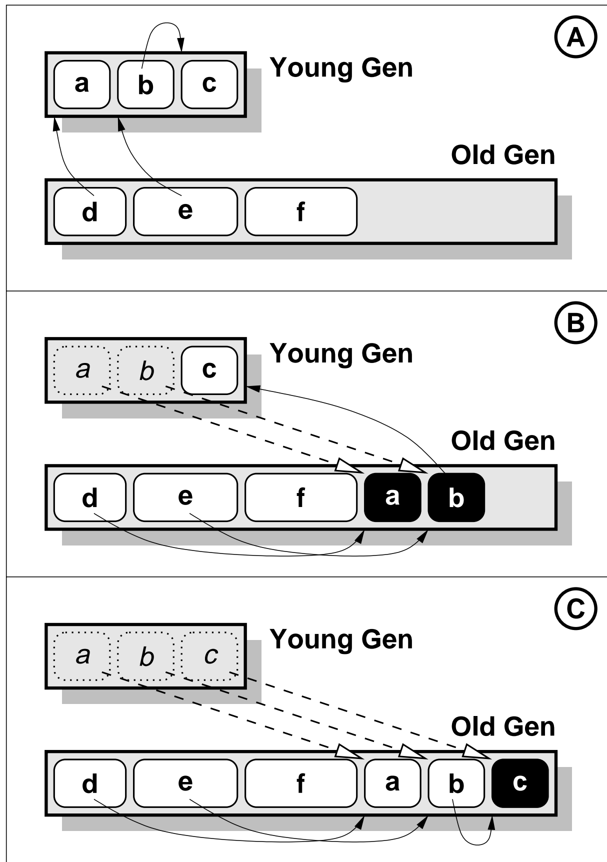


Figure 2: Young GC Operation for M&C.

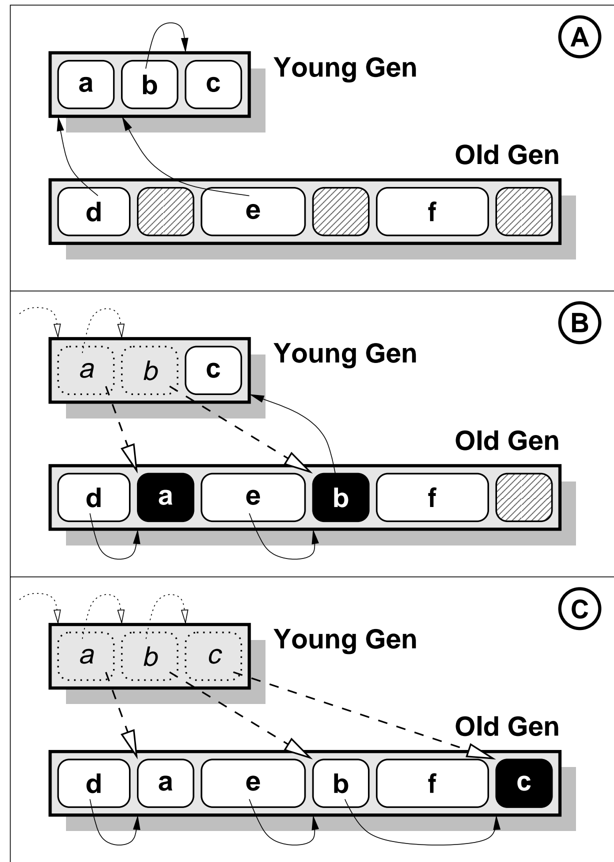


Figure 3: Young GC Operation for M&S.

that all newly-promoted objects will be allocated linearly into the old generation. This allows the above algorithm to discover them and iterate over them (in order to potentially promote more objects) very efficiently.

Figure 3 illustrates the operation of young GCs over M&S. This is largely similar to the one corresponding to M&C, with one important difference: M&S does not guarantee that all newly-promoted objects will be allocated linearly into the old generation. This makes scanning them less efficient, compared with M&C.

The way we achieve this is to chain the old images of newly-promoted objects into a linked list and iterate over it when we need to iterate over promoted objects. The forwarding references from the old images of the objects to the newly-allocated ones need to be installed anyway (for the reasons described above), hence the only extra expense is the construction of the list. In practice however it turns out that this is not as efficient as the equivalent operation of M&C.

To summarise, M&C not only provides faster allocation to the old generation (⇝§2.1), it also has the potential to allow faster object-promotion operations (e.g. iteration over newly-promoted objects) during young GCs. These two advantages that M&C has over M&S are, we believe, the main causes of the faster young GC times that can be achieved using M&C (timing results that demonstrate this

are included in sections 2.4 and 4).

2.3 Old Generation GC Issues

Even though M&C outperforms M&S in young GC times (⇝§2.1 and §2.2), it performs worse in old GC times, even up to 2–3 times slower than M&S. The main reason for this is that the compacting phase of M&C (in our implementation, a variation of the *Lisp 2 Algorithm*, described in Jones' book [14]) needs to

❏ relocate objects, possibly causing excessive memory copying, and

❏ patch reference fields to reflect the object relocations.

These two operations turn out to be more expensive than M&S's relatively simple operation of adding free chunks to free lists. However, they can affect performance only if a non-trivial percentage of garbage objects resides in the heap. If there are no garbage objects, no live objects need to be relocated and no reference fields need patching. In practice, if the percentage of garbage objects is very low, old GC times for M&S and M&C are very close. However, as this percentage increases, M&S starts outperforming M&C.

