

Limits of Parallel Marking Garbage Collection *

Dr. Fridtjof Siebert

aicas GmbH
Haid-und-Neu-Str. 18
76131 Karlsruhe, Germany
siebert@aicas.com

Abstract

More and more, parallel multicore systems will be used even in low-end devices such as embedded controllers that require real-time guarantees. When garbage collection is used in these systems, parallel or concurrent garbage collection brings important performance advantages. In the context of realtime systems, it has to be shown that a parallel garbage collector implementation not only performs well in most cases, but guarantees on its performance in the worst case are required.

This paper analyses the difficulties a parallel mark-and-sweep garbage collector faces during a parallel mark phase. The performance of such a garbage collector degrades when only some of the available processors can perform scanning work in the mark phase. Whenever the *grey* set contains fewer elements than the number of available processors, some processors may be stalled waiting for new objects to be added to the *grey* set. This paper gives an upper bound for the number of stalls that may occur as a function of simple properties of the memory graph.

This upper bound is then determined for the Java applications that are part of the SPECjvm98 benchmark suite and the theoretical worst-case scalability of a parallel mark phase is analysed. The presented approach is then applied to a Java virtual machine that has uniform mark steps, which at first results in poor worst-case scalability. A small change in the implementation is then proposed and analysed to achieve good scalability even in the worst case.

Categories and Subject Descriptors C.3 [Computer Systems Organization]: SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS—Real-time and embedded systems; D.3.4 [Software]: PROGRAMMING LANGUAGES—Processors: Memory Management (garbage collection); D.4.7 [Software]: OPERATING SYSTEMS—Organization and Design: Real-time systems and embedded systems

General Terms ALGORITHMS, LANGUAGES, PERFORMANCE, RELIABILITY

Keywords multicore, parallel, concurrent, realtime, garbage collection, Java

* This work was partially funded by the European Commission's 7th framework program's JEOPARD project, #216682.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISMM'08, June 7–8, 2008, Tucson, Arizona, USA.
Copyright © 2008 ACM 978-1-60558-134-7/08/06...\$5.00

1. Introduction

The emergence of multicore computer architectures will have a profound effect on the software development process and the implementation of programming languages. With multicore systems, parallel systems become the norm even for low-end computers such as embedded controllers. The use of languages or systems that perform their memory management using garbage collectors requires garbage collector implementations that make use of the parallel processing power provided by these systems.

The goal of such parallel garbage collector implementations is typically an increased average performance of the memory management system. Systems with realtime requirements, such as embedded controllers, however, need proven upper bounds on the worst case behaviour of the implementation. For a parallel garbage collector, this means that a bound on the worst-case performance has to be found. In particular, a limit has to be given on the time lost due to processors that are idle since the garbage collection work cannot be parallelised fully.

1.1 Terminology

The term *realtime* is often used in the memory management community in a way that is much more relaxed than its use in the realtime community. In the realtime community, a realtime system is a system that has hard deadlines that must not be missed. It is in this strict sense that realtime is understood within this paper: a realtime system needs proven upper bounds on its performance.

The literature categorises garbage collectors as incremental, concurrent or parallel. These categories are not disjoint, e.g., a parallel garbage collector may also be concurrent.

An *incremental* garbage collector is a garbage collector that can operate incrementally, i.e., it can perform a part of a full garbage collection cycle and allow the application to perform work in between the incremental garbage collection steps. A *concurrent* garbage collector can operate concurrently with the application. A concurrent garbage collector must therefore be incremental to an extreme extent such that only very simple primitive operations, such as compare-and-set, must be atomic with respect to the application. Finally, a *parallel* garbage collector is a garbage collector that can use several processors simultaneously to perform its work in parallel.

A parallel garbage collector may be non-concurrent, i.e., it may disallow the application to run concurrently with the garbage collector. Also, a concurrent garbage collector can be non-parallel, i.e., it could be restricted to a single processor that performs garbage collection work sequentially, but concurrently with the application.

