Reducing Sweep Time for a Nearly Empty Heap

Yoo C. Chung Soo-Mook Moon
Seoul National University
{chungyc,smoon}@altair.snu.ac.kr

Kemal Ebcioğlu
IBM T. J. Watson Research Center
kemal@watson.ibm.com

Dan Sahlin
Ericsson Utvecklings AB
dan@cslab.ericsson.se

Abstract

Mark and sweep garbage collectors are known for using time proportional to the heap size when sweeping memory, since all objects in the heap, regardless of whether they are live or not, must be visited in order to reclaim the memory occupied by dead objects. This paper introduces a sweeping method which traverses only the live objects, so that sweeping can be done in time dependent only on the number of live objects in the heap.

This allows each collection to use time independent of the size of the heap, which can result in a large reduction of overall garbage collection time in empty heaps. Unfortunately, the algorithm used may slow down overall garbage collection if the heap is not so empty. So a way to select the sweeping algorithm depending on the heap occupancy is introduced, which can avoid any significant slowdown.

1 Introduction

Automatic management of memory at run time is called *garbage collection* [1, 2]. Garbage collection eases the development of applications tremendously, as anyone who has had to chase down an obscure dangling pointer reference or memory leak can attest to.

Mark and sweep garbage collection [3] is a simple and popular way to implement garbage collection. However, most implementations of mark and sweep collection suffer from the problem that when the heap size is greatly increased, the time spent in each garbage collection also increases by a similar factor, regardless of the amount of objects that are actually live. This is because the sweep phase must visit every object in the heap, whether live or dead, in order to check if it was reached during the mark phase.

This linear dependence of sweep time on the heap size poses a fundamental lower bound on the overall sweep time when using traditional mark and sweep garbage collection, since even though the collection frequency is inversely proportional to the heap size, the linear increase in time for each individual sweep prevents any further decrease in overall sweeping time. (That is, the time doesn't decrease until the heap size becomes large enough so that garbage collection doesn't occur at all. We'll pretend that the heap size will never grow *that* large.)

This is one of the reasons why copying collection is sometimes claimed to be better than mark and sweep collection, since the time spent by copying collection depends linearly on the amount of memory occupied by live objects instead of depending on the size of the heap. Because of this, when the heap is very large and the number of live objects is very small (such as is expected with the youngest generation of a generational garbage collector [4]), copying collection is usually much faster than mark and sweep collection [5].

On the other hand, copying collectors do have problems of their own. For example, they can only use half of the heap at any given time, regularly copy all live data during each collection, and require that all references be precisely identified (there are ways to work around some of these problems [6, 7], but they add additional complexity to implementations). Thus many systems use mark and sweep garbage collection instead.

This paper shows that mark and sweep garbage collection can also use time independent of the size of the heap using an algorithm which will be called *selective sweeping*. This partially offsets one of the disadvantages that mark and sweep garbage collection suffers against copying garbage collection. Since the algorithm can be slower than the traditional sweeping algorithm when there are many live objects, a way to avoid this slowdown is also introduced.

For simplicity, only simple non-incremental garbage collection is considered, and the heap is assumed to be contiguous. (It is a simple matter to extend the approach to a non-contiguous heap. It should also not be too difficult to integrate the approach with existing incremental mark and sweep garbage collectors, although whether this would be worthwhile is rather questionable.)

The remainder of this paper is organized as follows. Section 2 gives an overview of traditional mark and sweep garbage collection. Section 3 introduces selective sweeping, and section 4 shows how to avoid any significant slowdown that might result from using this algorithm. Section 5 describes an actual implementation and reports some empirical results. Section 6 gives pointers to other approaches that also work on reducing or hiding sweep time. Finally, section 7 concludes.

POPL 2000 Boston MA USA

2 Traditional Mark and Sweep Collection

This section introduces traditional mark and sweep garbage collection and its time complexity, and shows why there is a lower bound on the overall garbage collection time.

2.1 The Algorithm

In the simplest version of mark and sweep garbage collection, garbage collection is done by marking all the reachable objects (called *live objects*, since they may eventually be used by the program) in the *mark phase*, and then reclaiming memory occupied by unreachable objects (called *dead objects*, since it is certain that they can never be used again) in the *sweep phase*.

The mark phase is done by first marking the objects in the root set (the set of objects that can be referenced directly by the application, e.g. machine registers or global variables), and then continually marking unmarked objects that are referenced from marked objects, until there are no more objects to be marked. When the mark phase is done, all objects that can be reached from the root set would have been marked.

Algorithm 1 is the typical way to mark all objects that are reachable from the root set. Prior to marking with this algorithm, all objects in the heap must have been unmarked (root objects will not be considered as part of the heap). Otherwise, a path to a still reachable object might not be completely traversed because of the existence of a marked object on the path. This can be ensured by unmarking all marked objects in the heap during the sweep phase of the previous garbage collection.

Algorithm 1 Marking reachable objects

```
M ← {r : r is in root set}
while M ≠ ∅ do
  a ← some element of M
  M ← M − {a}
  for each object b referenced by a do
    if b has not been marked then
      mark b, M ← M ∪ {b}
    end if
  end for
end while
```

M in algorithm 1 contains the set of marked objects whose children may or may not have been marked (this will be called the *boundary set* in this paper). It is in effect the boundary between marked objects whose children have also been marked and objects that have not been marked, as in figure 1.

At any point after the initial stage is complete, i.e. after all the root objects have been put into the boundary set, any unmarked objects reachable from the root set must also be reachable from the boundary set. Thus when there are no more objects in the boundary set, there are no more unmarked objects reachable from the root set, and the mark phase is complete.

Depending on the order objects are removed from the boundary set, the algorithm traverses objects in depth-first order using a mark stack (LIFO), or it may traverse objects in breadth-first order using a mark queue (FIFO). These are in fact the most common orderings used in mark and sweep collectors.