Experimental results in section 4 reflect the claims made in this section on old GC times.

2.4 Measurements

This section contains a concrete example to demonstrate the claims on young and old GC times that have been presented so far. Figure 4 shows the behaviour of young GC times for the large Javac benchmark (⇝§4.1), when using M&C. Each point represents a single young GC, the x-axis represents the start of the young GC (in seconds into the benchmark), and the y-axis represents the duration of the young GC (in milliseconds). Additionally, the two old GCs that took place are indicated with vertical arrows.

The figure shows that the behaviour of the application clearly affected the young GC times. It has in fact three distinct phases: the first one up to 21 seconds into the application (slower and more varied than the rest), the second one between 21 and 50 secs (more consistent and slightly faster than the rest, but with a few outliers), and the third one after 50 seconds (the end of the first and second phase is indicated in the figure).

Figure 5 shows the behaviour of young GC times for the same benchmark, when using M&S. Its data points, again representing single young GCs, are split into the following four categories.

① **100% Linear**: all the allocations to the old generation took place linearly to a single contiguous free chunk.

② **90%–100% Linear**: between 90% and 100% of allocations to the old generation took place linearly, the rest were directly off the free lists. For the ones that took place 100% linearly, no single contiguous free chunk that was large enough could be found, hence at least two non-contiguous ones were used.

③ **0%–90% Linear**: between 0% and 90% of allocations to the old generation took place linearly.

④ **0% Linear**: no allocations to the old generation took place linearly and all of them were directly off the free lists.

Looking at figure 5, and comparing it with figure 4, the following observations can be made.

❏ It is clear that young GC times using M&S are generally slower compared with the young GC times over M&C. This is most prominent during the first phase of the benchmark (0–21 seconds).

❏ The majority of young GCs from category ① took place before the first old GC (as the heap had not been fully used and a single contiguous free space was still available). Subsequently, after free space had been made available by the old GCs, contiguous large free chunks had not been produced, hence most young GCs did not fall into category ①. Interestingly, after each old GC, allocations still took place mostly-linearly but to smaller and more than one contiguous chunk (category ②). Then, when these were exhausted, all allocations took place off free lists (category ④). We believe that the above is clear evidence of fragmentation being introduced in the heap[2].

❏ Figure 5, when compared with figure 4, shows some evidence that the category ② and ④ young GCs are slightly slower than the ones of category ① (their data points seem to be higher).

Further, figure 6 shows the minimum, average, and maximum times for young and old GCs for the same benchmark. For the reasons described above, M&S does provide slower young GC times, compared with M&C. However, as seen on the same figure, it also provides much faster old GC times[3].

[2] Johnstone and Wilson claim that fragmentation only exists in a heap when the free space is fragmented *and* the free chunks are too small to satisfy individual allocation requests [13]. In this paper, we will use a slightly different definition of fragmentation: *"the heap is fragmented and no large enough free chunks are available to support linear allocations during young GCs, even if free chunks are available to satisfy non-linear allocations."*

[3] The figure is slightly misleading because, as it has already been mentioned, only two old GCs took place. Still, this result follows the general pattern we have observed with old GC times.

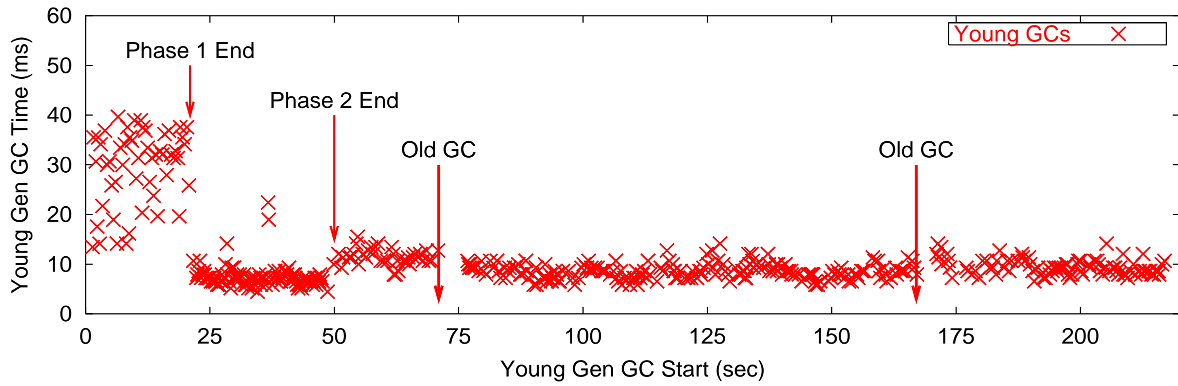


Figure 4: **Javac** benchmark with M&C — young GC trace.