This paper addresses parallel garbage collection. For a parallel realtime system, a parallel and concurrent garbage collector may be most useful for best application response times. In this paper, however, no assumptions are made whether the garbage collector is concurrent or not.

1.2 Parallel Mark and Sweep

A mark-and-sweep garbage collector has two distinct phases: The mark phase marks all reachable objects until no new objects can be marked, and the sweep phase collects unmarked objects and adds their memory to the free list. In real systems, a means to deal with fragmented memory needs to be implemented as well, either via measures that avoid fragmentation [26, 2] or via an explicit compaction phase [15, 1, 21, 24]. Dealing with fragmentation is a difficult task that is not addressed in this paper.

The sweep phase is relatively easy to parallelise [13]: Different processors can sweep different address ranges independently, but the details are important for the implementation [8]. This paper will focus only on the parallel mark phase.

During the mark phase, the allocated objects are partitioned into three disjoint sets: unmarked (*white*), reachable but unscanned (*grey*) and reachable and scanned (*black*). The mark phase starts with all objects reachable from root references in the *grey* set. A single mark step then consists of taking one object b from the *grey* set, moving all objects that are in the *white* and that are directly reachable from b into the *grey* set and moving b itself to the set of *black* objects. The mark phase terminates when the set of *grey* objects becomes empty.

In a concurrent or incremental mark-and-sweep collector, a write barrier is used to ensure that modifications made to the memory graph by the application will not cause the accidental reclamation of the memory of objects that are still reachable.

In a parallel garbage collector, several processors will perform mark steps in parallel. Parallel mark steps should operate on different *grey* objects. To simplify the distinction between *grey* objects that are currently being scanned by one processor and those that are not, I introduce a subset of *grey*: the set of *anthracite* objects contains all objects that are currently being scanned by a processor performing a mark step.

A stall occurs whenever one processor is blocked during the mark phase waiting for work, i.e., when the set $grey \setminus anthracite$ is empty, i.e., there are not enough *grey* objects available for all available processors to perform mark steps in parallel.

With this, we can describe the actions to be performed in a parallel mark step in more detail as shown in Figure 1. The mark phase starts with all objects reachable from roots being in the *grey* set and all other allocated objects being *white*. The mark steps are executed repeatedly and in parallel until the *grey* set becomes empty. Then, the sweep phase can reclaim the memory of all objects that are still in the *white* set.

For a parallel GC to be efficient, the number of stalls during the mark phase must be low. For a system that requires predictable timing of the parallel GC work, we must even be able to find a hard upper bound for the number of stalls that may occur during the mark phase. Such an upper bound will be presented in section 2.

1.3 Limitations and assumptions

In this paper, the analysis is limited to finding bounds on the number of stalls that occur due to the set of *grey* objects having fewer elements than the number of processors that can perform parallel mark steps.

A parallel mark-and-sweep garbage collector implementation has to deal with other potential performance bottlenecks as well. In particular, parallel accesses to the sets of *white*, *grey*, and *black* objects must be possible and the amount of contention incurred by parallel modifications of these sets must be limited.

Limiting the contention due to the set implementation or finding a good implementation of these sets is not addressed in this paper. The area of implementations of these sets for parallel garbage collectors and how these sets can be shared between processors without incurring too much contention has been addressed by other

```

01: while  $grey \setminus anthracite = \emptyset \wedge grey \neq \emptyset$ 
02:   {
03:     stall();
04:   }
05: if  $grey = \emptyset$ 
06:   {
07:     start sweep phase
08:   }
09: else
10:  {
11:    choose  $b$  such that  $b \in grey \setminus anthracite$ 
12:     $anthracite := anthracite \cup \{b\}$ 
13:    for all  $c$  directly reachable from  $b$ 
14:      {
15:        if  $c \in white$ 
16:          {
17:             $grey := grey \cup \{c\}$ 
18:             $white := white \setminus \{c\}$ 
19:          }
20:      }
21:     $black := black \cup \{b\}$ 
22:     $grey := grey \setminus \{b\}$ 
23:     $anthracite := anthracite \setminus \{b\}$ 
24:  }

```

Figure 1. The simplified code that needs to be performed by a parallel mark step. The required synchronisation is omitted for brevity. A correct implementation requires synchronisation mechanisms to ensure that intermediate states do not become visible to other processors.

publications [13, 15, 22, 24]. Instead, this paper only analyses when a processor might be idle since the memory graph can be scanned in an order that causes the *grey* set to contain fewer elements than the number of available processors.