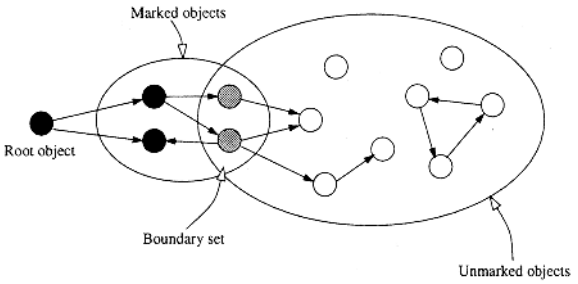


Figure 1: An example snapshot of the marking process. Nodes are colored according to Dijkstra's tricolor marking abstraction [8], where marked objects whose children have been marked are black, marked objects whose children might still be unmarked are grey, and unmarked objects are white.

The sweep phase is when memory occupied by dead objects is actually reclaimed. This is traditionally done by visiting each object in the heap one by one and reclaiming the memory occupied by an object if it is not marked, since only marked objects can be reached from the root set. Also, any marked object is unmarked in order to prepare for the next garbage collection, as can be seen in algorithm 2.

Algorithm 2 Traditional sweeping

```
for each object a in the heap do
  if a is marked then
    unmark a
  else
    reclaim memory occupied by a
  end if
end for
```

2.2 Time Complexity

During the mark phase, each live object is pushed and popped exactly once from the boundary set. Each reference field of every reachable object is also visited exactly once. Thus if R is the total number of references in every live object and n is the number of live objects in the heap, $O(n + R)$ time is used. If L is the amount of memory occupied by live objects, then marking can be done in $O(L)$ time, since $R \leq L$ and $n = O(L)$.

During the sweep phase, every object is checked to see whether they are marked or not. If it is marked, it is then unmarked. If it is not marked, the memory occupied by the object is reclaimed. If each reclamation can be done in constant time (which is certainly possible, e.g. using segregated free lists [9]), then $O(N)$ time is used during the sweep phase, where N is the total number of objects on the heap.

When the sizes of objects in the heap are independent of the heap size, which will almost certainly be the case,[1] then N will be proportional to the heap size. Thus sweep time is in $O(H)$, where H is the size of the heap.

When the memory occupied by live objects is a significant and constant fraction of the heap size, the time used by

[1]The only way object sizes can be dependent on the heap size is for them to vary as the heap size varies, but such programming styles are extremely rare.

the mark phase and the sweep phase are asymptotically similar, since $O(L) \equiv O(H)$ when $L \propto H$. In practice, marking time dominates in such cases.

However, when the heap occupancy is low, sweep time dominates, since the sweep time increases linearly with the heap size while the marking time stays relatively constant. This is the main reason why increasing the heap size over a certain threshold does not help very much in reducing the overall time spent in garbage collection using traditional mark and sweep collection, since even though the frequency of collections depends inversely on the heap size, the time spent in each collection increases linearly with the heap size.

This is in contrast to copying collection, where the frequency of collections depends inversely on the heap size while each collection takes relatively constant time, so that the overall garbage collection time is inversely proportional to the heap size [5].

(More accurately, the frequency of collections is inversely proportional to the size of the *dead* portion of the heap for both mark-sweep and copying collection. This does not change the above assertions for higher heap sizes, since the dead portion occupies most of the heap at such sizes.)

3 Selective Sweeping

As was pointed out in section 2.2, sweep time, which depends linearly on the size of the heap, dominates the garbage collection time when there are very few live objects compared to the size of the heap. This is because every object in the heap, regardless of whether they are live or dead, must be traversed so that memory occupied by unreachable objects can be reclaimed.

However, if one knows that there are no live objects between two given live objects, one can reclaim the space between them in a single operation, instead of having to visit all the objects between them, since these objects are obviously dead.

This can also be done in traditional sweeping by checking if all objects between two marked objects are unmarked. If this is the case, then the memory occupied by the unmarked objects can be reclaimed in a single operation, instead of reclaiming memory for each dead object separately. However, since each object is still visited at least once, sweep time still depends linearly on the heap size.

Unfortunately, it is difficult to avoid having to deal with every object in the heap when algorithm 1 is used as the marking algorithm, since it leaves behind no information about which objects are live, except for the marking that is done on the live objects. This forces one to check every object to see if they are marked.

So as the first step in making sweep time independent of the heap size, instead of discarding objects that have been removed from the boundary set as in algorithm 1, a set consisting of all the objects that are reachable should be constructed.

Algorithm 3 is able to construct the set of marked objects while marking all the live objects. It is an adaption of Cheney's scanning algorithm [10], which is typically used in copying collectors.

Algorithm 3 essentially works like algorithm 1. Unlike algorithm 1, which only maintains the boundary set (the set of marked objects which may still refer to objects that have not yet been marked), algorithm 3 also maintains the set of marked objects which are known to refer only to other marked objects.

Algorithm 3 Constructing set of live objects

```
s ← 0, t ← −1
for each root object r do
  t ← t + 1, m_t ← r
end for
while s ≤ t do
  for each object a referenced by m_s do
    if a has not been marked then
      mark a, t ← t + 1, m_t ← a
    end if
  end for
  s ← s + 1
end while
{m_i : 0 ≤ i ≤ t} is set of live objects
```

Comparing the algorithm to algorithm 1, we see that the set $\{m_i : 0 \leq i < s\}$ is the set of marked objects which are known to refer only to other marked objects, and that the set $\{m_i : s \leq i \leq t\}$ is the boundary set. When the algorithm completes, the boundary set is empty and the set $\{m_i : 0 \leq i \leq t\}$ contains all the objects reachable from the root set.

Each object is pushed into and popped from the boundary set exactly once, and each reference in every live object is also visited exactly once, as in algorithm 1. So obtaining the set of live objects takes the same time as marking all the live objects, which is in $O(L)$, where L is the amount of memory occupied by the live objects.