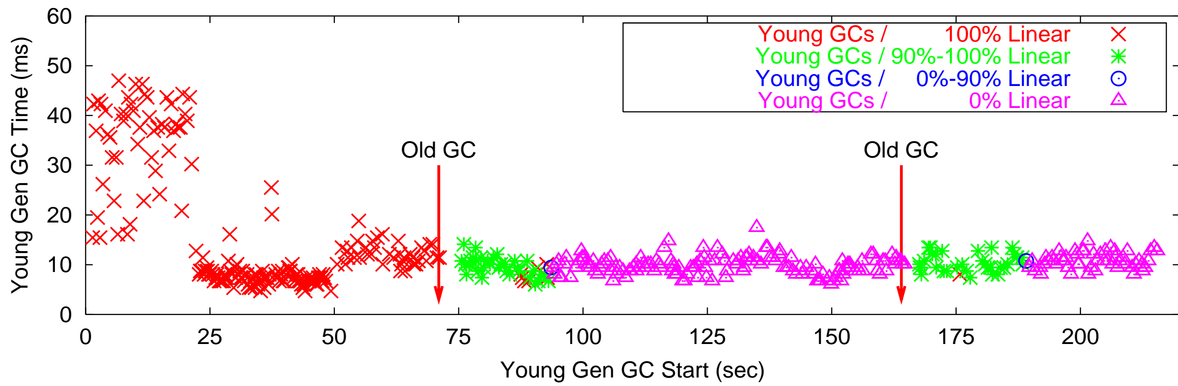


Figure 5: **Javac** benchmark with M&S — young GC trace.

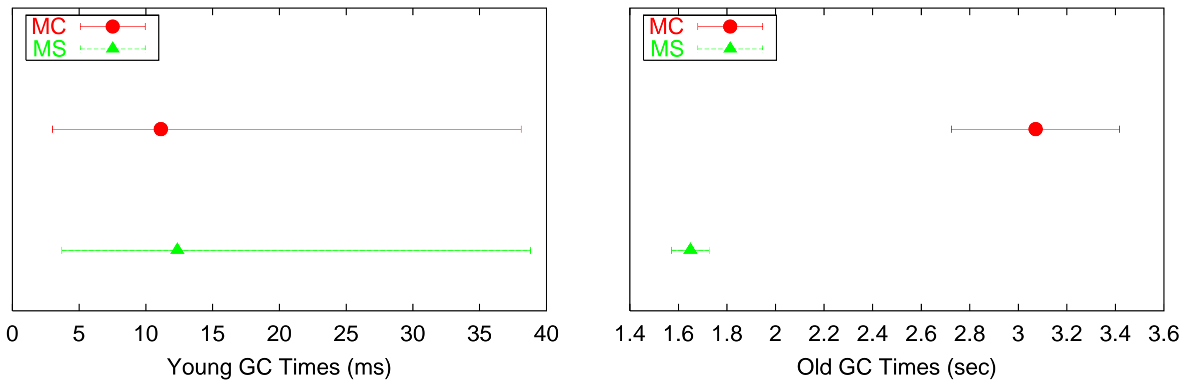


Figure 6: **Javac** benchmark, GC time distributions (min, avg, max).

From the above comparison, it would have been natural to conclude that M&S should perform better overall as, after all, its old GCs are seconds faster than those of M&C and its young GCs only a few milliseconds slower. Still, the fact that around 2,270 young GCs took place, compared with only 2 old GCs, amplified the slowness of young GCs and did not make the faster old GCs as beneficial as originally thought. It turns out that the total GC time for M&S was only 0.03 seconds faster than that of M&C.

We have observed this pattern (elapsed times being very similar for both algorithms) in a variety of applications. However, if an application performs mainly young GCs and very few old GCs, it will benefit from M&C. Alternatively, if it relies on old GCs, it will benefit from M&S. We have observed applications that fall into these two categories and measurements for some of them are presented in section 4.

3 The Hot-Swapping Mechanism

Section 2 demonstrated the differences in performance and trade-offs between M&S and M&C, when applied to the old generation of a generational memory system. This section presents a mechanism for dynamically switching between M&S and M&C, in an attempt to achieve the "best of both worlds". This mechanism will be referred to as *Hot-Swapping* (H-S). Section 3.1 presents the requirements that we imposed on H-S. Sections 3.2 and 3.3 describe how the the switch between M&C and M&S can be performed efficiently. Then, section 3.4 outlines the heuristic used to determine when to perform the switch and section 3.5 concludes.

3.1 Requirements

The requirements that we imposed on the H-S mechanism were the following.

① The switch between M&C and M&S should happen efficiently, preferably in constant time.

② Apart from the mechanisms to switch between the two algorithms and to determine whether to perform the switch, no further performance penalty should be imposed.

③ Flexibility should be provided on when the switch between M&C and M&S is allowed to be performed, so that different heuristics can be easily adopted and evaluated.

④ Only minimal changes should be imposed on the implementation of the existing M&C and M&S algorithms.

3.2 Switching from M&C to M&S

The operation to switch from M&C to M&S is straightforward. M&C ensures that all the free space resides at the end of the heap. When switching to M&S, it is only necessary to add a single free chunk that spans all the free space (this can be done in constant time). Having done this, M&S can operate over the heap as it would normally have done. Figure 7 illustrates this operation with a concrete example. First, a free chunk is added at the end of the heap. Then, M&S operates as it would normally do, finding **a**, **c**, and **f** to be garbage and replacing them with free chunks.

3.3 Switching from M&S to M&C

Switching from M&S to M&C is slightly more complicated. The reason for this is the presence of the free chunks scattered around the heap, which M&C should not be aware of. One way to deal with this, is to iterate over all free chunks and transform them to unreferenced Java objects (scalar arrays), which M&C will consider garbage and reclaim. However, there might be a large number of free chunks in the heap and these transformations might prove to be a performance overhead.

To avoid having to apply the format changes during the switch, we decided to change the format of free chunks and "mask" them as garbage objects. Figure 8 illustrates the format of scalar arrays in ResearchVM [26]. The object header of an array contains three fields: *(i)* a pointer to its NearClass[4], *(ii)* a flags field that is mainly used by the synchronisation mechanism [3], and *(iii)* a field containing the length of the array.