2. Limiting stalls in mark phase

The number of stalls depends on the memory graph. Imagine a graph that consists only of a singly-linked list of objects as illustrated in Figure 2. In this case, a parallel mark phase may have no advantage over a non-parallel one since after each mark step, there is only one element that has been added to the *grey* set, such that only one processor may continue work, and all others will stall.

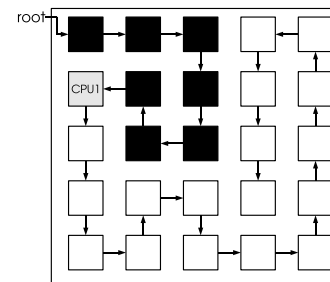


Figure 2. A simple memory graph that causes all but one processor to stall during mark phase.

The goal of these thoughts is to find measures on the memory graph that enable finding an upper limit for the number of stalls that show that with a certain memory graph, a parallel garbage collector can guarantee a performance that is better than that of a sequential GC.

2.1 Sequential Case

On a system with only a single processor performing mark steps, no stalls will occur since the set of *anthracite* objects is empty before and after each step shown in Figure 1. The described parallel mark step algorithm will consequently use all available processor time in the case of a single processor.

2.2 Two Processors

As shown with the example of a memory graph consisting of a singly-linked list in Figure 2, stalls may occur when two processors attempt to perform mark steps. On a heap with A reachable objects, the number of stalls that may occur is therefore limited by A on a two processor system: whenever one processor is performing a mark step on one reachable object, the other may be stalled waiting for an element to be added to the *grey* set such that the set $grey \setminus anthracite$ becomes non-empty.

In contrast, if the heap consists of a structure that is branched, full parallel mark may be possible. E.g., For a heap that consists of two singly-linked list of the same length, the mark may run fully in parallel with no stalls at all if both lists are traversed in parallel as shown in Figure 3.

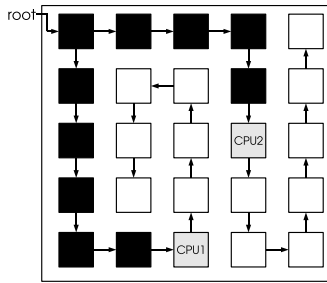


Figure 3. A simple memory graph with two linked lists that permits a fully parallel mark phase on a two processor system.

In the case of two singly-linked lists, however, full parallel mark is not guaranteed. Instead, it may happen that one of the lists will be traversed and scanned fully before the second list will be traversed¹. Then, scanning of the remaining list may cause one processor to stall while the other one performs a mark step causing the number of stalls to reach the length of this list, as illustrated in Figure 4.

2.2.1 Definitions

To simplify the following discussions, I will introduce some definitions:

Definition: A heap graph $H = (B, E, r)$ is a set of objects B with directed links $E \subseteq B \times B$ and a root object $r \in B$.

For this paper, I assume that there is only a single root object to simplify the argument. Using only a single root object makes a root scanning phase trivial, but it adds some overhead to the application that has to ensure that root references are also present on the heap [27]. The results can easily be generalised for arbitrary numbers of roots if needed.

Definition: A root path in a heap graph $H = (B, E, r)$ is a path $w = (b_0, b_1, b_2, \dots, b_n)$ with $b_0 = r$ and $(b_i, b_{i+1}) \in E$. The set of root paths in H is $W(H)$.

Hence, a root path is any path that starts in the root object and follows a chain of objects that are connected via references.

¹ this might happen if one processor is busy with a different task such that only one processor performs mark first. Since the *grey* set will always contain one element of each list, the processor may pick only elements of one list leaving the other one untouched until the first list is fully scanned.

