Mark and Split [*]

Konstantinos Sagonas [1,2] Jesper Wilhelmsson [1]

[1] Department of Information Technology, Uppsala University, Sweden
[2] School of Electrical and Computer Engineering, National Technical University of Athens, Greece
{kostis,jesperw}@it.uu.se

Abstract

The *mark-sweep* garbage collection algorithm constructs a list of memory areas to allocate into (the *free list*) during its sweep phase. This phase needs time proportional to the size of the heap which is collected. We introduce *mark-split,* a non-moving garbage collection algorithm that constructs the free list *during* the mark phase by maintaining and splitting free intervals. With mark-split, the sweep phase of mark-sweep becomes unnecessary and the cost of collection is proportional to the size of the live data set. Our performance evaluation, using a high performance Java implementation running standard benchmarks, shows that mark-split can significantly reduce collection times compared with mark-sweep and requires little extra space to do so. The overhead to the cost of marking is moderate and often pays off for itself by avoiding the sweep phase. Since there is no guarantee that this is always the case, we also propose adaptive schemes that try to combine the best performance characteristics of mark-split and mark-sweep collection.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—memory management (garbage collection)

General Terms Algorithms, Performance, Languages

Keywords Non-moving Garbage Collectors, Mark-and-Sweep, Mark-and-Split, Java

1. Introduction

Many modern programming languages employ tracing collectors to reclaim unused portions of dynamically allocated memory. Copying [6] and mark-sweep [10] are the two most commonly used kinds of such collectors. The choice between them involves tradeoffs, which by now are well understood both theoretically and experimentally. A copying collector needs to reserve some memory to copy the traced objects (the *live* data set) to, but it provides cheap allocation, avoids fragmentation, and its running time is proportional to the amount of live data, not the size of the heap which is collected. On the other hand, a mark-sweep collector can complete its mark phase in time proportional to the amount of live data, but needs time proportional to the size of the heap for its sweep phase. Since it is not uncommon for the amount of live data to be only a small fraction of the heap which is collected, it has often been argued that copying collection is superior to mark-sweep when judged from a purely performance perspective.

In practice, considerations other than performance may dictate the choice between copying vs. mark-sweep. Unlike mark-sweep, a copying collector moves objects. As such, it is not suitable in environments (for example, in C or C++ with a standard compiler) in which pointers cannot be identified with certainty. Also, copying is often not a viable option for memory-demanding applications since, for safety, half of the memory has to be reserved for the needs of the collector rather than be available for the allocation needs of the application. In some sense, this inherent drawback of copying is very unfortunate since it is precisely in the case where the heap size is large that the performance of a mark-sweep collector suffers most from the complexity of its sweep phase.

In this paper, we develop an algorithm that has the advantages of both techniques. More specifically, we propose *mark-split,* a new garbage collection algorithm that does not move objects, requires little extra space in practice, and has complexity proportional to the size of the live data set, not the size of the heap which is collected. The mark-split collector achieves these properties by building the free list *during* marking, thereby rendering the subsequent sweep phase completely unnecessary. The price to pay is increased marking cost. However, when the size of the live data set is small compared to the size of the heap which is collected, or when the live data forms big contiguous areas, the additional overhead is moderate and the use of the mark-split technique typically pays off by avoiding the cost of the sweep phase. Adaptive schemes, combining mark-sweep and mark-split, are of course also possible and are briefly discussed.

We have implemented the mark-split garbage collection algorithm in the BEA JRockit system, a commercial Java implementation. Our performance evaluation using the SPECjvm98 programs as benchmarks shows that mark-split can achieve significantly better garbage collection performance than the mark-sweep collector, is able to reduce individual garbage collection pauses, and requires little additional space. In fact, as the size of the allocated heap increases, the performance advantage of mark-split over mark-sweep manifests itself more and more clearly.

Overview We begin by reviewing the mark-sweep algorithm and related work in this context. Then, we present the mark-split algorithm in detail (Section 3). We discuss implementation aspects, variants and optimizations in Section 4. An analysis of the time and space complexity of various garbage collection algorithms, including mark-split, is given in Section 5, followed by a brief description of possible adaptive schemes (Section 6). The paper ends with a detailed performance evaluation (Section 7) and concluding remarks.

[*] Research supported in part by grant #621-2003-3442 from the Swedish Research Council (Vetenskapsrådet).

ISMM'06 June 10–11, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-221-6/06/0006...$5.00.

```
proc $mark\_sweep\_gc()$ $\equiv$
  foreach $root \in rootset$ do $mark(*root)$
  $sweep()$

proc $mark(object)$ $\equiv$
  if $marked(object) = \mathbf{false} \rightarrow$
      $marked(object) := \mathbf{true}$
      foreach $pointer$ in $object$ do
        $mark(*pointer)$
```

Figure 1. The mark-sweep algorithm and its mark phase in detail.

2. Mark-Sweep Collection

2.1 The basic algorithm

In its traditional form, mark-sweep collection is done in two separate phases. The first phase, called the *mark phase*, starts from a set of *roots* and marks all reachable objects using an algorithm similar to the one shown in Figure 1. This phase takes time proportional to the size of live data in the heap. The second phase, called the *sweep phase*, reclaims memory occupied by unreachable objects. Reclamation is traditionally done by visiting each object in the heap, examining its marked status, and reclaiming the memory occupied by the object if the object is not marked. Marked objects are unmarked in the process, in preparation for the next collection. Memory for reclaimed objects is placed in a *free list*, which is subsequently used by the allocator. Sometimes many free lists are built, one per object size, which then makes allocation a very fast operation; essentially of constant cost. The sweep phase of the algorithm takes time proportional to the size of the entire heap, since every object in the heap must be visited.

When the memory occupied by live objects is a significant portion of the size of the heap, the cost of the sweep phase is asymptotically similar to that of the mark phase. In practice, marking time dominates in this case, often significantly, since modern hardware is optimized for the sequential memory scan that the sweep phase performs. However, when the size of the live data set is only a small fraction of the heap size, sweep time dominates. In big heaps, the time difference can also be big.

2.2 Variants and optimizations of mark-sweep

The algorithm we show in Figure 1 is recursive, but this is only for simplicity of presentation. In practice, marking is implemented using a non-recursive procedure where the stack of pointers known to be live (the *mark stack*) is managed explicitly. Sometimes, a technique known as *pointer reversal* or one of its variants is used. Some other memory managers, for example the one described by Boehm [5], use separate memory areas for storing objects with and without pointers and employ mark-sweep only for memory areas containing objects with pointers. In the same paper, it is also noticed that, due to cache effects, a significant fraction (roughly one third) of the mark time is spent executing the load instruction(s) that retrieve memory of objects that have been placed on the mark stack. To improve cache performance, Boehm suggests a *prefetch on grey* optimization [5], i.e. prefetching objects when these are placed by the collector on the mark stack.

All these techniques affect the mark phase of mark-sweep. They are orthogonal to the topic of this paper and do not change the complexity of the mark-sweep algorithm. Moreover, they are immediately applicable to mark-split collection too. But there also exist techniques that affect the sweep phase of mark-sweep.

Mark-sweep using a separate bitmap This technique is folklore but is described nicely in a paper by Dimpsey et al. [8]. In languages where pointers only point to the beginning of each object, one bit per object suffices for marking. Then, if the mark bits are stored in a separate bitmap table, the size of this table can be inversely proportional to the size of the smallest object and can often be held in main memory. More importantly, the sweep phase often examines only the bitmap (i.e. not the heap) and can examine it many bits at a time. Although this technique is often quite effective, it lowers only the constant factor of the sweep phase cost, not its asymptotic complexity.

Parallel sweep and mark during sweep The sweep phase can be easily parallelized by horizontally partitioning the memory and running multiple sweep phases, one per partition, in parallel. Also in a parallel setting, Quinnec et al. [11] have suggested a technique to reserve some memory for allocation and perform the sweep phase of one collection in parallel with the mark phase of the next collection. Again, although effective in their setting, these techniques do not affect the complexity of the sweep phase.