Figure 10 illustrates the updated format of free chunks. The first field of the free chunk header points to a NearClass constructed for this purpose (essentially a copy of the scalar array NearClass, but with a different address). The second field is used for linking free chunks in the free lists (⇝§2.1). The third field contains the length of the free chunk. This format allows M&S to determine whether an object is a free chunk or not (by comparing its NearClass address to the well-known fake NearClass), determine its length, and have access to a field that can be used for linking purposes. Additionally, it allows M&C to consider free chunks as garbage scalar arrays (note that the GC does not use the flags field during its operation,

[4]A *NearClass* is a small data structure that contains GC-related information about instances of that class, mainly on object layout, type of each field, type of entries (in the case of arrays), etc. All objects point to their corresponding NearClasses and they, in turn, point to the full class structure [26]. The NearClass structure was introduce in an attempt to allow more efficient access to the information that the GC will need during its operation by keeping it grouped together, hopefully causing less secondary cache misses.

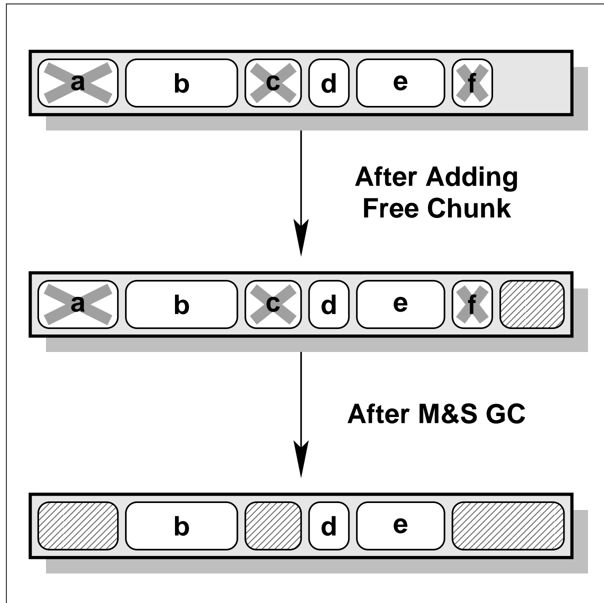


Figure 7: Switching from M&C to M&S.

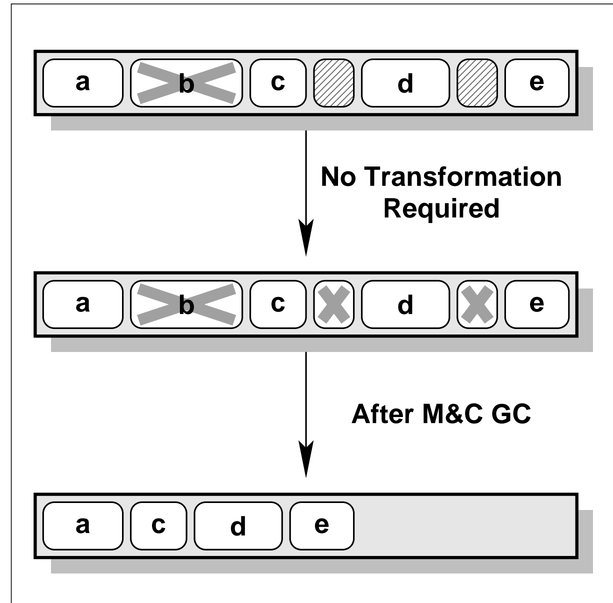


Figure 9: Switching from M&S to M&C.

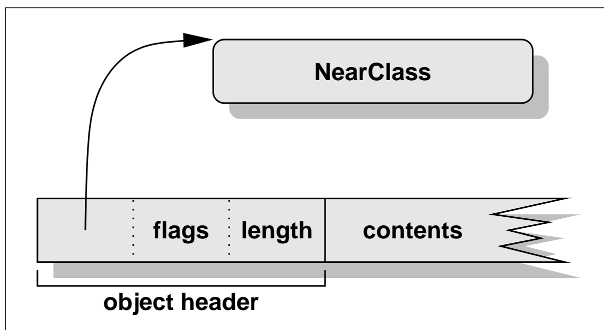


Figure 8: Scalar array format in ResearchVM.

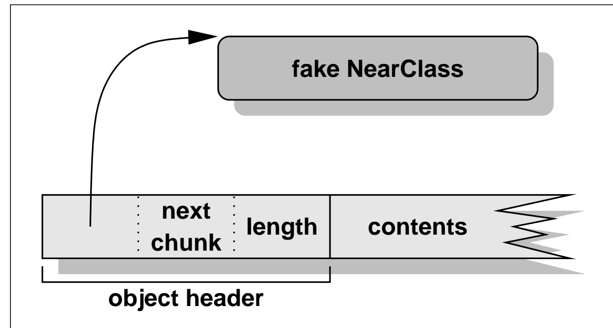


Figure 10: Free chunk "masked" as scalar array.

therefore M&C will not try to access the second field of the object header).

Given the above format of free chunks, switching from M&S to M&C is very efficient as no changes to the heap are necessary: all free chunks will be considered as garbage objects by M&C and their space will be reclaimed. Figure 9 illustrates this operation. M&C operates directly over the heap and compacts live objects **a**, **c**, **d**, and **e**, reclaiming the space taken up by the two free chunks and garbage object **b**, providing a single contiguous area of free space.