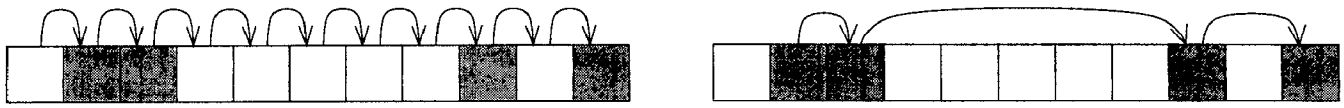
However, the set of live objects alone does not allow one to check the nonexistence of live objects between two given live objects in constant time, because without checking all the objects in the set, one cannot be sure if there are no live objects between two given live objects. This is because the order which the objects were entered into the set is independent of the order which they are positioned in the heap.

In order to make this possible, we can sort the objects by address. Since it is guaranteed that there are no live objects between two consecutive objects in the sorted set, it is now possible to sweep the heap without looking at every object in the heap, as in algorithm 4 (which will be called *selective sweeping* in order to distinguish it from traditional sweeping).

The second and last conditional statements in algorithm 4 are required so that the boundaries of the heap are properly dealt with. These statements can be removed if there are dummy objects, considered live at sweep time, at the start and end of the heap, which would act as sentinels. Using such sentinels would also absolve of the need for handling the special case where there are no live objects, since such a case could never happen.

Algorithm 4 can reclaim the memory in each gap between the live objects in constant time (assuming that the reclamation operation itself takes constant time, of course). This is in contrast to traditional sweeping, where reclaiming the memory for each gap takes time proportional to the number of dead objects that reside in them (see figure 2). Since there can be at most $n+1$ gaps when there are n live objects, the time taken to reclaim all the memory is proportional to the number of live objects.

Since the set of live objects is maintained separately, the sweep phase does not need to check whether each object



(a) With traditional sweeping

(b) With selective sweeping

Figure 2: Object traversal during sweeping. Shaded nodes denote live objects.

Algorithm 4 Selective sweeping

```
M is the set of live objects
if M = ∅ then
  free all memory in heap
else
  sort M so that m_0, ..., m_n ∈ M are in order of address
  unmark m_0
  if m_0 does not start at beginning of heap then
    free memory before m_0
  end if
  for 1 ≤ i ≤ n do
    unmark m_i
    if m_{i-1} and m_i are not adjacent then
      free memory between m_{i-1} and m_i
    end if
  end for
  if m_n does not end at end of heap then
    free memory after m_n
  end if
end if
```

is marked or not (in fact, it is guaranteed that only the objects it does visit are marked). In this case, marking is required only by the mark phase in order to avoid redundant traversals, unlike in traditional mark and sweep, where it is also required by the sweep phase in order to check whether each object is reachable from the root set.

Selective sweeping takes $O(n)$ time when reclaiming memory, where n is the number of live objects and assuming that each reclamation operation takes constant time. However, since the set of live objects must be sorted by address prior to reclaiming memory, sweeping time will be in $O(n + f(n))$ time, where $O(f(n))$ is the time required for sorting n objects.

Since comparison sorts such as merge sort or heap sort take $O(n \log n)$ time to sort n objects, sweeping can also be done in $O(n \log n)$ time, which is independent of the size of the heap. Or a distribution sort such as radix sort [11] could be used, which has a $O(n \log H)$ worst case time bound, where H is the size of the heap, but is much faster than comparison sorts at large n and in fact can be done in $O(n)$ time on average.

These worst case time bounds are asymptotically worse compared to the $O(H)$ time bound of traditional sweeping when $n \propto H$, where n is the number of live objects and H is the heap size.[2] However, when n is a very small fraction of H, i.e. when the heap is nearly empty of live objects, the time spent in sweeping with the selective algorithm would be insignificant compared to the time spent in sweeping with the traditional algorithm.

Also, since the time spent in sweeping is now independent of the heap size, overall garbage collection time is now inversely proportional to the heap size. This is because each individual garbage collection takes relatively constant time, while the collection frequency is inversely proportional to the heap size. This is in contrast to traditional mark and sweep collection, where there is a lower bound to the overall sweep time because of the linear increase of sweep time along with the size of the heap.

[2]Certain implementations could use $O(n \log n)$ time or worse when freeing memory [12, 13], in which case this statement would not apply.

4 Adaptive Mark and Sweep Collection

When the number of live objects is insignificant compared to the number of all the objects in the heap, it is quite obvious that sweep time can be reduced significantly using selective sweeping. However, when the number of live objects is a significant fraction of the total number of objects, traditional sweeping would probably be better, since it is linear on the heap size and requires no extra step (such as the sorting step in selective sweeping) before making only a single pass through the heap. Also, traditional sweeping typically will use much less memory during the marking phase, since it does not have to keep track of all the live objects.

Therefore it can be advantageous to select the sweeping algorithm depending on the number of live objects in the heap.

4.1 Adaptive Marking

It is possible to use traditional sweeping to reclaim dead objects even when algorithm 3, which constructs the set of live objects, is used in the mark phase. However, it would be a waste of memory, since algorithm 1, which only maintains the boundary set, can mark all the live objects with the same efficiency. It is also much more prone to overflows when a fixed array is used to maintain the set of live objects.

On the other hand, using the traditional algorithm for marking does not help much to reduce sweep time for a heap with few live objects. So it might be better if there were some way to select the marking algorithm to be used depending on the number of live objects.

This is not simple to do before starting the mark phase, since the number of live objects is not known. One possible solution would be to predict the number of live objects based on the detected number of live objects during previous garbage collections. This would be akin to the adaptive tenuring policy in a generational garbage collector as proposed by Ungar and Jackson [14], which adjusts the tenuring threshold based on the number of tenured objects during past minor collections.

Unlike their adaptive tenuring policy, however, there is no need to obtain the actual number of live objects, but only to detect whether the number of live objects exceeds some certain threshold. This can be done very simply in the marking phase itself.

This can be done by checking whether the number of live objects go over a certain threshold when inserting objects into the set of live objects with algorithm 3. If at a certain point this threshold is exceeded, the marking algorithm is switched to algorithm 1, which does not construct the set of live objects.