Treadmill Baker proposed the *treadmill* [2], a collector based on mark-sweep where the sweep phase only has to update a few pointers to doubly linked lists of objects. The sweep phase of the treadmill has constant time complexity. However, the technique heavily relies on the fact that all objects have the same size. Attempts to extend the treadmill to size-segregated storage have resulted in algorithms of complexity linear in the size of the heap and have not enjoyed much success in practice.

Selective sweeping Chung et al. developed a technique, called *selective sweeping* [7], which performs well when the heap is nearly empty. The technique works as follows. During the mark phase, the addresses of the live data are recorded in an auxiliary memory area outside the heap. Mark bits are used to prevent multiple insertions and this area has size proportional (and comparable) to the size of the live data set. Its contents are then sorted and guide the subsequent sweep phase to reclaim all free areas in time which is proportional to the size of the live data set.

Lazy sweeping Back in 1982, Hughes suggested that the sweep phase can be deferred until the allocator requests more memory and performed lazily (i.e., in chunks) at that time [9]. The initial motivation for doing so was to reduce garbage collection pause times and thereby obtain a semi-incremental version of mark-sweep. Since then many other researchers, including Zorn [12] and Boehm [5], have suggested variants of lazy sweeping in order to improve paging and/or cache performance of the collector. The lazy sweeping technique improves performance compared with mark-sweep by exploiting the fact that the allocator will find the free space in memory and in today's machines most probably in the cache.

To make the case for lazy sweeping stronger, Boehm has argued that the comparison of asymptotic complexity between collection algorithms should also take allocation into account [4]. Under this prism, all garbage collection algorithms have complexity proportional to the size of the heap. With lazy sweeping in particular, the cost of the sweep phase is reduced to effectively zero, since a cost similar to the cost of the sweep phase is paid by the allocator.[1]

In this paper we will stick to the more traditional view of garbage collection complexity; that in which the cost of allocation is not taken into account and the sweep phase dominates the complexity of mark-sweep collection. We will not rely on laziness, but will completely eliminate the sweep phase with the mark-split garbage collection algorithm described in the next section.

After presenting the algorithm, we will analyze the complexity of different garbage collection algorithms and discuss the advantages and disadvantages of mark-split compared with selective and lazy sweeping in detail.

[1] This complexity argument assumes that objects are initialized.

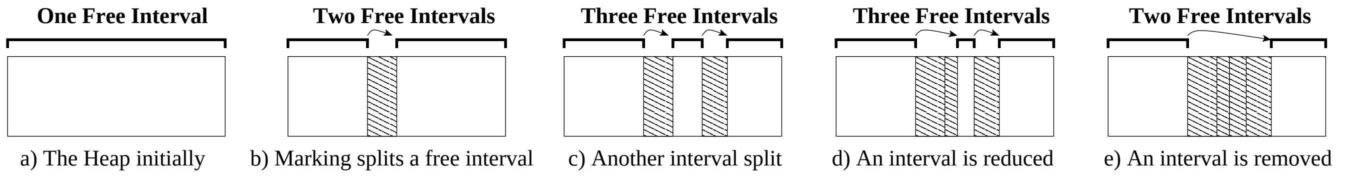


Figure 2. Illustration of the mark-split collection algorithm.

```
proc mark_split_gc() $\equiv$
  new_interval($heap\_start, heap\_end$)
  foreach $root \in rootset$ do mark($*root$)

proc mark($object$) $\equiv$
  if marked($object$) = false $\rightarrow$
      marked($object$) := true
      split(find_interval($\&object$), $object$)
      foreach $pointer$ in $object$ do
        mark($*pointer$)
```

Figure 3. The basic mark-split algorithm and its mark phase.

```
proc split($interval, object$) $\equiv$
  $objectEnd := \&object +$ size($object$)
  $keepLeft :=$ keep_interval($\&object - interval.start$)
  $keepRight :=$ keep_interval($interval.end - objectEnd$)
  if $keepLeft \wedge keepRight \rightarrow$
      new_interval($objectEnd, interval.end$)          // Case 1
      $interval.end := \&object$
  else if $keepLeft \rightarrow$
        $interval.end := \&object$                     // Case 2
  else if $keepRight \rightarrow$
        $interval.start := objectEnd$                  // Case 3
  else remove_interval($interval$)                     // Case 4

funct keep_interval($size$) $\equiv$
  return $size \geq T$                       // $T$ is a threshold
```

Figure 4. The procedure that splits an interval.

3. Mark-Split Collection

Let us initially assume that the heap to be collected is contiguous. This is by no means a requirement, but it simplifies presentation.

3.1 The idea

If we are to do away with the sweep phase, we must somehow build the list of free memory areas during marking. We do this by turning the view of a mark-sweep collector's work around.

> *Instead of first marking live objects and then sweeping the heap to find free areas, we optimistically assume that the entire heap will be free after collection and we then let the mark phase repair the free list by rescuing the memory of live objects.*

By *rescue memory for live objects* we mean "make sure that the memory interval where this object resides is taken off the free list".

Since we rescue heap *intervals* of live objects, it is natural to represent the free memory also as intervals. We will refer to the latter as the set of *free intervals*.

Figure 2 shows a mark-split collection in action. The collection starts with only one free interval that spans the entire heap; see Figure 2(a). Mark-split collection then proceeds with marking live objects, as in mark-sweep. During marking, rescuing a live object often splits a free interval into two; hence the name of the algorithm. This is shown in Figures 2(b) and 2(c). However, notice that it is also possible that a free interval is reduced in size (Figure 2(d)) or that an interval is removed from the set (Figure 2(e)).

When marking is finished, the intervals left are the ones that are actually free. The set of free intervals can directly be used for allocation at this point. No sweep is required.

3.2 The algorithm

Figures 3 and 4 show the actual algorithm. The notation $\&object$ has its familiar C meaning. We assume that there is a way to get the size of an object and denote this by size($object$).

The free intervals are stored in a global data structure, not shown in the code. Each interval is a record with two fields: $start$ and end. Given the address of an object, the auxiliary procedure *find_interval* returns the free interval containing that address.

Mark-split collection starts by calling the auxiliary procedure *new_interval*. This creates a new interval that spans the entire heap. We then proceed to the mark phase. Notice the similarities in the code of Figures 3 and 1. The only differences are that there is no sweep phase and that the *split* procedure is called for each unmarked object we find, in order to update the free intervals.

The *split* procedure (Figure 4) splits an interval into two, one at the left and one at the right of the live object. The left interval ranges from the start of the original interval to the start of the object. The right interval ranges from the end of the object to the end of the interval which is split. At this point we have to decide whether to keep these intervals or not and this is determined by the *keep_interval* Boolean function. We then have the following cases:

Case 1 Both intervals will be kept. This means we have to insert a new interval into the set. The original interval is updated to the values of the left interval and the new interval which is inserted is the right interval.

Case 2 The left interval is the only one kept. The end value of the original interval is updated with that of the left interval.

Case 3 The right interval is the only one kept. The $start$ value of the original interval is updated with that of the right interval.

Case 4 None of the two intervals is kept. The original interval is removed from the set.

In Figure 4, the condition for keeping an interval is that its size is greater or equal to some threshold T, which is set globally. One possible value for this threshold is 1, in which case all intervals are kept. A more reasonable alternative is to set T equal to the size of the smallest allocatable object. Yet another is to set T equal to the smallest size of memory that a mark-sweep collector keeps. We note that T can have a big impact on the cost of mark-split as it determines the number of elements in the set of free intervals. More on this below.

3.3 The free intervals data structure

The cost of mark-split collection depends highly on the choice of data structure used to store the set of free intervals.