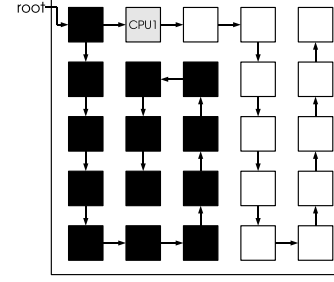


Figure 4. The same memory graph from Figure 3, in a case in which the mark phase progressed such that, on a two processor system, one processor may stall on every remaining mark step. This situation may occur, e.g., in a concurrent GC if first only one processor performs GC work and marks the first list, but later two processors are available for the remaining GC work.

Definition: The $length(w)$ of a root path $w = (b_0, b_1, b_2, \dots, b_n)$ is the number of edges along this path: n .

Definition: For a heap graph $H = (B, E, r)$, an object $b \in B$ is called reachable if b is part of any root path in H . b is called unreachable if it is not reachable. The set of reachable objects is $R(H)$:

$$R(H = (B, E, r)) := \{b \in B \mid \exists_{w \in W(H)} b \in w\} \quad (1)$$

Any reachable object will eventually be marked during the mark phase of the garbage collector.

Definition: The $depth(b, H)$ of a reachable object $b \in B$ in a heap graph $H = (B, E, r)$ is the length of the shortest root path in H that ends in b .

The depth of an object is hence the shortest chain of objects when starting with the root object and following references until the object is reached.

Definition: A stall is the situation in which the set of *white* objects contains reachable objects and the set of *grey* objects contains fewer elements than the number of processors p .

$$stall \iff (white \cap R(H)) \neq \emptyset \wedge |grey| < n \quad (2)$$

A stall hence occurs when a processor's request for a *grey* object may not be fulfilled immediately even though there are reachable objects that are still *white*. The situation at the end of the mark phase when the *grey* list becomes empty due to the fact that there are no remaining reachable objects that have not been found yet is not considered to be a stall by this definition.

2.2.2 Upper Bound for Two-Processor Mark Phase

With the definitions from the previous section, I can now formulate a first theorem that will be needed to define the upper bound for the number of stalls in a two processor mark phase:

Theorem T1: During a parallel mark phase on a two processor system, the n th stall may occur only after all objects with $depth(b) \leq n$ have been scanned and marked *black*.

Proof: After the first stall, it is ensured that all objects b with $depth(b) = 1$ are scanned. If an object b with $depth(b) = 1$ was not scanned, the stalling processor would scan it since b is directly reachable from the root and consequently *grey*.

For any n th stall with $n > 0$, we can assume that all objects with $depth(b) < n$ have been scanned. At the n th stall, all objects with $depth(b) \leq n$ will have been scanned or are currently being scanned. If any object with $depth(b) = n$ was not scanned already, b must be *grey* since all objects with $depth(b) < n$ have been scanned. Consequently, the *grey* set would not be empty and the processor would not stall. \square

With the proven theorem T1, we can now prove the following theorem T2, which gives us a first upper bound for the number of stalls:

Theorem T2: The number of stalls s during a parallel mark phase on a two processor system for a heap graph H is limited by the length of the longest root path in H , i.e.,

$$s \leq \max_{w \in W(H)} \text{length}(w) \quad (3)$$

Proof: Assume the longest root path is w with $\text{length}(w) = l$ and there were l stalls already. At the l th stall, all objects with $\text{depth}(b) \leq l$ will have been scanned or are currently being scanned.

Now, at the $l + 1$ th stall, we are sure that all objects with $\text{depth}(b) \leq l$ will have been scanned. If the mark phase is not completed yet, there must be an unscanned object b with $\text{depth}(b) > l$, so there must be a path w' with $\text{length}(w') > \text{length}(w)$ in contradiction to our assumption that w is the longest path. \square

With this proof, we can become even more precise and say:

Theorem T3: The number of stalls s during parallel mark phase on a two processor system for a heap graph $H = (B, E, r)$ is limited by the maximum $\text{depth}(b, H)$ for any reachable object b in H , i.e.,

$$s \leq \max_{b \in R(H)} \text{depth}(b, H) \quad (4)$$

describes the maximum number of stalls.