It is worth mentioning here that an alternative way to perform the switch from M&S to M&C would have been to modify M&C to be aware of free chunks. However, we chose the approach outlined above purely on grounds of simplicity, as it was less complicated to modify the format of free chunks and leave M&C totally unchanged.

3.4 Heuristic

Sections 3.2 and 3.3 demonstrated how the switch between M&C and M&S is accomplished. However, an interesting question is when to hot-swap between the two GCs. The heuristic that we implemented is the following.

> *"Use M&C after expanding the heap to take advantage of its fast allocation, otherwise use M&S to take advantage of its fast old GC times, unless linear allocation fails sufficiently, in which case switch back to M&C once to eliminate fragmentation and move back to M&S."*

A more detailed explanation follows.

- At the beginning, and also immediately after a heap expansion, a large free area is available at the end of the heap, therefore we use the operation of M&C to

satisfy allocation requests to the old generation, as it is the fastest.

- ❏ When an old GC is initiated, we use M&S (after switching from M&C if necessary), as it provides faster old GC times. Subsequently, we use the M&S allocation mechanism to satisfy allocation requests to the old generation.

- ❏ If less than 60% of young GCs since the last old GC did not use full linear allocation (i.e. less than 60% of young GCs were not in category ①, ⇝§2.4), ensure that the next old GC will switch from M&S to M&C, because the heap is assumed to be fragmented and M&C will eliminate this.

The above heuristic, even though it is simplistic, attempts to take advantage of the best qualities of each GC. It also efficient to implement, as only a flag needs to be set when linear allocation fails and two counts need to be updated once per young GC.

3.5 Summary

This section has presented a mechanism to efficiently hot-swap between a M&S and a M&C GC. It satisfies the requirements enumerated in section 3.1 because the switch between the two algorithms can be performed in constant time (requirement ①), no further restrictions (apart from the ones already in place, e.g. all the threads must be stopped [1]) are imposed on when the switch can take place (requirement ③), and the only necessary change was the alteration of the free chunk format, with the M&C code being left completely unaltered (requirement ④). Additionally, the GC Interface of ResearchVM uses virtual calls to invoke most operations implemented by each GC [26]. This allows us to, say, change the allocation operation from that of M&C to that of M&S by simply replacing the appropriate virtual call and imposing no further performance penalty. This satisfies requirement ②.

4 Measurements

This section presents the timing results from the experiments that were performed to compare the performance of H-S to that of M&C and M&S on their own. Section 4.1 gives a brief description of the six benchmarks used and section 4.2 presents and analyses the timing results.

All experiments were run on a lightly-loaded Sun Ultra™ 80 workstation [22] with four 450MHz UltraSPARC-II CPUs and 2GB of main memory running the Solaris 7 operating system. All timing results reported were obtained using the `gethrtime` Solaris call and are the average of ten runs, after the worst time has been removed.

4.1 The Benchmarks

The six benchmarks we used for our evaluation were the following.

- ❏ **GCBench**[5]: A benchmark written to stress the allocation and promotion operations of the Java system. It creates a large long-lived tree and then spawns two threads, which create smaller and shorter-lived trees. This process is repeated five times.

- ❏ **MarkTest**: A benchmark written to evaluate optimisations applied to the marking phase of old GCs. It allocates a very large object array and assigns repeatedly new objects to its entries in an attempt to generate garbage. This process is repeated ten times.

- ❏ **GCOld**: A benchmark written to evaluate the performance of incremental GCs. It creates disjoint tree structures and performs operations on them, creating short- and long-lived objects, while rendering some others garbage. It is described in more detail elsewhere [19, 20].

- ❏ **JOS**: It de-serialises a tree data structure from a file using the standard Java Object Serialization™ facilties [23], performs some updates to it, and re-serialises it to a second file.

- ❏ **DNA**: An application that creates a suffix tree [24], populates it with a DNA sequence, and performs searches over it. The process is repeated for 5 different parts of a DNA sequence (that of merged genomes of two kinds of yeast: saccharomyces cervisæ and saccharomyces pombæ).

- ❏ **Javac**: A large Javac job that compiles all the standard classes of the Java 1.2.2 distribution. It reads 2,740 `.java` files, containing 776,488 lines of code, and generates 4,638 `.class` files.

While the first four benchmarks are clearly synthetic, the last two are actually "real" applications.

4.2 Results

Figure 11 plots the total GC times (i.e. the amount of time the application was stopped for either a young or an old

[5] This benchmark was originally written by John Ellis, Pete Kovac, and Hans Boehm. We have altered it so that it has a longer elapsed time and uses more memory. Its original version is available from `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench/`.