The most frequent operations on this data structure are locating existing intervals and inserting new ones. These two operations, and locating intervals in particular, constitute the bulk of the cost for the mark-split algorithm. For decent performance, it is essential that the *find_interval* operation has sub-linear — if not constant — cost. Also, it is desirable that the allocator can use the free intervals data structure directly — or at least without requiring a costly translation — after collection is finished. Data structures with such properties are for example balanced search trees, splay trees, or skip lists.

Note that, unless we reserve space in the header of the objects, it is not possible to store the information about free intervals in the heap cells during mark-split collection. In a mark-sweep collector this is possible, but only because the sweep phase starts after the marking is finished, and at that time we know that the objects we overwrite with the free list contain garbage. In the mark-split algorithm this is not the case. During marking, we store intervals for memory areas that might eventually turn out to be live. Therefore, we can not overwrite the data in the heap and the set of free intervals must be stored in a separate memory area.

3.4 The good cases

The absolute best case for mark-split occurs when the heap contains no live data. More generally, mark-split performs well when the size of the live data set is relatively small compared to the size of the heap. However, notice that another very good case is when the live objects form a small number of big contiguous chunks as in Figure 5. If marking visits live objects in such a way that the size of the free intervals set remains small, the performance of mark-split is not significantly different than that of the mark phase of a mark-sweep collector. Since the sweep phase is avoided, mark-split's performance is better than mark-sweep's.

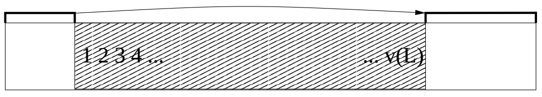


Figure 5. A very good case for mark-split: live data forms one big chunk and marking is consecutive (as shown by the numbers).

3.5 The worst case

Assume that the threshold T is chosen equal to the size of the smallest allocatable object. Then performance-wise, the worst case for mark-split occurs when the entire heap is filled with the objects of size T and marking creates lots of holes, even if these are created only temporarily. For each live object, we need to find the corresponding interval in the set and either split it, reduce the size of it, or remove it. If there are I free intervals and we use a binary search tree for representing them, locating an interval is a $\log I$ operation. Notice however that I will never grow larger than the number of objects that fit in half the heap. This is shown in Figure 6. The live objects which have been marked occupy precisely half the heap. Moreover, each of them is located exactly one object apart. The number of free intervals I reaches its maximum value at this point. If, at this point, another live object is found, the *remove_interval* procedure will be called and the set of free intervals will decrease in size.

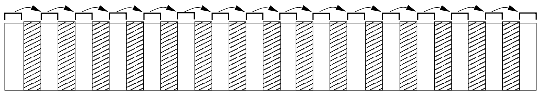


Figure 6. Worst case in space need for storing the free intervals.

4. Implementation Aspects of Mark-Split

We have implemented the mark-split algorithm in the runtime system of BEA JRockit (thereafter abbreviated as JRockit). Performance comparisons amongst commercial Java implementations, for example those shown on the SPEC benchmark web pages, show that JRockit is one of the systems with the best performance.

JRockit offers a selection of garbage collection strategies tailored to different types of applications and environments [3]. In its default setting, JRockit uses a two-generational collector. The young generation offers a choice between a single- and a multi-threaded copying collector. If desired, the young generation can be turned off which makes the collector non-generational and does not use copying. Since our intention is to directly compare mark-split with mark-sweep, we turned off the young generation. The old generation (and the non-generational collector) uses a mark-sweep collector which is available in parallel, concurrent, and so called deterministic (i.e. time-limited) flavors.

The mark-sweep collector has existed for quite some time in JRockit and its implementation has been quite fine-tuned. For instance, the runtime system stores the mark bits for objects on a bitmap table separated from the objects. This allows the sweep phase both to benefit from cache effects and to be faster by examining a whole word at a time and compare it directly with zero. In fact, even if a single bit is set, the entire word is considered live and the corresponding space for objects is simply discarded (i.e., not used for allocation). This means that free memory chunks of size up to 62 words may be discarded by the mark-sweep collector. Furthermore, to avoid the cost of the heap lock which is a major bottleneck in multi-threaded Java applications, *thread local heaps (TLHs)* [8] of size 2 KB are used for allocation. Consequently, JRockit does not store any areas smaller than 2 KB in the free list because this is the minimum free area size it has any use for. This in turn means that a single live bit in the middle of a 4 KB block of otherwise free memory can invalidate the entire block. In the mark-sweep collector, all this is done in order to save sweep time. The mark-split collector on the other hand can easily consider this space as free, which means that it can recover more space than the mark-sweep collector in most collections.

4.1 Our current implementation

For simplicity, we have presented the mark-split algorithm assuming that the heap to be collected is contiguous. If the heap consists of n different memory areas, the set of free intervals can be built to contain one interval per memory area (n intervals) at the start of the collection, instead of one interval for the entire heap. This means there will be a small setup cost proportional to the number of contiguous areas that are collected. In JRockit, the holes between heap areas are considered as live objects and marked as such in the mark phase to exclude the areas from the free list. This means that we did not have to worry about holes in the heap in our implementation.

The free intervals are stored in a general balanced search tree. We used the AA tree [1] in our implementation. Each tree node occupies five words: two for the interval's start and end address, two for the node's children, and one for the node's level (its height in the tree).[2] To avoid breaking any implicit assumptions of JRockit, we chose to use the same limit (2 KB) as the mark-sweep collector as threshold for keeping an interval in the tree. In the algorithm, this is done by setting T to 2 KB. The implementation of the routines that manage the interval tree (in C) was straightforward, but for making sure to implement them without recursion. Everywhere an object is marked as live we call the function that updates the free interval tree. Other than that, the marking code is completely unchanged.

[2] If we controlled the addresses, each node could occupy four rather than five words; for example, the level fits in a byte and could be squeezed in some other word. However, since the size of the tree is small, we saw little incentive to resort to such tricks in the current implementation.

We wanted to implement the mark-split algorithm in JRockit without changing any parts of the system outside of the actual collector. The free interval tree is therefore translated after the mark phase into the regular in-heap linked list that JRockit normally uses for allocation. This translation has negligible cost; see Section 7.1. In principle, it can be removed completely and allocation can directly use the free interval tree instead of the linked list. The performance of allocation will remain unaffected. Apart from the actual implementation of the algorithm (about 600 lines of C), the only change we made to JRockit code was to add a call to the *split* procedure in the places where objects are marked (three places), add some initialization in the start of a garbage collection, and comment out the code for the sweep phase.

4.2 Some possible optimizations

When the size of the free interval set remains small, mark-split performs very well. Therefore, mark-split can benefit from optimizations which manage to keep this size as small as possible during collection. In effect, such optimizations need to control the order of marking, which in turn means that most probably they have to rely on profiling to achieve good results. Some additional optimizations of mark-split's implementation are described below. The last one has been implemented but, since it is not tuned yet, we disabled it during the performance evaluation.

Storing the free intervals in-heap In languages where objects are reasonably large, as for example in Java, a few header words per object can be reserved for maintaining the free interval structure inside the heap. Doing so, is desirable for improving cache performance. The heap objects need to be scanned to find live pointers, and the free intervals data structure needs to be modified when a live object is found. Keeping the object and the free intervals in the same memory area will most likely decrease memory traffic. However, this is purely an optimization for speed. Considering the small memory overhead of storing the free intervals in our current implementation, adding header words to every object in the heap would most likely require more memory.

Storing mark bits in the objects In a mark-sweep collector, keeping the mark bits on a separate bitmap typically improves the performance of the sweep phase. But since mark-split removes the sweep phase, the argument for keeping mark bits on the side is not valid anymore. Keeping mark bits together with the objects may improve cache performance.

Maintaining a free interval cache To lower the cost of the *find_interval* operation, especially in the case when the same free interval is repeatedly trimmed in size, a *free interval cache* could be used. The cache could store references to the last n intervals that have been updated. When a new live object is found, we look to find the interval in the cache before traversing the set of free intervals. For objects such as lists, where a number of linked objects are allocated directly after each other on the heap, a cache like this can significantly improve performance.