2.3 Several Processors

On a system with an arbitrary number of processors, we may see a larger number of stalls during the mark phase. In particular, in the worst case of the heap consisting of a singly-linked list of objects as in Figure 2, only one processor may make progress while all others are stalled. So, an obvious upper limit for the number of stalls s during parallel mark on a system with p processors depends on the number of reachable objects:

$$s \leq (p - 1) * |R(H)| \quad (5)$$

This limit is, however, not satisfactory and we need to find a better estimate. To find this better limit, we start with a generalisation of theorem T1 as follows.

Theorem T1': During parallel mark phase on a p processor system, the n th stall may occur only after all objects with $(p - 1) \cdot \text{depth}(b) \leq n$ have been scanned and marked *black*.

Proof: After the first stall, it is ensured that all objects b with $\text{depth}(b) = 1$ are scanned. If an object b with $\text{depth}(b) = 1$ was not scanned, the stalling processor would scan it since b is directly reachable from the root and consequently *grey*. At this stall, at most $p - 1$ processors may stall since at least one processor will make progress. Otherwise the *grey* set would be empty and the mark phase would be finished.

For every $(p - 1) \cdot n$ th stall, we can assume that all objects with $\text{depth}(b) < n$ have been scanned. At the $(p - 1) \cdot n$ th stall, all objects with $\text{depth}(b) \leq n$ will be scanned. If any object with $\text{depth}(b) = n$ would not be scanned, b would have been *grey* since all objects with $\text{depth}(b) < n$ have been scanned. Consequently, the *grey* set would not be empty and the processor would not stall. \square

The generalised T1' enables us to generalise T2, which gives us a first upper bound for the number of stalls:

Theorem T2': The number of stalls s during a parallel mark phase on a p processor system for a heap graph H is limited by the product of $p - 1$ and the length of the longest root path in H , i.e.,

$$s \leq \max_{w \in W(H)} (p - 1) \cdot \text{length}(w) \quad (6)$$

holds.

Proof: Assume the longest root path is w with $\text{length}(w) = l$ and there were l stalls already. At the l th stall, all objects with $(p - 1) \cdot \text{depth}(b) \leq l$ will have been scanned or are currently being scanned. Now, at the $l + 1$ th stall, we are sure that all objects with $(p - 1) \cdot \text{depth}(b) \leq (p - 1) \cdot l$ will have been scanned. If the mark phase is not completed yet, there must be an unscanned object b with $(p - 1) \cdot \text{depth}(b) > (p - 1) \cdot l$, so there must be a path w' with $\text{length}(w') > \text{length}(w)$ in contradiction to our assumption that w is the longest path. \square

With this proof, we can become even more precise and generalise theorem T3:

Theorem T3': The number of stalls s during a parallel mark phase on a p processor system for a heap graph H is limited by the maximum of $(p - 1) \cdot \text{depth}(b, H)$ for any reachable object b in H , i.e.,

$$s \leq (p - 1) \cdot \max_{b \in R(H)} \text{depth}(b, H) \quad (7)$$

is true.

3. Concurrent Mark Phase

The previous section defined upper limits for the number of stalls in a parallel mark phase. This section now will look at the effects a concurrent execution of the parallel mark phase may have on these results.

A concurrent garbage collector needs some means to make sure that modifications that are made to the memory graph by the application will never cause the garbage collector to detect any object as garbage that is still reachable by the application. The means to ensure this is the use of a write barrier. Such a write barrier typically moves objects from the *white* set to the *grey* set if these objects could potentially be missed by the garbage collector as a consequence of the modification of the memory graph that is made by the application.

There are two possible write barrier techniques, incremental-update algorithms and snapshot-at-beginning algorithms [23, 30]. For a parallel, concurrent garbage collector, a snapshot-at-beginning algorithm is typically employed since it is inherently hard to implement an efficient incremental-update write barrier on a parallel system.