This is possible because the boundary set of algorithm 3, $\{m_i : s \leq i \leq t\}$, can be used without any change as the boundary set for algorithm 1, since both have exactly the same roles (they are both the set of marked objects which may refer to unmarked objects).

Also, detecting whether the number of live objects becomes greater than some threshold does not need to introduce any additional overhead, since it can use the same mechanism used to detect stack overflows when marking with an explicit array-based mark stack (e.g. using explicit checks or using a guard page [15]), which would have been used anyways in order to maintain correctness. (When marking with other methods, such as with a list-based mark queue, marking would already be expensive enough that maintaining a simple count would be a negligible overhead.)

With these techniques, it is now possible to select the sweeping algorithm based on the number of live objects in the heap without wasting too much memory in the mark phase.

4.2 Adaptive Sweeping

When the number of live objects is a significant fraction of the total number of objects in the heap, it would be better to use traditional sweeping, whereas when there are very few live objects, it would be better to use selective sweeping, which only traverses the live objects.

However, when using selective sweeping, there is the issue of what kind of sorting algorithm to use when sorting live objects by address.

Using radix sort with a large enough radix, sweeping can be done in time practically linear on the number of live objects (see section 5.1). But when the number of live objects is small enough, comparison sorts might be better since radix sort does multiple passes, and the difference in the complexity is less important for smaller instances.

Thus, when using selective sweeping, a comparison sort should be used when the number of live objects is very small, and a distribution sort such as radix sort should be used when the number is much larger (though still small compared to the size of the heap).

The exact range of the number of live objects where selective sweeping with a comparison sort, selective sweeping with a distribution sort, and traditional sweeping is best will depend on the application and system. It may be the case that there is no range where selective sweeping with a distribution sort is better than selective sweeping with a comparison sort, or vice versa. But it is certain that selective sweeping will be much better than traditional sweeping when the heap occupancy is very low.

5 Actual Case Study

This section outlines the implementation of a garbage collector used in a real system which uses selective and adaptive sweeping, and gives some experimental results showing that the algorithms actually do work in practice.

5.1 Implementation

A garbage collector using the algorithms described in this paper was implemented for LaTTe [16]. LaTTe is a freely available Java virtual machine with a JIT compiler for the UltraSPARC. The JIT compiler uses a novel register allocation algorithm [17] and does many optimizations such as common subexpression, loop invariant code motion, customization, etc. It uses lightweight monitors [18] and does on-demand translation of exception handlers [19]. There is also a limited adaptive compilation framework, which includes a reasonably fast bytecode interpreter [20].

The non-incremental garbage collector uses mark and sweep and is partially conservative. Small and large objects are maintained separately, where a small object is an object with size less than a kilobyte. Allocation of small objects are done with pointer increments [21]. This allocation scheme causes small objects with widely differing sizes to be adjacent to each other, so the mark bits are bundled with the objects instead of using a separate mark bitmap.

Marking functions for each object type, which mark other objects referenced by an object, are generated at runtime. These are then called directly to mark objects referenced by an object, instead of using descriptor fields which describe the fields of an object. This approach is also used in other systems such as SmallEiffel [22].

The mark phase uses an explicit array-based mark stack when marking with the traditional algorithm. This mark stack is also used for storing the set of live objects.

For this experiment, enough memory is reserved for the mark stack so that overflows never occur. For actual use, a garbage collector which uses only selective sweeping would not be practical since there might not be enough memory to store the set of live objects. A garbage collector which chooses the sweeping algorithm adaptively, on the other hand, can use the same methods that traditional systems use to handle stack overflows [23].

Selective and adaptive sweeping are done only on the *small object area*, which is the portion of the heap which contains small objects. For this reason, the term "heap size" in this section actually means the size of the small object area. In addition, garbage collection is never triggered while allocating large objects.[3]

For the sorting algorithm used by selective sweeping, radix sort based on counting sort was used. It uses a radix of 2048, and *always* does three passes to sort the set of live objects. Since with three passes we can deal with an address range of 2048^3 bytes, or 8 gigabytes, we can always sort an arbitrary set of 32 bit pointers, since a 32 bit pointer has an address range of only 4 gigabytes.

Also, since each pass takes time in $O(n)$, where n is the number of elements to sort, and we always do three passes, the sorting algorithm used in this implementation takes time *linear* on n.

More details on LaTTe's memory management will be described in Chung et al. [24].

[3] In fact, management of large objects had not yet been properly implemented at the time of writing.

5.2 Experimental Results

For the test programs, _202_jess, _227_mtrt, and _228_jack from the SPECjvm98 benchmark suite [25] were used. _200_check, _201_compress, _209_db, and _222_mpegaudio were not used since they do almost no garbage collection, while _213_javac was not used because the number of live objects during its execution is so large such that selective and adaptive sweeping would do little to shorten garbage collection time within the memory available.

The test machine was a 270MHz UltraSPARC with 256MB of RAM running Solaris 2.6. The overall sweep times used by traditional and selective sweeping are compared in figure 3. (The actual timings from which the graphs were obtained are listed in the appendix.)

Most objects in _202_jess and _228_jack die quickly, so the amount of live objects is typically small (about 1MB). This is reflected in the results shown in figure 3, where selective sweeping is in fact always faster than traditional sweeping.

On the other hand, the amount of live objects is much larger for _227_mtrt (about 8MB). So at lower heap sizes, and hence higher heap occupancies, traditional sweeping is much faster, while at higher heap sizes, and hence lower heap occupancies, selective sweeping is much faster. This indicates that we should select the sweeping algorithm according to the heap occupancy.

Since each individual sweeping time for traditional sweeping is proportional to the size of the heap, while that of selective sweeping as implemented in this experiment is linear on the number of live objects, we can expect the threshold, under which selective sweeping would be better and over which traditional sweeping would be better, to be a constant fraction of the size of the heap. (This of course would not apply if a sorting routine which uses non-linear time were to be used.)