4.3 Generational, concurrent, and parallel variants

A generational variant of the mark-split algorithm is straightforward. In fact, since most objects tend to die young, it makes sense to use mark-split in the young generation and benefit from the fact that mark-split's complexity is proportional to the size of the live data set, not that of the heap which is collected. On the other hand, the benefit may be offset by the fact that the nursery is typically small in size.

In the beginning of each mark-split collection, the entire heap is considered free. Any remainder of the free list that the mutator uses for allocation is forgotten (i.e., it is automatically included in the single free interval which spans the entire heap). This approach is normally used by stop-the-world collectors. Many concurrent and on-the-fly collectors however keep the old free list live to allow the mutator to allocate in it during the collection. The same technique can easily be used with the mark-split algorithm as described below.

When mark-split is used in a concurrent setting, the old free list is not forgotten and the mutator is allowed to allocate in it during the collection. In this case, the mark-split collector can still consider the whole heap as free in the beginning of the collection provided it ensures that objects allocated during the collection are not considered free at the end of the collection. This can be done in several different ways. For instance, all objects allocated during the collection could be pushed on the mark stack or the allocator could call the *split* procedure for all allocated objects to remove their memory from the free intervals. Both actions would effectively act as a write barrier for the allocator.

If the allocator calls *split* during a collection and this call results in a modification of a free interval, a lock on the interval would be required to avoid synchronization problems with the collector in a concurrent setting. If this technique incurs too much overhead, either on allocation or on accessing the free intervals data structure, one could instead remember all objects allocated during the collection and mark them as live as part of the final stage of the collector. In this case the *split* is performed by the collector and synchronization is only needed at the end to make sure that no new objects are allocated while the free intervals data structure is made up-to-date.

Several different ways of parallelizing mark-split are possible. In the most straightforward of them, all marker threads modify the same tree of free intervals and some form of (hierarchical tree) locking is used to prevent concurrent modifications made by *split*. Note that locking is *not* necessarily required for concurrent calls to *find_interval*. If the free interval data structure becomes a point of contention, another possible parallelization scheme is to let each marker thread maintain its own set of free intervals (all starting with an interval which represents the entire heap as free) and merge these trees when the collection is finished, thereby creating the final free list. Finally, if garbage collection is parallelized using a segmented heap, where each marker thread only collects objects residing in its own heap region, the mark-split collector can use one set of free intervals per marker thread. In this case, the final merging is straightforward. Evaluating the performance of different parallelization schemes of mark-split is future work.

5. Abstract Performance Comparison

We first analyze the time complexity and amount of auxiliary space that different garbage collection algorithms require. We then try to give some more insight and compare mark-split against variants of mark-sweep.

5.1 Complexity Analysis

Refer to Table 1. Its first row should be self-explanatory. Copying has $O(L)$ time complexity, where L is the size of live data. If we know this size in advance or allocate the *to-space* on demand, copying needs additional space of size L in the best and H in the worst case (when all of the heap is live).

Mark-sweep needs $O(L)$ time for its mark phase and $O(H)$ for its sweep phase.[3] It requires additional space for the bitmap. All objects, dead or alive, need a mark bit. We denote the size of the bitmap by M. If one bit per object suffices, then in the worst case M is bounded by $H/(wo)$. If one bit per word is needed, then $M = H/w$ as shown on the right of Table 1. Lazy sweeping has the same time and space complexity as mark-sweep collection.

[3] We slightly abuse notation and show the costs of each phase separately.

GC method	Time	Auxiliary space Best case	Auxiliary space Worst case
Copying	$O(L)$	L	H
Mark-sweep	$O(L) + O(H)$	M	M
Selective sweeping	$O(L) + O(L \log L) + O(L)$	$M + \nu(L)$	$M + \nu(H)$
Mark-split	$O(L \log I)$	$M + k$	$M + k\frac{H}{2o}$

H:	Size of heap to be collected
L:	Size of the live data set
I:	Number of free intervals
w:	Number of bits in a machine word
o:	Size of the smallest allocatable object
M:	Size of mark bit area (e.g., $M = \frac{H}{w}$)
k:	Size of each free interval tree node

Table 1. Time and space characteristics of different garbage collection algorithms.

Selective sweeping [7] has three separate phases: a mark phase with time complexity $O(L)$, a phase of complexity $O(L \log L)$ where the addresses of all live objects are sorted, and a selective sweep phase with time complexity $O(L)$.[4] Space-wise, it requires auxiliary space for the mark bits and space for the addresses of all live objects. We use the notation $\nu(L)$ to denote the number of objects in L, and similarly for H. The space requirements of selective sweeping are not negligible.

Characterizing the time complexity of mark-split is a bit involved. The problem is that mark-split depends on the number of free intervals, I, not only on L. I can be expressed in terms of L and H as follows:

$$I \text{ is bounded by } \begin{cases} \nu(L) + 1 & \text{if } L < \frac{H}{2} \\ \frac{H}{2o} & \text{if } L \geq \frac{H}{2} \end{cases}$$

In words: if less than half the heap is live ($L < H/2$), I is bounded by the number of live objects plus one; otherwise I is at most the number of objects that fit in half the heap. To see the latter, observe the worst case shown in Figure 6. When using a balanced search tree, each marking incurs an additional cost of at most $\log I$. Thus, the time complexity of mark-split is $O(L \log I)$.

The auxiliary space that mark-split requires is space for the mark bits and space for the free interval tree. If each free node has size k the space required for the latter is kI. In the best case, only one interval is ever created. In the worst case, I is equal to $H/(2o)$. Both values very unlikely to occur in practice.

5.2 Comparison between variants of mark-sweep

Mark-split vs. selective sweeping Compared to selective sweeping [7], which needs to record all heap addresses containing live objects, mark-split requires significantly less auxiliary space. In fact, there is even a complexity difference between the two algorithms. Consider the case shown in Figure 5, which is not so uncommon in practice. For example, it may correspond to a list whose elements are arranged in sequence. For this case, selective sweeping will need additional space of size $\nu(L)$, while mark-split only will need constant space, assuming that live cells are visited in a consecutive order. In fact, even in cases where marking does not happen consecutively, the space that mark-split needs to represent the free interval tree is typically very small — especially if the T threshold (the minimum size of intervals to keep) is chosen wisely. The performance numbers of Section 7 support this claim.

Selective sweeping chooses to pay a non-negligible space cost in order to save time during marking. In selective sweeping, the addresses of live objects are recorded in an array and, once marking is finished, this array is sorted. Effectively, this pass constructs the heap intervals which contain live data. Mark-split eagerly maintains these intervals.

To see the similarities and differences between mark-split and selective sweeping clearly, let us consider the dual version of mark-split, called *mark-coalesce*. The mark-coalesce algorithm views the heap as a set of *occupied* rather than free intervals. Collection starts with the empty set of occupied intervals. Marking makes intervals occupied and coalesces them with their neighbors as necessary. In this way, the mark-coalesce algorithm ends up doing the same amount of work as mark-split, with the only exception that at the end of the collection an extra pass (of complexity $O(L)$) is needed to construct the free list, which the allocator needs. By avoiding this pass, the mark-split algorithm is slightly faster than mark-coalesce. Also, we think that mark-split is conceptually more appealing than its dual. Notice however that mark-coalesce can be seen as a variant of selective sweeping that eagerly maintains live interval information in a compact representation, instead of being lazy and postponing this compaction till the end of the mark phase.

Mark-split vs. lazy sweeping As mentioned in Section 2, lazy sweeping has been suggested as a method to improve the cache performance of a mark-sweep collector and provide a "pay-as-you-go" mechanism for sweeping the heap which is collected. First, notice that lazy sweeping does not affect the complexity of mark-sweep; the cost of the sweep phase is paid in full no matter how it is split in installments. What lazy sweeping *does* affect is the cache performance of a mark-sweep collector. Indeed, it is plain to see that sweeping the entire heap at the end of each collection can be detrimental to the application's cache performance. Sweeping the heap in smaller chunks, when memory is requested by the allocator, is of course preferable.