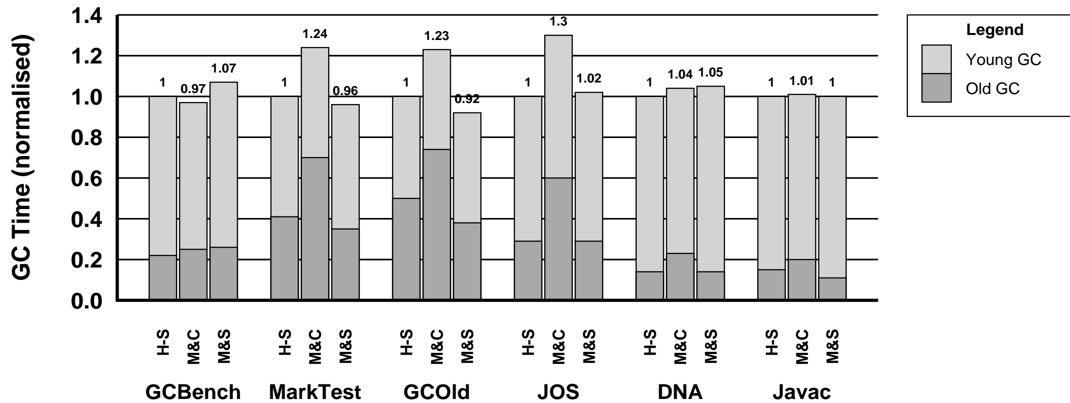


Figure 11: Young/Old GC times, normalised with respect to H-S (less is better).

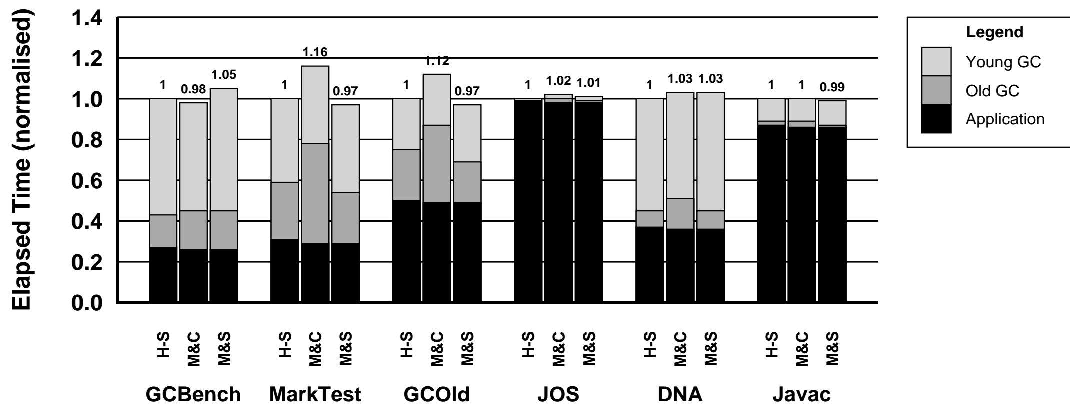


Figure 12: Elapsed times, normalised with respect to H-S (less is better).

GC) for all benchmarks, normalised with respect to H-S. The results show that, out of the three GC algorithms, no one performed best in all cases. However, H-S does appear to be the best choice as it performs best in three cases (**JOS**, **DNA**, and **Javac**) or is closer to the faster one, whether that is M&C (**GCBench**) or M&S (**MarkTest** and **GCOld**). It is interesting to note that, in the case of the **JOS** and **DNA** benchmarks, H-S outperforms the other two, even though M&C provides overall lower average young GC times and M&S lower average old GC times.

Figure 12 plots the application elapsed times, normalised with respect to H-S. This figure follows the pattern of figure 11, however the difference between the three GC algorithms is not as obvious; this is especially the case for the **JOS** and **Javac** benchmarks for which the GC times were a small fraction of the application elapsed times (interestingly, these are the two benchmarks that performed a lot of I/O).

It is important to note that, even though both H-S and M&C could run the **JOS** benchmark with a 90MB old generation, M&S failed to do so due to fragmentation-related reasons (an allocation for a large 8MB array was failing, even though there was at least twice as much space free in the generation in total). Additionally, when executing the **GCBench** benchmark, M&S performed one more old GC compared with H-S and M&C (see table 1), again due to fragmentation-related reasons. These two points are evidence to support that H-S can operate with the same heap sizes to those of M&C, even if they cause fragmentation problems to M&S.

Finally, it must be pointed out that, for a given benchmark, the difference in total GC times between two of the GC algorithms matches closely, as expected, the difference in application elapsed times. The only exception to this rule is the **Javac** benchmark. The reason behind

this is the excessive disk accesses that this benchmark performs (reading `.java` files and writing `.class` files) that caused the timing results to be slightly unpredictable (even though the total GC times obtained from that benchmark seemed to be consistent).

All the timing results obtained from the six benchmarks, when run with H-S, M&C, and M&S, are included in appendix C (tables 1–6 — entries surrounded by ||s indicate the best result out of the three algorithms).

5 Related Work

For a good introduction to garbage collection, the reader is referred to two excellent publications which touch on most of the techniques mentioned in this paper: Jones' book [14] and Wilson's survey [27]. On a related topic, Wilson, Johnstone, and others survey different dynamic allocators and discuss the problem of memory fragmentation [28, 13].

Generational garbage collection techniques were originally proposed by Lieberman and Hewitt [17], but Ungar reported the first implementation [25]. Appel has provided further analysis of their benefits [4, 5, 11].

Switching garbage collectors dynamically is not a new idea and has already been explored in the past. There are three notable efforts in this area.

Sansom proposed *dual-mode garbage collection* that switches between a single-space compacting [15] and a two-space copying [9] garbage collector. His motivation was to achieve maximum performance but improve on the high space requirements that the copying collector requires.

Bartlett's *mostly-compacting garbage collector* [6] is targetted for a system with ambiguous garbage collection roots. It allows live objects to be relocated, as long as they are not ambiguously referenced. To improve performance, a generational framework was added later [7]. The motivation behind this work was purely to take advantage of the benefits of compaction in an environment with ambiguous roots.