A snapshot-at-beginning write barrier ensures that no paths to a *white* object will be destroyed. This means, that on any pointer assignment, the object pointed to by the overwritten reference, will be added to the *grey* set if it exists and was *white*.

For the parallel mark phase, the snapshot-at-beginning write barrier means that basically the heap graph at the beginning of the mark phase will have to be marked, with three differences: erased referenced to *white* objects will be moved to the *grey* set earlier; new references may be added to the heap graph; and newly allocated objects will be added to the heap graph.

For erased references, adding *white* objects to the *grey* set earlier will not introduce additional stalls during the mark phase. These objects can be regarded as if they would be referenced by the last scanned object (or the root if the *black* set is empty). Therefore, the maximum depth of objects does not increase for erased references.

Newly added links in the heap graph will not increase the depth of any objects, so these new links will also not have an impact on the worst case number of stalls. It should, however, be noted that new references stored in already scanned objects will not be seen by the garbage collector, while references stored in *white* objects may cause the referenced objects to be added to the *grey* set earlier².

²References that are added to a *grey* object b that is currently being scanned by another processor may or may not get added to the *grey* set

Finally, references to newly allocated objects can increase the maximum depth of the heap graph during the concurrent mark phase. If, however, an algorithm is used that “allocates black”, i.e., adds newly allocated objects to the *black* set, these do not need to be treated by the mark phase and hence will have no effect on the execution of the mark phase.

In summary, the limits on the number of stalls found in section 2 still hold for a snapshot-at-beginning algorithm that allocates objects as *black*.

4. Analysis and Measurements

With the theoretical limit on the number of stalls found in theorem T3', it is now of interest to see what statements one can make on the performance of parallel garbage collection on real applications.

4.1 Approach

To obtain representative measurements of maximum depths of the heap graph for real applications, these depths were measured using several applications. The applications were taken from the SPECjvm98 benchmark suite [28].

The JamaicaVM Java virtual machine [19] was instrumented to measure the maximum depth of any object in the memory graph. The maximum depth can change only when a modification is made to the graph via a pointer store operation (including storing of *null*, i.e. deleting an existing reference). However, modifications to the graph by the applications are very frequent, such that it becomes infeasible to re-scan the heap after each store.

Instead, the maximum depth was re-evaluated regularly after a fixed number of reference stores³. The number of stores in between the evaluation of the depths was set to 10000 in the benchmarks performed. Using these samples causes some imprecision since the maximum depth might only occur in between two measurements. However, the measured results were very stable, such that it appears very unlikely that there is a significant error in these measurements. Increasing the frequency of the measurements would, of course, increase the precision. However, the measurements presented here took 24 hours for one run, so a higher frequency quickly becomes infeasible.

To obtain the results presented below, the maximum of all measurements was taken. For the depths, the absolute maximum of all measured values was taken. For the relative values, the maximum of the ratio between depth and reachable memory was taken. The maximum ratio is not necessarily at the same measurement as the maximum depth since the amount of reachable memory changes over time⁴.

As stated before, a realtime system requires a proof that timing constraints put on the system are respected. The use of measurements is usually not an adequate means to prove any feature of a realtime system. The measurements given here are therefore made only to give an idea about the features of the memory graph of different applications and the effect on parallel garbage collection. For a realtime application that relies on the worst-case behaviour of the parallel garbage collector, more trustworthy evidence⁵ is required

earlier depending on whether or not the store is to a part of the *b* that has already been scanned.

³ executions of Java bytecodes *putstatic*, *putfield* or *aarraystore* if the stored value was a reference.

⁴ in fact, during these measurements, the maximum depth was achieved in a large number of measurements, and the maximum ratio was always one of the measurements with maximum depth that had the lowest amount of reachable memory.

⁵ Such evidence could be a proof based on the code or a measurement based on data that can be proven to provoke the worst-case heap graph depth.

to ensure that the heap graph's depth is limited as required to meet the garbage collector's timing requirements of the system.