Although we do not provide experimental evidence for the following claim, we want to argue that, at least in principle, mark-split has better cache performance than lazy sweeping. Mark-split constructs the free list without any sweeping. At the end of the collection, whenever a new memory chunk is requested by the allocator, this chunk is readily available *without* touching any other memory. In contrast, lazy sweeping will sweep and touch some non-free memory, thereby affecting cache performance of the mutator, most probably to the worse. The cache advantage of lazy sweeping compared with a plain mark-sweep collector is based on the fact that a cache line is typically used shortly after is it swept. Thus, lazy sweeping often avoids one of a total of possibly two cache misses (one during sweeping, one during use) of a mark-sweep collector. With mark-split, the first cache miss never happens.

A final note Despite these properties, the time complexity of mark-split is cause for concern. The problem is that both phases of mark-sweep— and its mark phase in particular — have a really small constant. So, when $(L \log I) > (L + H)$, it is unlikely that a mark-split collector will manage to beat mark-sweep, especially as L approaches H. In such cases, an adaptive garbage collection scheme might be called for. In the next section, we describe such schemes.

6. Adaptive Schemes

The basic idea of all adaptive schemes is simple: optimistically start with a mark-split collection and if it is detected that the cost will be too high, simply revert to a mark-sweep collection. First of

[4] In languages where pointers only point to the beginning of objects, the complexity of the last two phases of selective sweeping is $O(\Lambda \log \Lambda)$ and $O(\Lambda)$ respectively, where $\Lambda = \nu(L)$.

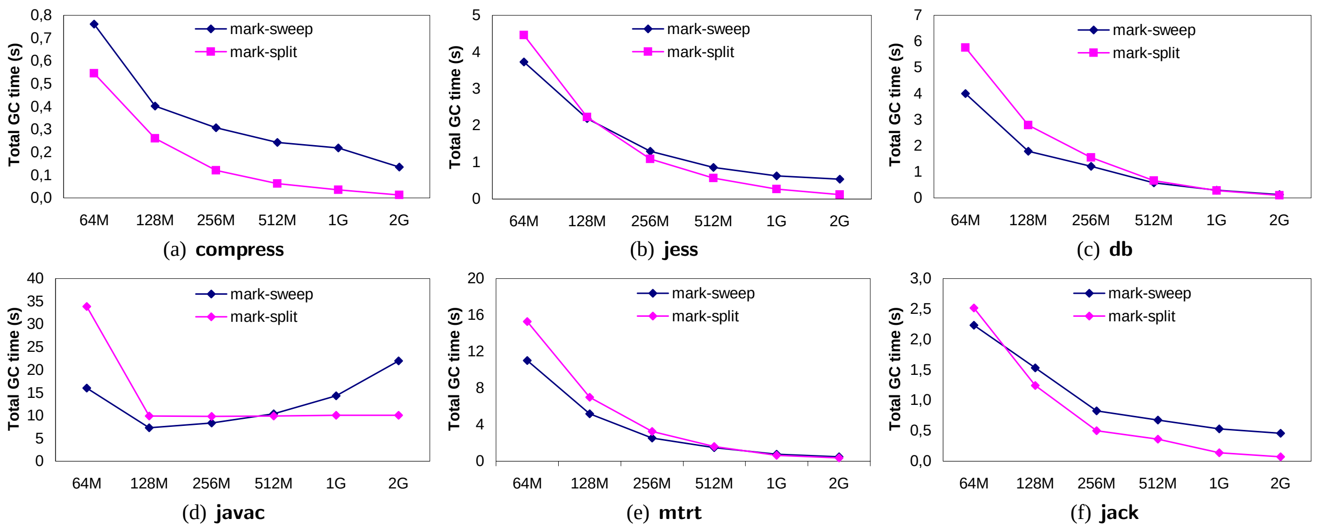


Figure 7. Total garbage collection times for mark-sweep and mark-split when varying the size of available heap from 64M to 2G.

all, notice that switching between the two algorithms on the fly is straightforward and comes at essentially no cost. This is because the mark-split collector performs all the work of the mark phase of a mark-sweep collector. If a switch takes place, we can continue with the mark phase until its completion and then run the sweep phase as if mark-split never occurred.

The main difference between alternative adaptive schemes is how to detect that the cost of mark-split collection will be high. The ways to do this are of course related to the additional costs of performing the mark phase of mark-split rather than the mark and the sweep phase of mark-sweep.

The major cost of the mark-split collector is due to locating free intervals. One easy way of limiting this cost is to put a limit on the maximum size of the data structure. For example, we can put a limit of 42 KB or of not more than 17% of the heap. Notice that besides serving as an adaptive scheme, this also provides a mechanism to put a limit on the memory overhead of a mark-split collector.

Another way of limiting the cost of searching though the data structure is to set some limit on the number of comparisons. This does not limit the size of the free interval tree, but rather how deep into the tree we are allowed go before giving up on mark-split. To avoid giving up prematurely because a single search in the data structure happens to go deeper than the allowed limit, we can calculate the mean of the last n accesses to the data structure and use that for comparison against the limit.

An even more adaptive scheme is that if several collections in a row are aborted, the mark-split collector can choose to (exponentially) back off for a number of collections and try again later to see if the number of free intervals has become smaller. Many other adaptive schemes are of course possible.

Naturally, no matter which adaptive scheme is adopted, no single collection that starts with mark-split and reverts to mark-sweep can ever be faster than the corresponding mark-sweep collection. However, notice that an adaptive scheme consisting of a set of complete mark-split collections and a set of collections that reverted to mark-sweep, *can* be faster than the corresponding set consisting of mark-sweep collections only. This is because the complete mark-split collections can finish significantly faster than their mark-sweep counterparts. As we will see in the next section, a mark-split collection being faster than a mark-sweep collection is not uncommon.

7. Performance Evaluation

We ran all benchmarks on an four-processor Intel Xeon 2GHz with hyper-threading, 512 KB of cache, and 8 GB of RAM, running Linux version 2.6.9-5.ELsmp. However, during the performance evaluation the garbage collector was run in single-threaded mode. We had this machine completely at our disposal when running the benchmarks, so we can reasonably expect that the time measurements we report have little noise.