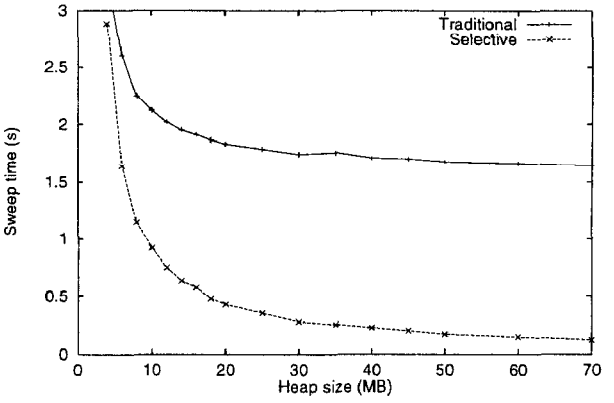
It would not be difficult to determine this fraction empirically. A good value for LaTTe would not apply to other systems that might use adaptive sweeping, of course, but it would give some insight on how adaptive sweeping would behave.

Since _227_mtrt was the benchmark that would benefit from adaptive sweeping, it was used to estimate a good value for the fraction. Figure 4 shows the sweeping times for the fractions 0 (equivalent to traditional sweeping), 1/512, 1/128, 1/64, and 1 (equivalent to selective sweeping).

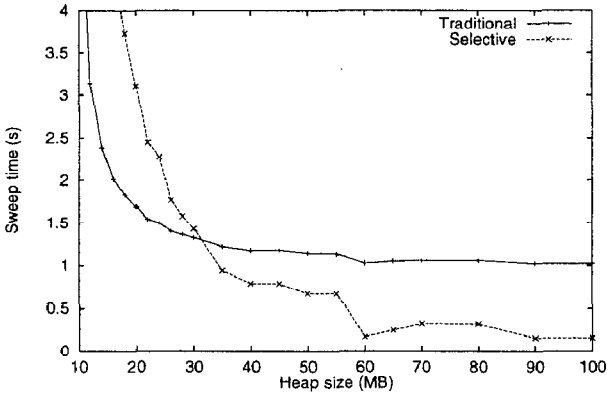
It should be noted that these fractions are *not* the ratio between the the amount of live memory and amount of total memory, but rather the ratio between the number of live objects and the amount of total memory. For example, if the average size of an object is 16 bytes (which is actually the practical minimum size for objects), then a fraction of 1/128 actually means that selective sweeping should be used only when the amount of live memory is less than an eighth of the size of the heap. (It might be better to use the ratio between the number of live objects and the number of all objects, but the memory allocator in LaTTe does not keep track of the number of allocated objects.)

As can be seen from figure 4, too large a fraction, such as 1/64 in this case, would cause selective sweeping to be used even when it would be much slower, so that adaptive sweeping is slower than traditional sweeping at low heap sizes.

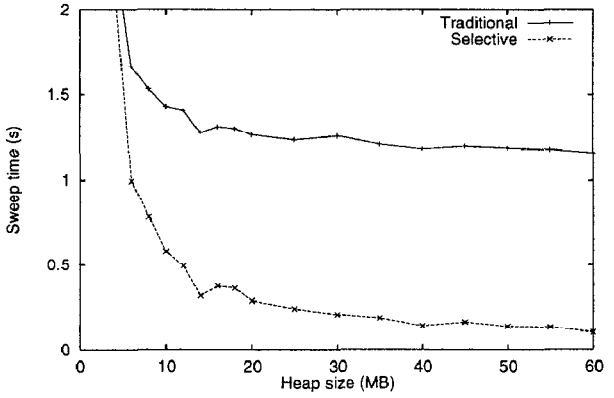
A too small a fraction such as 1/512, on the other hand, would cause traditional sweeping to be used even when it



(a) _202_jess



(b) _227_mtrt



(c) _228_jack

Figure 3: Comparison of sweep times.

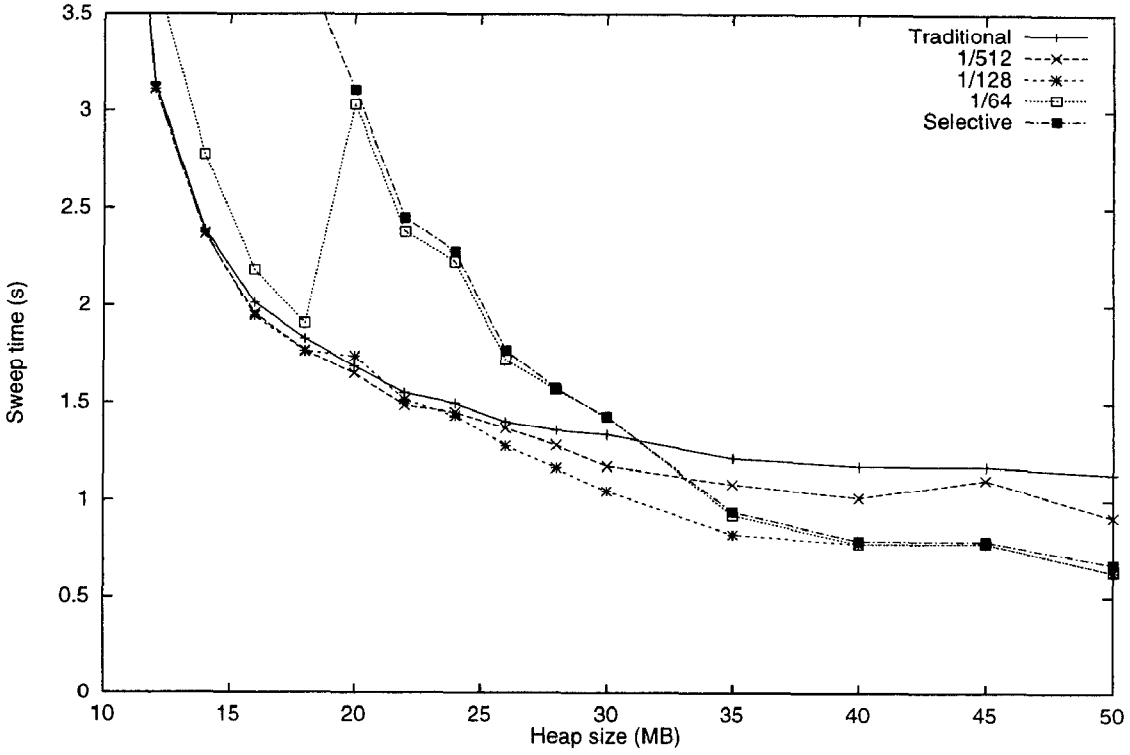


Figure 4: Adaptive sweeping times for _227_mtrt.

is the slower algorithm, so adaptive sweeping is slower than selective sweeping at high heap sizes. It is faster than traditional sweeping, though, so it might be better than using a too large a fraction.