4.2 Java Object Graph Depth

First, the maximum depth of the heap graph when regarding the references between whole Java objects was measured. The maximum depths that were encountered are illustrated in Figure 5.

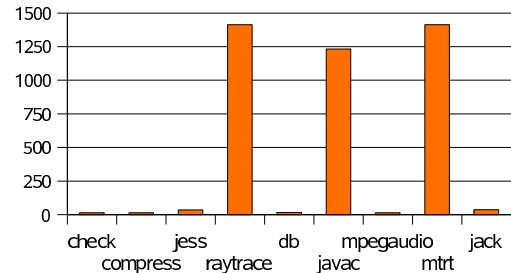


Figure 5. Maximum depths at the granularity of whole Java objects.

Most applications in this benchmark suite have a very shallow heap graph with a maximum depth of less than 40 objects. Only three applications, raytrace, javac and mrt, have a significantly larger maximum depth.

For the overall performance of the parallel mark phase, the ratio of maximum depth to number of reachable objects is important. These maximum relative depths are shown in Figure 6. In all tests, this ratio is less than 4%. Consequently, stalls will occur on less than 4% of the objects that are marked.

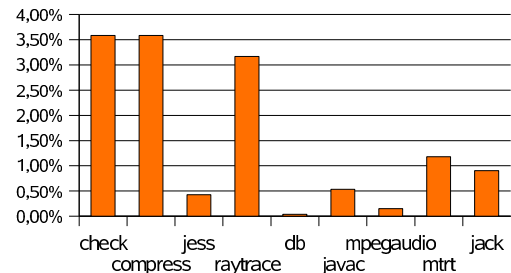


Figure 6. Maximum relative depths at the granularity of Java objects: maximum depth divided by number of reachable objects.

With the maximum depth we can now determine the maximum number of stalls for different number of processors and hence calculate the worst-case mark phase performance on a parallel system. The results are shown in Figure 7. The displayed values are the maximum expected speedup that can be expected for a given number of processors and the minimum fraction of the available CPU time that will be used. These first results are very promising: all applications scale well up to 32 processors, most see a significant performance increase even up to 256 processors. The mark phase will use more than 90% of the available CPU time for all tests on systems with up to 4 processors, more than 80% of the CPU time will be used on systems with up to 8 processors.

4.3 Uniform Mark Steps

The scalability results presented in the previous section were based on the assumption that the effort to perform a mark step is largely independent of the kind of object that is being scanned. Unfortunately, this is not true in the general case: Even though Java objects are small on average [6], Java applications usually have a