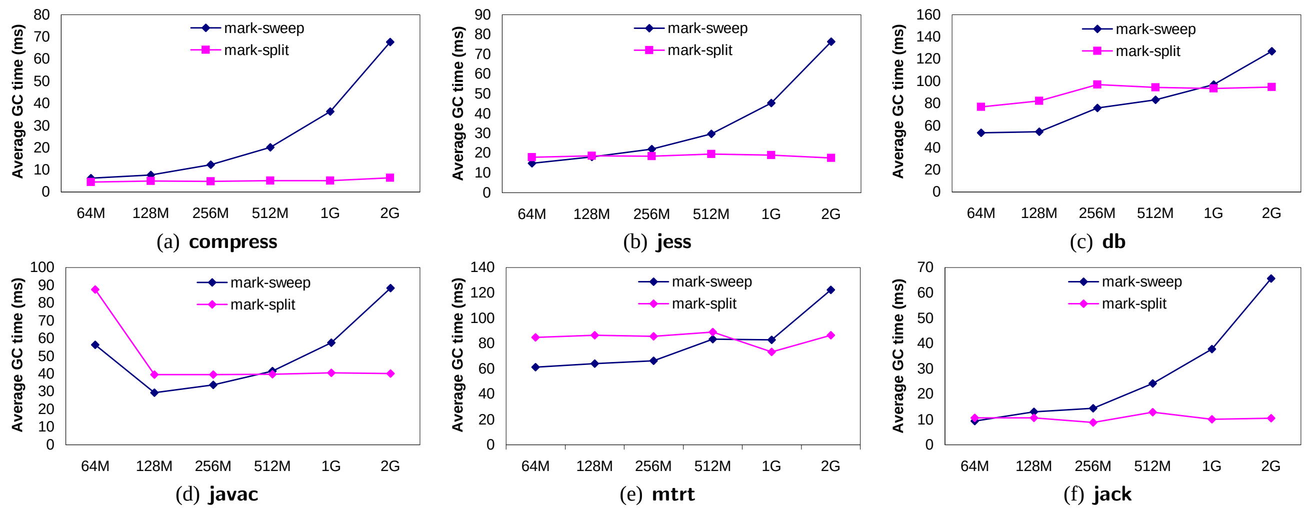
7.1 Performance on SPECjvm98

The first benchmark set we used were programs from SPECjvm98. The JRockit system can be started with a command-line option that specifies the size of the heap to use. When this heap is exhausted, a garbage collection is triggered. Since none of these programs allocates more than 300 MB during a single run, we run each of them 50 times without restarting the system or enforcing any garbage collection between runs of the same benchmark. Calls to `System.gc()` that the driver program performs at the start and end of each benchmark were *not* considered data for analysis. We run the SPECjvm98 programs with the mark-sweep garbage collector of JRockit using various different heap sizes, namely all power-of-two sizes in the range 64 MB–2 GB, and recorded the live data set sizes at the times of garbage collection. Their maximum values are shown in Table 2. Using JRockit, the benchmarks exhibit maximum live data set sizes ranging from approximately 4 MB to 46 MB. The total amount of allocation is also shown.

Benchmark	Max live data	Total allocation
compress	6,433 KB	125 MB × 50
jess	6,336 KB	326 MB × 50
db	10,025 KB	87 MB × 50
javac	46,468 KB	288 MB × 50
mtrt	18,597 KB	180 MB × 50
jack	4,473 KB	292 MB × 50

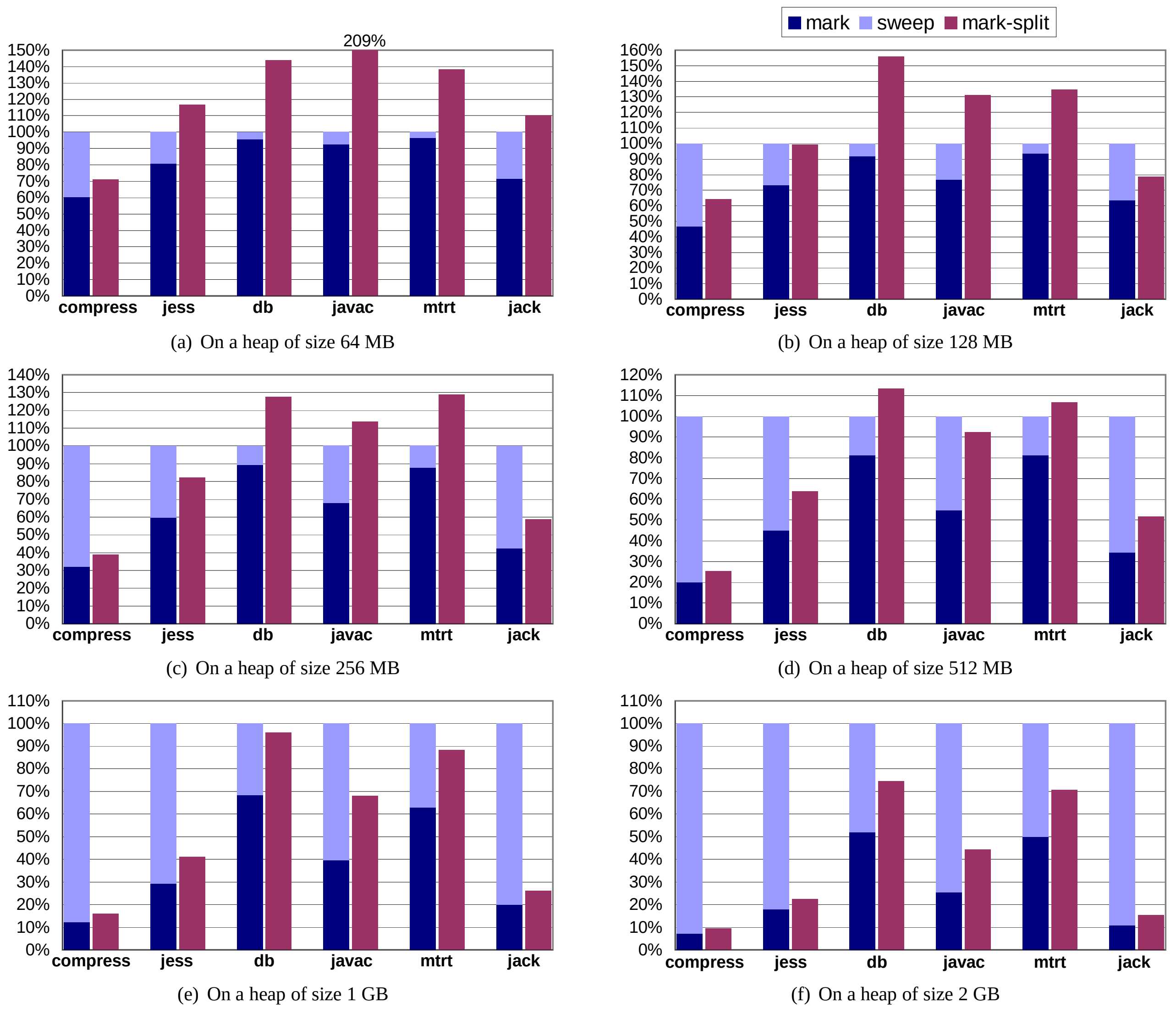
Table 2. Size characteristics of the SPECjvm98 benchmarks.

Total GC time performance We compared the performance of the mark-sweep and the mark-split collector using the set of heap sizes mentioned above. Total garbage collection times for these benchmarks are shown in the graphs of Figure 7. At 64 MB, which is the smallest heap size, the mark-sweep collector has better



(a) **compress** (b) **jess** (c) **db** (d) **javac** (e) **mtrt** (f) **jack**

Figure 8. Average garbage collection times for mark-sweep and mark-split when varying the size of available heap from 64M to 2G.



(a) On a heap of size 64 MB (b) On a heap of size 128 MB (c) On a heap of size 256 MB (d) On a heap of size 512 MB (e) On a heap of size 1 GB (f) On a heap of size 2 GB

Figure 9. Normalized total garbage collection times for mark-sweep and mark-split collection with different heap sizes.

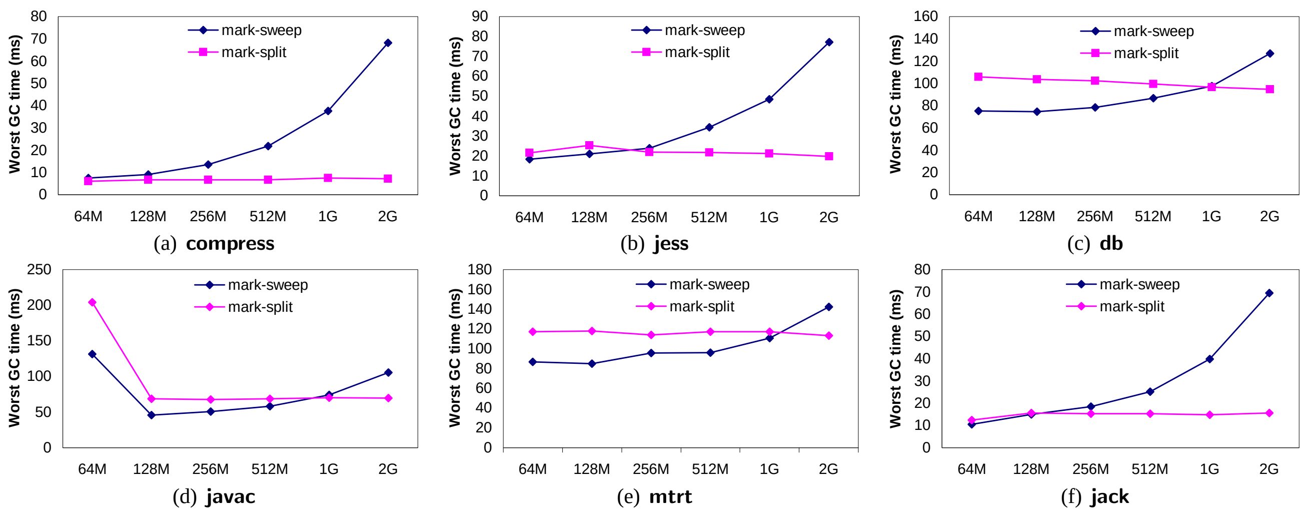


Figure 10. Worst garbage collection times for mark-sweep and mark-split when varying the size of available heap from 64M to 2G.