Finally, Lang and Dupont proposed an *incremental incrementally-compacting garbage collector* [16]. According to this algorithm, most free space is de-allocated in-place during a collection phase, with a small region of the heap being compacted using a two-space copying collector. This is performed by piggy-backing a Cheney-style copying operation [9] on the marking phase of the collector. The region to be evacuated is chosen by splitting the heap into fixed size regions and cycling through them. This scheme has similar motivation to ours (i.e. uses both in-place de-allocation and compaction, the latter to decrease fragmentation). However, it does require extra memory to operate (one region must always be free in order to evacuate live objects to) and it is not clear whether always compacting a small region of the heap during each collection is more beneficial than compacting the whole heap when necessary (provided total pause time is not an issue).

6 Conclusions and Future Work

This paper compares the performance and trade-offs of two stop-the-world garbage collectors, Mark&Compact and Mark&Sweep, when applied to the old space of a generational memory system of a high-performance Java virtual machine. It then proposes Hot-Swapping, a technique to dynamically switch between these two collectors in order to benefit from the advantages of both algorithms. Performance results, included in the paper, show that the H-S technique can either outperform M&C and M&S, or be slightly slower than the faster of the two, while never being the slowest.

We are interested in further improving the Hot-Swapping idea. We would like to determine what effect on the performance of the H-S mechanism, if any, have some of its parameters (young generation size, object-promotion order, etc.). Further, the heuristic on when to hot-swap (⇝§3.4) is currently very simplistic and we would like to implement and evaluate alternatives.

Finally, the ability of our hybrid scheme to eliminate fragmentation and improve young GC times seems very attractive for our *mostly-concurrent generational GC* [19], which currently relies only on free-list-based allocation and is sometimes affected by these two problems. Hence, we would like to explore the possibility of incorporating a form of hot-swapping in such a concurrent environment.

A Acknowledgements

This work was supported by Sun Microsystems with funding through Sun's External Research Program, that also kindly donated the workstation used for the development and evaluation of the ideas presented in this paper. The author is grateful to the members of the Java Technology Research Group at SunLabs East, especially Steve Heller, Dave Detlefs, and Alex Garthwaite, for their support, contributions, ideas, and all the food over the last few years! The author would also like to thank Huw Evans, Rolf Neugebauer, Andy King, Ole Agesen, Richard Jones, and Malcolm Atkinson for their life-saving, last-minute, constructive feedback and Ela Hunt for providing the **DNA** application.

B Trademarks

Sun, Sun Microsystems, Java, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

References

[1] O. Agesen. GC Points in a Threaded Environment. Technical Report TR-98-70, Sun Microsystems Laboratories, Palo Alto, CA, December 1998.

[2] O. Agesen and D. Detlefs. Finding References in Java™ Stacks. In *Proceedings of the OOPSLA'97 Workshop on Garbage Collection and Memory Management*, Atlanta, GA, USA, October 1997.

[3] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An Efficient Meta-lock for Implementing Ubiquitous Synchronization. In *Proceedings of OOPSLA'99*, pages 207–222, Denver, Colorado, USA, November 1999.

[4] A. W. Appel. Garbage Collection can be Faster than Stack Allocation. *Information Processing Letters*, 25(4):275–279, January 1987.

[5] A. W. Appel. Simple Generational Garbage Collection and Stack Allocation. *Software — Practice and Experience*, 19(2):171–183, March 1988.

[6] J. F. Bartlett. Compacting Garbage Collection with Ambiguous Roots. Technical Report 88/2, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA, February 1988.

[7] J. F. Bartlett. Mostly-Copying Garbage Collection Picks up Generations and C++. Technical Report TN-12, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA, October 1989.

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

[9] C. J. Cheney. A Non-Recursive List Compacting Algorithm. *Communications of the ACM*, 11(13):677–678, November 1970.

[10] D. Detlefs and O. Agesen. Inlining of Virtual Methods. In *Proceedings of ECOOP'99*, pages 258–278, Lisbon, Portugal, June 1999.

[11] M. J. R. Goncalves and A. W. Appel. Cache Performance of Fast-Allocating Programs. Technical Report CS-TR-482-94, Princeton University, December 1994.

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

[13] M. S. Johnstone and P. R. Wilson. The Memory Fragmentation Problem: Solved? In *Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, Vancouver, Canada, October 1998. ACM Press.

[14] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Ltd, 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[15] H. B. M. Jonkers. A Fast Garbage Compaction Algorithm. *Information Processing Letters*, 9(1):26–30, July 1979.

[16] B. Lang and F. Dupont. Incremental Incrementally Compacting Garbage Collection. *ACM SIGPLAN Notices*, 22(7):253–263, July 1987.

[17] H. Lieberman and C. E. Hewitt. A Real-Time Garbage Collector based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429, 1983.

[18] T. Printezis. Hot-Swapping between a Mark&Sweep and a Mark&Compact Garbage Collector in a Generational Environment. Technical Report TR-2001-78, Department of Computing Science, University of Glasgow, Scotland, March 2001.

[19] T. Printezis and D. Detlefs. A Generational Mostly-Concurrent Garbage Collector. In *Proceedings of the 2000 International Symposium on Memory Management*, pages 143–154, Minneapolis, MN, USA, October 2000. ACM Press.

[20] T. Printezis and D. Detlefs. A Generational Mostly-Concurrent Garbage Collector. Technical Report TR-2000-88, Sun Microsystems Laboratories, June 2000.