However, with a fraction of 1/128, we can see that adaptive sweeping is never significantly slower than both traditional and selective sweeping. This is to be expected with a good value for the threshold, since at high heap occupancies traditional sweeping is used, while at low heap occupancies selective sweeping is used.

It is interesting to note that at intermediate heap occupancies, adaptive sweeping with a good threshold is noticeably faster than *both* traditional and selective sweeping. This is because the heap occupancy actually varies somewhat between each collection, so that the sweeping algorithm which is faster also varies. With a good threshold (such as 1/128 in figure 4), the faster of the two algorithms are used in each collection, so that adaptive sweeping ends up faster overall than either traditional or selective sweeping. Of course, this does not happen with a bad threshold (such as 1/64 or 1/512 in figure 4), so choosing a good threshold is important.

These results show that selective sweeping indeed results in much larger reductions in sweep time at low heap occupancies, and that choosing the sweeping algorithm at run time according to the heap occupancy can avoid the slowdown experienced by selective sweeping at high heap occupancies. In fact, doing so can result in shorter collection times than either just using traditional or selective sweeping.

6 Related Work

Sahlin [26] introduced an $O(n \log n)$ time algorithm for mark and compact garbage collection, which is also generally known as taking time proportional to the heap size. He uses an idea very similar to the one outlined in this paper, with a couple of important differences.

First, the implementation was for a mark and compact garbage collector, not a mark and sweep garbage collector, so some of the algorithms are substantially different.

Second, a separate *gathering phase* is done to construct a linked list of non-garbage memory blocks, which is basically a repetition of the mark phase with a few modifications. This is done separately so as to identify the garbage cells in which the links can be stored, which cannot be identified before completing a mark phase. With this method, no memory need be reserved for storing the set of live objects. This is important for a collector which uses pointer reversal for marking [27], since the whole point of using pointer reversal is to avoid having to reserve extra space for auxiliary data structures, such as a mark stack or a set of live objects.

This is in contrast to the approach used in this paper, where marking and construction of the set of live objects are done concurrently. On the other hand, memory for storing the set of live objects must be reserved separately, though this memory can be shared with the mark stack when an explicit array-based mark stack is used during marking.

Boehm [28] pointed out that comparisons of asymptotic complexity between garbage collection algorithms should also take allocation time into account, not just garbage collection time. When mark-sweep and copying collection is

compared in this way, they both turn out to have a time complexity of $O(H)$, where H is the size of the heap.

This is very apparent in garbage collectors that do lazy sweeping, where only enough sweeping is done during allocation in order to find an appropriate free memory chunk [29]. In this case, there appears to be no time spent in sweeping during garbage collection. However, sweeping done during allocation is still work that is being done, and selective sweeping could help reduce this work when the heap occupancy is low.

Reducing sweep time is also possible with mark bitmaps, since a fixed number of objects can be reclaimed in a single operation, using the fact that it is possible to check in a single operation whether the bits in a word, i.e. the mark bits in that portion of the mark bitmap, are all set to zero, i.e. all the objects in that portion are dead.

However, with mark bitmaps there is only a linear reduction in sweep time (albeit a large one), so there is still a theoretical lower bound on the overall garbage collection time. Marking also becomes more expensive instruction-wise than when the mark bits are bundled with the objects. On the other hand, the improvements in cache effects and sweeping time make it unclear whether the approach outlined in this paper would be better. In any case, there should be no serious difficulties in combining the two approaches, although whether this would be worthwhile is also unclear.

Baker [30] introduced the ultimate algorithm for sweep time reduction: mark-sweep using a doubly-linked list (which he called the *treadmill*), where the sweeping is done in *constant* time. Unfortunately, all objects must be of the same size for it to be usable, so its usefulness in general-purpose systems is limited.

Wilson and Johnstone [31] extended Baker's algorithm to multiple object sizes using segregated storage. However, unlike Baker's algorithm, where sweeping can be done in constant time, their algorithm required that each dead object be processed before being reused, so sweeping still uses time linear on the heap size (although they do sweep each dead object in a lazy manner during allocation).

These list-based mark and sweep garbage collectors are inappropriate as non-incremental garbage collectors, however, since list manipulation is typically much more expensive than using a simple array-based mark stack. They were designed as real-time garbage collection algorithms, though, so this is not a problem in their intended domain.

7 Conclusions

Mark and sweep garbage collection traditionally required time proportional to the heap size in order to sweep the heap. This is in contrast to copying collection, where copying takes time proportional to the amount of live memory, which is independent of the size of the heap.

This paper has shown that by sorting the live objects by address and visiting only the live objects during the sweep phase, mark and sweep garbage collection can also be done in time independent of the heap size. Thus, instead of having a fixed lower bound in overall sweep time as in traditional sweeping, selective sweeping can decrease the overall sweep time almost arbitrarily by increasing the heap size.

This allows mark and sweep garbage collection with selective sweeping to reduce overall garbage collection time by a much more significant factor than with traditional sweeping when the heap size is increased. In other words, selective sweeping uses much less time than does traditional sweeping when the heap occupancy is very low.

A low heap occupancy is not a desired trait of modern software, however. For one thing, even though memory prices have dropped dramatically, there has also been a corresponding increase in memory usage. Another point is that applications typically share memory with other applications, so it would be a bad thing if applications indiscriminately increased their heap size in order to lower the heap occupancy.