performance than mark-split in all but one benchmark (**compress**). Other than that, all graphs show a similar picture. Naturally, as the size of the available heap increases, fewer garbage collections occur. As a result, the total time spent in GC drops; **javac** is the only exception. Also, notice that as the initial heap size increases, for each benchmark there comes a point where mark-split outperforms mark-sweep. The explanation is simple: the sweep phase becomes the bottleneck of the mark-sweep collector. The graph of **javac** in particular (Figure 7(d)) shows the problem with mark-sweep quite clearly.

Average garbage collection pauses show the performance differences between mark-sweep and mark-split collection as the heap size increases more clearly; see Figure 8. One conclusion that can be drawn from these graphs, is that a system that employs a mark-split collector relieves the programmer from having to worry about specifying a small rather than a big initial heap size solely for the sake of the garbage collector.[5] When the size of the live data set is small or moderate in size, the mark-split collector is naturally immune to bad performance solely due to the size of the heap which is collected.

Overhead on marking We show a detailed breakdown of the time spent in the different phases of the two algorithms in Figure 9. In the figure, all times are shown normalized to the mark-sweep total GC times. For mark-sweep, the sweep phase occupies a significant portion of the time spent in garbage collection. When using a heap size of 64MB, the sweep phase takes 4–40%. As the heap grows, the percentage of time spent in the sweep phase increases. At 2 GB, the sweep phase accounts for 48–93% of the total garbage collection time. On the other hand, the overhead of maintaining live intervals during marking is, in absolute terms, between 2–64% of the total mark-sweep GC time; in relative terms, it adds 18–74% cost to the mark phase. As the heap grows, paying this overhead to avoid the cost of the sweep phase pays off.

Pause times As can be seen in Figure 10, the mark-split collector has smaller worst garbage collection pauses than mark-sweep; **db**, **javac**, and **mtrt** are the exceptions in small heap sizes. The remaining three benchmarks (**compress**, **jess**, and **jack**) clearly show the complexity difference between mark-sweep and mark-split.

[5] Of course, a smaller heap size might be desirable for other reasons (e.g., to run many applications on the same platform or to improve cache performance).

Space overhead due to the free interval tree A natural concern for mark-split collection is the additional space required for the maintenance of the tree of free intervals. Table 3 shows the number of nodes of this tree, both the maximum during (second column) and the maximum at the end of garbage collection (third column).

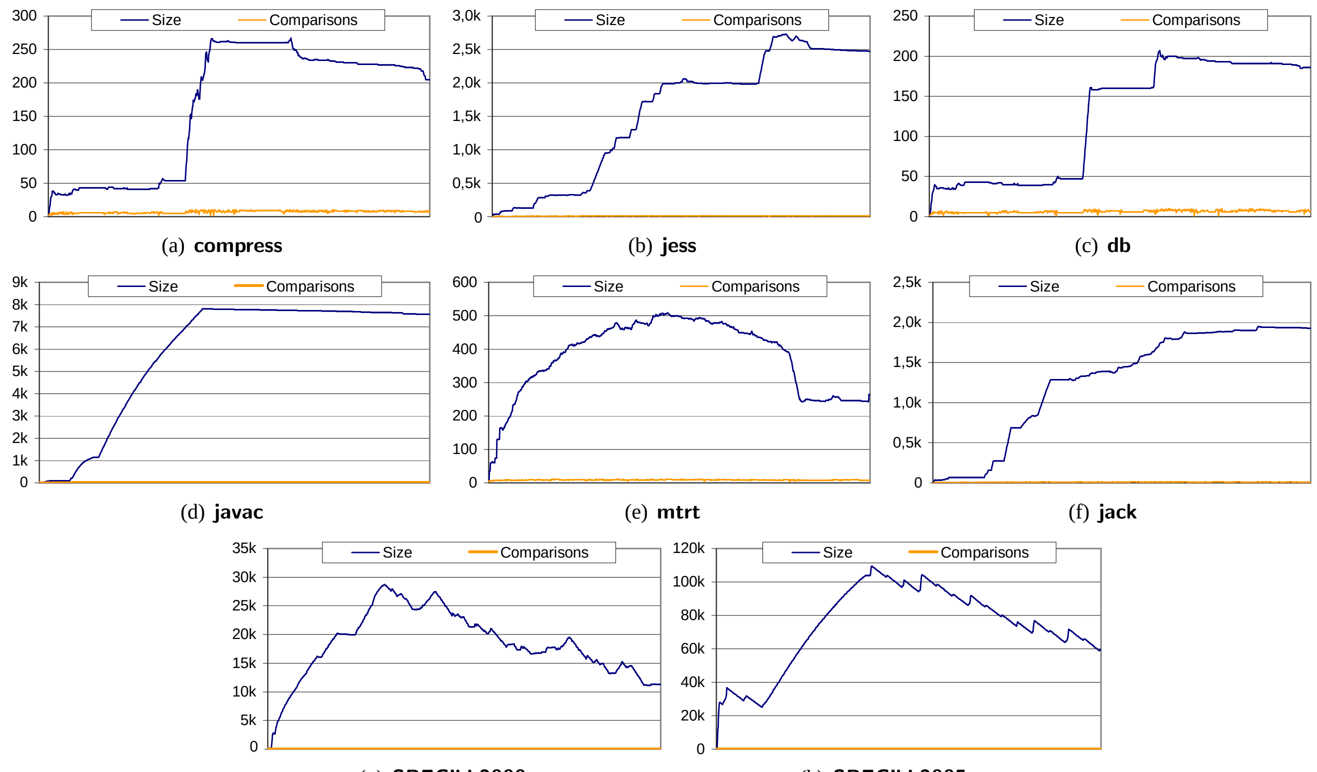
In our current implementation, each tree node occupies five words. The next column of the table shows the space required for storing these nodes as a percentage of the maximum size of the live data set for the benchmark. As can be seen, the space overhead for maintaining the tree of free intervals for these benchmarks is extremely small; less than 1% of the live data set. In fact, this additional space is so small that it can be considered practically negligible. For example, the biggest of these trees, the one for the **javac** benchmark, requires $(7,802 * 5 * 4) \approx 153$ KB on a 32-bit machine. To put this number in perspective, assuming that each mark bit is stored in a single bit, a heap of size 64 MB requires 512 KB of memory for storing the mark bits alone. If a copying collector could possibly guess the size of the live data set and allocate only that amount of memory for its to-space, it would need 46,468 KB for this benchmark (cf. Table 2). In short, in the worst case for these benchmarks, the free interval tree requires roughly $1/4$ the memory needed for the mark bits of a mark-sweep collector, and only $1/303$ of the additional space that a copying collector would minimally require.

The last two columns of Table 3 show the total and average number of comparisons performed during the extended mark phase of mark-split; i.e. the number of tree nodes visited in order to locate the interval that marking possibly affects. The numbers are taken from the collection that resulted in the largest tree during the execution of each benchmark. Finally, in Figure 11 we show plots for the total number of nodes in the tree and number of nodes examined while mark-split garbage collection takes place. Of course, these plots are highly dependent on the order of marking, but they confirm that in most cases the free interval tree grows monotonically in size — only **mtrt** shows some significant reduction — and that in our implementation the tree is indeed balanced.