[21] P. Sansom. Combining Single-Space and Two-Space Compacting Garbage Collectors. In *Functional Programming, Glasgow 1991: Proceedings of the 1991 Workshop, Portree, UK*, pages 312–323. Springer-Verlag, 1992.

[22] Sun Microsystems Inc. Ultra™ 80 Workstation Profile.
http://www.sun.com/desktop/products/Ultra80/
[November 5, 2000].

[23] Sun Microsystems Inc. *Java™ Object Serialization Specification — JDK™ 1.2*, November 1998. Revision 1.43.

[24] E. Ukkonen. On-Line Construction of Suffix-Trees. *Algorithmica*, 14(3):249–260, 1995.

[25] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.

[26] D. White and A. Garthwaite. The GC Interface in the EVM. Technical Report TR-98-67, Sun Microsystems Laboratories, 1999.

[27] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the First International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St Malo, France, September 1992. Springer-Verlag.

[28] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of the Second International Workshop on Memory Management*, number 986 in Lecture Notes in Computer Science, Kinross, Scotland, September 1995. Springer-Verlag.

C Timing Results

GCBench	H-S	M&C	M&S
Elapsed Time (sec)	42.92	42.18	45.08
Total GC Time (sec)	31.74	30.94	33.96
Young Gen Size (MB)	2	2	2
Old Gen Size (MB)	32	32	32
Young GC Avg (ms)	32.63	30.27	34.11
Young GC Max (ms)	198.1	79.7	197
Young GC Total (sec)	24.70	22.92	25.82
Old GC Avg (ms)	292.57	333.11	324.94
Old GC Max (ms)	499.3	965.1	587.9
Old GC Total (sec)	7.02	7.99	8.12
M&Cs Performed	4	24	0
M&Ss Performed	20	0	25

Table 1: **GCBench** results.

MarkTest	H-S	M&C	M&S
Elapsed Time (sec)	15.10	17.60	14.58
Total GC Time (sec)	10.70	13.17	10.21
Young Gen Size (MB)	2	2	2
Old Gen Size (MB)	32	32	32
Young GC Avg (ms)	56.00	50.69	57.41
Young GC Max (ms)	103.81	69.54	101.45
Young GC Total (sec)	6.32	5.72	6.48
Old GC Avg (ms)	397.37	676.27	338.61
Old GC Max (ms)	645.54	702.90	345.18
Old GC Total (sec)	4.37	7.43	3.72
M&Cs Performed	2	11	0
M&Ss Performed	9	0	11

Table 2: **MarkTest** results.

GCOld	H-S	M&C	M&S
Elapsed Time (sec)	75.58	84.65	72.54
Total GC Time (sec)	38.78	47.84	35.77
Young Gen Size (MB)	2	2	2
Old Gen Size (MB)	350	350	350
Young GC Avg (ms)	15.09	14.75	16.22
Young GC Max (ms)	64.1	41.9	63.4
Young GC Total (sec)	19.5	19.05	20.95
Old GC Avg (ms)	6423	9588	4937.7
Old GC Max (ms)	9511.9	10727.7	5162.2
Old GC Total (sec)	19.26	28.76	14.81
M&Cs Performed	1	3	0
M&Ss Performed	2	0	3

Table 3: **GCOld** results.

JOS	H-S	M&C	M&S
Elapsed Time (sec)	147.17	149.03	147.70
Total GC Time (sec)	4.00	5.21	4.08
Young Gen Size (MB)	2	2	2
Old Gen Size (MB)	110	110	110
Young GC Avg (ms)	24.40	24.25	25.10
Young GC Max (ms)	59.9	59	63.8
Young GC Total (sec)	2.83	2.81	2.91
Old GC Avg (ms)	1169.8	2400.3	1167.4
Old GC Max (ms)	1169.8	2400.3	1167.4
Old GC Total (sec)	1.17	2.39	1.16
M&Cs Performed	0	1	0
M&Ss Performed	1	0	1

Table 4: **JOS** results.

DNA	H-S	M&C	M&S
Elapsed Time (sec)	203.37	208.32	209.45
Total GC Time (sec)	130.59	135.88	136.5
Young Gen Size (MB)	2	2	2
Old Gen Size (MB)	400	400	400
Young GC Avg (ms)	226.39	214.40	238.29
Young GC Max (ms)	333	308.4	342.4
Young GC Total (sec)	112.29	106.34	118.19
Old GC Avg (ms)	6097.5	9842.57	6099.77
Old GC Max (ms)	6448.2	9929.9	6442.8
Old GC Total (sec)	18.29	29.52	18.3
M&Cs Performed	0	3	0
M&Ss Performed	3	0	3

Table 5: **DNA** results.

Javac	H-S	M&C	M&S
Elapsed Time (sec)	224.63	223.77	224.20
Total GC Time (sec)	31.38	31.41	31.38
Young Gen Size (MB)	2	2	2
Old Gen Size (MB)	135	135	135
Young GC Avg (ms)	11.79	11.12	12.35
Young GC Max (ms)	38.3	38.1	38.8
Young GC Total (sec)	26.79	25.24	28.05
Old GC Avg (ms)	2282.4	3071.6	1649.25
Old GC Max (ms)	2824.8	3416.6	1726.5
Old GC Total (sec)	4.56	6.14	3.29
M&Cs Performed	1	2	0
M&Ss Performed	1	0	2

Table 6: **Javac** results.