On the other hand, a low heap occupancy *is* a desired and expected trait of the youngest generation in a generational garbage collector. Thus selective sweeping can speed up minor collections for a generational garbage collector which uses only the mark and sweep algorithm.

Also, since the sweeping algorithm can be selected according to the heap occupancy at garbage collection time, the slowdown that would be experienced by the selective sweeping algorithm at high heap occupancies can be avoided. Thus selective sweeping is also useful in a simple non-incremental mark and sweep garbage collector when the heap occupancy varies by large amounts, by using traditional sweeping when the heap occupancy is high and selective sweeping when the heap occupancy is low.

References

[1] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, 1996.

[2] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994.

[3] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

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

[5] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[6] Michael Hicks, Luke Hornof, Jonathan T. Moore, and Scott M. Nettles. A study of large object spaces. In ISMM98 [32], pages 138–145.

[7] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Research Report 88/2, Compaq Western Research Laboratory, February 1988.

[8] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46.* Springer-Verlag, 1976.

[9] P. W. Purdom, S. M. Stigler, and Tat-Ong Cheam. Statistical investigation of three storage algorithms. *BIT*, 11:187–195, 1971.

[10] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11), November 1970.

[11] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*, chapter 5, pages 168–179. Addison-Wesley, second edition, 1998.

[12] C. J. Stephenson. Fast fits — new methods for dynamic storage allocation. In *Proceedings of the 9th Symposium on Operating Systems Principles*, pages 30–32, Bretton Woods, New Hampshire, October 1983. ACM Press.

[13] R. P. Brent. Efficient implementation of the first-fit strategy for dynamic storage allocation. *ACM Transactions on Programming Languages and Systems*, 11(3):388–403, July 1989.

[14] David Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, January 1992.

[15] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107. ACM Press, 1991.

[16] LaTTe: A fast and efficient Java VM just-in-time compiler. `http://latte.snu.ac.kr/`.

[17] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioğlu, and Erik Altman. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 128–138, New Port Beach, California, October 1999.

[18] Byung-Sun Yang, Junpyo Lee, Jinpyo Park, Soo-Mook Moon, Kemal Ebcioğlu, and Erik Altman. Lightweight monitor for Java VM. *ACM SIGARCH Computer Architecture News*, March 1999.

[19] SeungIl Lee, Byung-Sun Yang, Suhyun Kim, Seongbae Park, Soo-Mook Moon, Kemal Ebcioğlu, and Erik Altman. On-demand translation of Java exception handlers in the LaTTe JVM just-in-time compiler. In *Proceedings of the 1999 Workshop on Binary Translation*, New Port Beach, California, October 1999.

[20] Yoo C. Chung. Interpreter design for LaTTe. Available at `http://pallas.snu.ac.kr/~chungyc/papers/interpreter.ps`.

[21] Yoo C. Chung. Allocation with increments in a non-moving collector. Available at `http://pallas.snu.ac.kr/~chungyc/papers/fast-alloc.ps`.

[22] Dominique Colnet, Philippe Coucaud, and Olivier Zendra. Compiler support to customize the mark and sweep algorithm. In ISMM98 [32], pages 154–165.

[23] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[24] Yoo C. Chung, Junpyo Lee, Soo-Mook Moon, and Kemal Ebcioğlu. Memory management in the LaTTe Java virtual machine. In preparation.

[25] SPEC JVM98 Benchmarks. `http://www.spec.org/osg/jvm98/`.

[26] Dan Sahlin. Making garbage collection independent of the amount of garbage. Research Report R87008, Swedish Institute of Computer Science, 1987.

[27] H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[28] Hans-Juergen Boehm. Mark-sweep vs. copying collection and asymptotic complexity. Available at `http://www.hpl.hp.com/personal/Hans_Boehm/gc/complexity.html`.

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

[30] Henry G. Baker. The Treadmill: Real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.

[31] Paul R. Wilson and Mark S. Johnstone. Real-time non-copying garbage collection. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[32] *Proceedings of the International Symposium on Memory Management (ISMM '98)*, Vancouver, British Coumbia, Canada, October 1998. ACM Press.

Appendix

Tables 1, 2, and 3 tabulate the execution times, marking times, sweeping times, and numbers of collections for each given heap size, using the selected benchmarks from the SPECjvm98 benchmark suite that were run on top of the LaTTe Java virtual machine on a 270MHz UltraSPARC. The heap size, which is actually the size of the small object area as mentioned in section 5.1, is given in units of megabytes. The units for the times are in seconds.

Looking at the tables, one might notice that the marking times are larger when constructing the set of the live objects (i.e. doing selective sweeping) than when just marking the live objects (i.e. doing traditional sweeping). This could be considered a little odd, since the instruction sequences for constructing the set of live objects and just marking the live objects are nearly identical.

This is probably due to cache effects, since when constructing the set of live objects, the location at which an object is pushed in and the location from which the object to walk is popped can be quite a distance apart, unlike when marking with the traditional algorithm, in which the object that is pushed into the mark stack is more likely to be popped immediately for walking. Also, algorithm 3, which is used to construct the set of live objects, does a breadth-first traversal, which tends to have lower locality of reference than depth-first traversal, as done when marking with an explicit mark stack.

These differences may eventually be mitigated by using techniques such as prefetching or using a different algorithm to construct the set of live objects.

However, this contributes only a constant multiplicative factor to the marking time, so it does not change the fact that each individual garbage collection can be done in time independent of the heap size with selective sweeping, and hence the overall garbage collection time is still reduced by a much larger factor at high heap sizes when compared to traditional sweeping.

Table 4 shows the times for `_227_mtrt` using adaptive sweeping, with the threshold set to the given constant fractions of the heap size.