As mentioned, in our current implementation we linearize the free interval tree at the end of the collection and translate it to the in-heap representation of the free list that JRockit normally uses. For these benchmarks, the cost of this translation is between 0.1% and 3.5% of the cost of mark-split collection and as such is negligible.

Benchmark	Number of nodes Max	Number of nodes Final	Max tree size as % of max live data	Comparisons Total	Comparisons Avg
compress	267	205	0.0811%	56k	7.1
jess	2,731	2,472	0.8419%	270k	10.2
db	207	186	0.0403%	45k	6.3
javac	7,802	7,561	0.3279%	456k	12.0
mtrt	509	275	0.0535%	1,320k	9.1
jack	1,953	1,928	0.2528%	199k	9.6

Table 3. Size characteristics of the free interval tree.



(a) **compress** (b) **jess** (c) **db** (d) **javac** (e) **mtrt** (f) **jack** (g) **SPECjbb2000** (h) **SPECjbb2005**

Figure 11. Number of nodes in the free interval tree and nodes examined by marking during mark-split collection (X-axis is time).

Benchmark	Sizes Heap	Sizes Max live data	Num GCs	Number of Nodes Max	Number of Nodes Max Final	Max tree size as % of Heap	Max tree size as % of Max live data	Comparisons Total	Comparisons Avg
SPECjbb2000	2 GB	329,740 KB	79	28,792	13,508	0.0126%	0.0800%	17,840k	14.1
SPECjbb2005	2 GB	879,080 KB	103	109,574	60,674	0.1020%	0.2434%	148,710k	16.4

Table 4. Characteristics of the two SPECjbb benchmarks.

7.2 Performance on SPECjbb

As mentioned, when the size of the live data set is relatively big compared to the size of the heap, using mark-split is probably not a good idea. Still, we wanted to see how bad its performance can be. The **SPECjbb** benchmarks, and **SPECjbb2005** in particular, are designed to stress the garbage collector. Indeed, the size of their live data set is big (cf. Table 4) and the mark phase heavily dominates the sweep phase when using mark-sweep.

We ran the **SPECjbb2000** benchmark using 8 warehouses and the **SPECjbb2005** using 16. Both benchmarks were run on a 2 GB heap. In this setting, **SPECjbb2000** spends 17% of the total garbage collection time in the sweep phase, while **SPECjbb2005** spends only 4%. This means that the mark-split collector has basically no chance of improving performance in these benchmarks. Still, its performance is only 31% worse than the mark-sweep collector in **SPECjbb2000**, and 69% worse in **SPECjbb2005**.

We also tried an adaptive scheme that first tries mark-split in all collections but when a tree size limit is reached the collection switches to the mark-sweep collector. The adaptive scheme is slower than mark-sweep in these two benchmarks, but at least it manages to keep the amount of damage under control. In **SPECjbb2005**, all but two collections switch to the mark-sweep collector using this scheme. The overall performance is only 7% worse than directly using the mark-sweep collector; see Figure 12. In **SPECjbb2000**, 12 out of a total of 79 collections use mark-split and the rest dynamically switch to mark-sweep. The

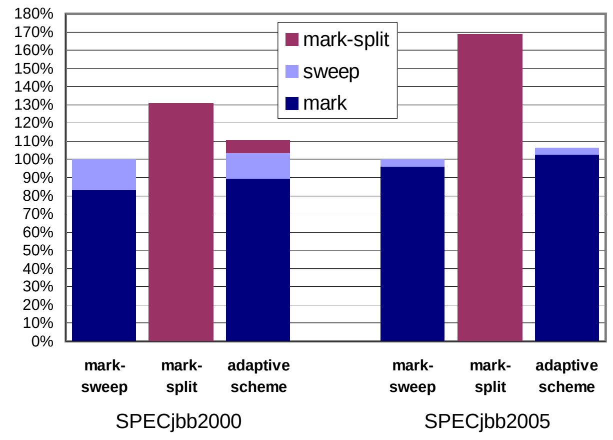


Figure 12. Normalized collection times for SPECjbb versions.

adaptive scheme performs 10% worse than the mark-sweep collector. It might not be noticeable in Figure 12, but compared to the mark-sweep collector there is actually a 16% decrease in time spent in the sweep phase and a 7% increase in mark time when using the adaptive scheme on this benchmark. This increase in mark time is larger in **SPECjbb2000** than in **SPECjbb2005** because in **SPECjbb2000** the free interval tree grows at a slower pace, and it takes quite a while before mark-split collections revert to mark-sweep. To see this, contrast the *y*-axis scales and the rates of increase in the size of the tree in Figures 11(g) and 11(h).

8. Concluding Remarks

We have proposed mark-split, a new non-moving garbage collection algorithm whose cost is proportional to the size of the live data set. When the size of the live data set is small compared with the size of the heap which is collected or when live data naturally cluster together, the use of mark-split achieves better performance than mark-sweep. As shown by our performance evaluation, this improvement is often significant. In fact, because the two algorithms have a difference in complexity, mark-split allows for an arbitrarily large decrease in total garbage collection times simply by performing the most natural action: increasing the heap size. This is something which is not possible when using mark-sweep.

At the very least, mark-split is just another garbage collection algorithm that improves performance in some situations. Independently of the practical usefulness of the actual algorithm, the insight that sweeping is unnecessary if one maintains information about free intervals while marking also offers a different — and in our opinion intriguing — way of looking at the problem of memory management.

Acknowledgments

We thank Joakim Dahlstedt, Mattias Joelson, David Lindholm, and Noora Peura of BEA Systems Stockholm Engineering AB for letting us play with JRockit, for their help in conducting the performance evaluation, and for their hospitality. We also thank the anonymous reviewers of ISMM'06 for comments that have helped us improve the presentation of our work.

References

[1] A. Andersson. General search trees made simple. In *Proceedings of the Third Workshop on Algorithms and Data Structures (WADS'93)*, volume 709 of *LNCS*, pages 60–71. Springer-Verlag, Aug. 1993.

[2] H. G. Baker. The treadmill, real-time garbage collection without motion sickness. *SIGPLAN Notices*, 27(3):66–70, Mar. 1992.

[3] BEA JRockit: Java for the enterprise. Technical White Paper (publicly available), Dec. 2003.

[4] H.-J. Boehm. Mark-sweep vs. copying collection and asymptotic complexity, 1995. Web page describing an IWMM'95 presentation.

[5] H.-J. Boehm. Reducing garbage collector cache misses. In T. Hosking, editor, *Proceedings of ISMM'2000: ACM SIGPLAN International Symposium on Memory Management*, pages 59–64, New York, N.Y., Oct. 2000. ACM Press.

[6] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, Nov. 1970.

[7] Y. C. Chung, S.-M. Moon, K. Ebcioĝlu, and D. Sahlin. Selective sweeping. *Software – Practice and Experience*, 35(1):15–26, Jan. 2005.

[8] R. T. Dimpsey, R. Arora, and K. Kuiper. Java server performance: A case study of building effi cient, scalable Jvms. *IBM Systems Journal*, 39(1):151–174, 2000.

[9] R. J. M. Hughes. A semi-incremental garbage collection algorithm. *Software – Practice and Experience*, 12(11):1081–1084, Nov. 1982.

[10] J. L. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, Apr. 1960.

[11] C. Queinnec, B. Beaudoing, and J.-P. Queille. Mark during sweep rather than mark then sweep. In *Proceedings of Parallel Architectures and Languages Europe (PARLE'89)*, volume 365 of *LNCS*, pages 224–237, Berlin, Germany, June 1989. Springer-Verlag.

[12] B. Zorn. Comparing mark-and sweep and stop-and-copy garbage collection. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 87–98, New York, NY, USA, June 1990. ACM Press.