		Traditional			Selective		
Heap	Colls.	Exec.	Mark	Sweep	Exec.	Mark	Sweep
4	125	43.044	4.738	3.302	43.477	5.475	2.886
6	69	40.065	2.651	2.600	39.549	3.034	1.634
8	48	38.832	1.841	2.243	38.000	2.089	1.147
10	38	38.260	1.465	2.122	37.420	1.696	0.920
12	31	38.175	1.174	2.019	37.260	1.359	0.748
14	26	37.802	0.971	1.950	36.276	1.116	0.633
16	23	37.628	0.851	1.912	36.316	0.977	0.577
18	20	37.533	0.748	1.865	36.167	0.861	0.476
20	18	37.383	0.642	1.830	35.975	0.751	0.428
25	15	37.684	0.535	1.786	36.006	0.615	0.351
30	12	37.403	0.408	1.741	35.660	0.479	0.276
35	11	37.376	0.382	1.755	36.065	0.443	0.252
40	10	37.355	0.340	1.709	35.758	0.390	0.226
45	9	37.330	0.296	1.700	35.769	0.345	0.199
50	8	37.180	0.255	1.674	35.845	0.299	0.170
60	7	37.643	0.214	1.659	35.914	0.246	0.147
70	6	37.497	0.175	1.639	36.060	0.204	0.120

Table 1: Execution times for _202_jess.

		Traditional			Selective		
Heap	Colls.	Exec.	Mark	Sweep	Exec.	Mark	Sweep
10	79	53.952	10.411	6.097	76.079	16.084	22.765
12	36	44.420	4.385	3.149	53.341	6.664	9.503
14	25	42.457	2.805	2.383	48.520	4.324	6.140
16	19	41.216	2.082	2.010	44.977	3.184	4.523
18	16	40.373	1.715	1.830	43.206	2.630	3.724
20	14	39.881	1.424	1.687	42.110	2.167	3.109
22	12	39.504	1.538	1.538	41.093	1.716	2.447
24	11	40.010	1.035	1.497	40.988	1.604	2.272
26	10	40.106	0.828	1.396	40.562	1.265	1.768
28	9	39.292	0.751	1.359	40.740	1.151	1.577
30	8	39.317	0.678	1.320	40.137	1.028	1.425
35	7	38.703	0.466	1.220	39.376	0.697	0.942
40	6	39.437	0.386	1.174	39.786	0.578	0.792
45	6	39.744	0.407	1.179	39.499	0.582	0.790
50	5	39.420	0.306	1.144	39.280	0.473	0.670
55	5	39.258	0.305	1.137	38.548	0.463	0.671
60	4	38.032	0.095	1.028	37.238	0.136	0.169
65	4	40.077	0.141	1.051	39.457	0.206	0.246
70	4	40.111	0.161	1.062	39.474	0.238	0.320
75	4	40.191	0.158	1.062	39.581	0.237	0.314
80	4	40.354	0.160	1.062	39.719	0.237	0.315
90	3	40.444	0.086	1.024	39.585	0.124	0.154
100	3	40.747	0.085	1.024	39.913	0.124	0.154

Table 2: Execution times for _227_mtrt.

		Traditional			Selective		
Heap	Colls.	Exec.	Mark	Sweep	Exec.	Mark	Sweep
4	117	49.116	3.190	2.326	49.995	3.634	2.161
6	52	46.604	1.430	1.665	46.721	1.627	0.996
8	36	46.341	1.182	1.532	46.536	1.351	0.786
10	27	45.767	0.891	1.427	45.730	0.997	0.582
12	22	45.543	0.731	1.406	45.326	0.827	0.495
14	18	45.032	0.498	1.275	44.858	0.547	0.319
16	16	45.282	0.548	1.311	44.994	0.616	0.377
18	15	45.362	0.498	1.300	45.099	0.586	0.365
20	13	45.259	0.414	1.263	44.809	0.463	0.290
25	11	45.181	0.335	1.233	44.758	0.381	0.242
30	9	45.027	0.283	1.256	44.759	0.312	0.207
35	8	45.063	0.243	1.205	44.906	0.283	0.187
40	7	45.114	0.196	1.177	44.761	0.213	0.140
45	7	45.381	0.221	1.194	44.984	0.240	0.163
50	6	45.300	0.174	1.182	45.048	0.192	0.137
55	6	45.406	0.166	1.174	44.010	0.184	0.132
60	5	45.474	0.122	1.154	43.990	0.135	0.103

Table 3: Execution times for _228_jack.

			1/64			1/128			1/512	
Heap	Colls.	Exec.	Mark	Sweep	Exec.	Mark	Sweep	Exec.	Mark	Sweep
10	79	58.170	13.839	7.015	55.909	12.145	6.130	54.924	10.919	6.139
12	36	46.646	6.033	3.691	45.342	5.247	3.113	45.228	4.567	3.117
14	25	43.899	4.039	2.774	43.229	3.530	2.369	43.262	3.003	2.363
16	19	42.213	3.068	2.181	41.525	2.657	1.945	41.924	2.219	1.958
18	16	41.044	2.596	1.909	41.001	2.262	1.764	41.381	1.843	1.767
20	14	42.559	2.225	3.032	40.520	1.928	1.738	40.840	1.543	1.649
22	12	41.642	1.736	2.374	40.540	1.574	1.514	40.874	1.248	1.486
24	11	41.359	1.623	2.222	39.934	1.470	1.432	40.499	1.179	1.450
26	10	40.222	1.276	1.720	39.837	1.207	1.278	40.497	0.940	1.367
28	9	40.511	1.168	1.569	39.915	1.107	1.166	40.392	0.860	1.282
30	8	40.094	1.041	1.429	38.728	0.999	1.046	40.641	0.774	1.175
35	7	39.040	0.708	0.924	39.144	0.697	0.824	39.739	0.548	1.083
40	6	39.878	0.597	0.776	39.824	0.587	0.776	40.086	0.470	1.013
45	6	39.004	0.612	0.779	39.596	0.589	0.779	40.485	0.483	1.113
50	5	38.593	0.469	0.629	38.978	0.467	0.628	39.919	0.386	0.912

Table 4: Adaptive sweeping times for _227_mtrt.