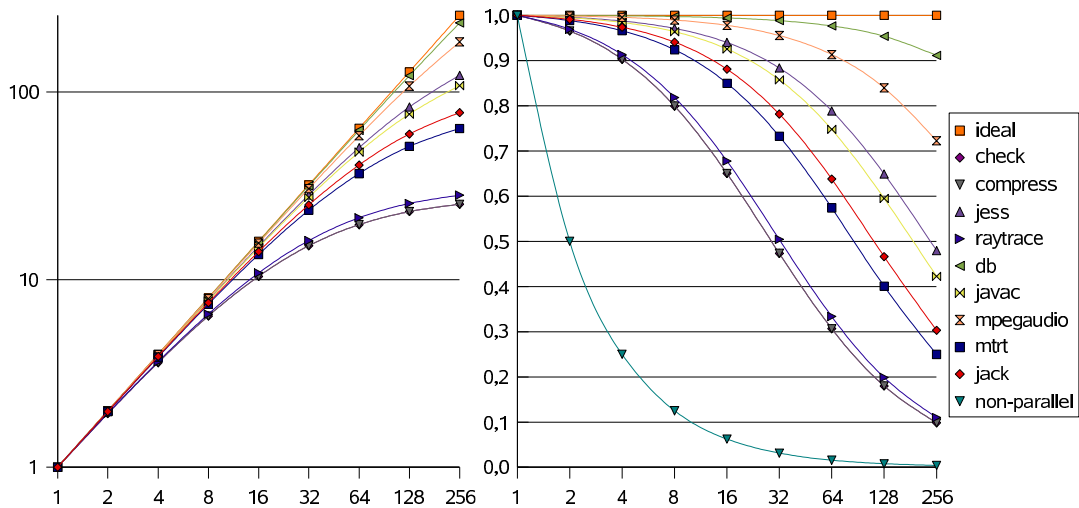


Figure 7. The expected worst-case scalability of the mark phase using the depth measured for Java objects for systems with 1 through 256 processors available for garbage collection. The left graph shows the expected performance increase (speedup of mark phase compared to a non-parallel system) as a function of the number of available processors. The right graph illustrates the expected CPU utilisation (performance divided by number of available processors). The curves *ideal* and *non-parallel* are added to illustrate the extreme cases of full parallel and purely sequential execution.

large range of different object sizes ranging up to large arrays. This means that a mark step on such a large array will require significantly longer than a mark step on an average object. Consequently, if processors can stall during such an expensive mark step, then the processing power lost due to stalls can become significantly higher.

One solution is a garbage collector that has more uniform mark steps. The garbage collector in the JamaicaVM attains such uniform mark steps by treating the heap as an array of fixed size blocks and composing all Java objects out of these blocks [26]. Consequently, the garbage collector no longer works on Java objects directly, but instead traverses the graph of fixed size blocks. The fixed size for blocks used by the analysed virtual machine is 32 bytes.

Since Java objects in JamaicaVM are constructed from a graph of fixed size blocks, the maximum depth in the memory graph seen by the garbage collector is different from the maximum depth when regarding whole Java objects only. Consequently, the measurements presented in Figures 5 through 7 have been redone on the graph of single fixed size blocks.

The resulting absolute and relative depths are shown in Figures 8 and 9. For some applications, these depths are significantly larger than the values found when looking at the Java object graph only. The reason for these larger values lies mainly in the fact that arrays with a contiguous layout in memory are scanned block by block by the garbage collector as if the array was a long singly-linked list⁶: each block in such an array has an implicit reference to the next block in the array.

The worst-case scalability of the mark phase when uniform blocks are scanned is not very good (Figure 10). The results are much worse in this case: one test, *mpegaudio*, stays below a factor 2 independent of the number of processors. For all other tests, the performance gain with more than 16 processors is diminishing. In *mpegaudio*, the presence of a large array prevents scalability. For the worst-case, it has to be assumed that this array may need

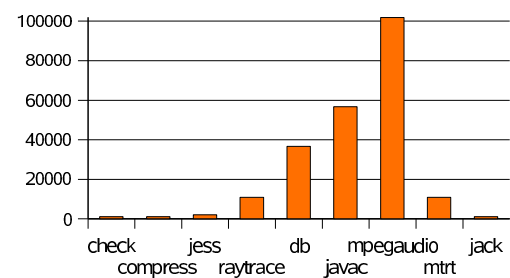


Figure 8. Maximum depths at the granularity of 32 byte blocks used by JamaicaVM.

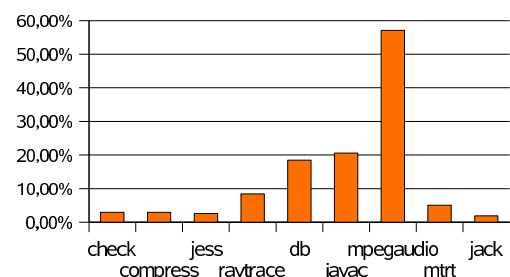


Figure 9. Maximum relative depths at the granularity of 32 byte blocks used by JamaicaVM: maximum depth divided by number of reachable blocks.

⁶The Java application does not access arrays as singly linked lists, though. Instead, an array that is contiguous in memory is accessed directly, while an array that uses fragmented memory has a tree-structure that enables efficient access logarithmic in the length of the array.

to be scanned sequentially by a single processor. Even on a 256 processor parallel mark phase, only a 75% speedup compared to a single processor system can be guaranteed.

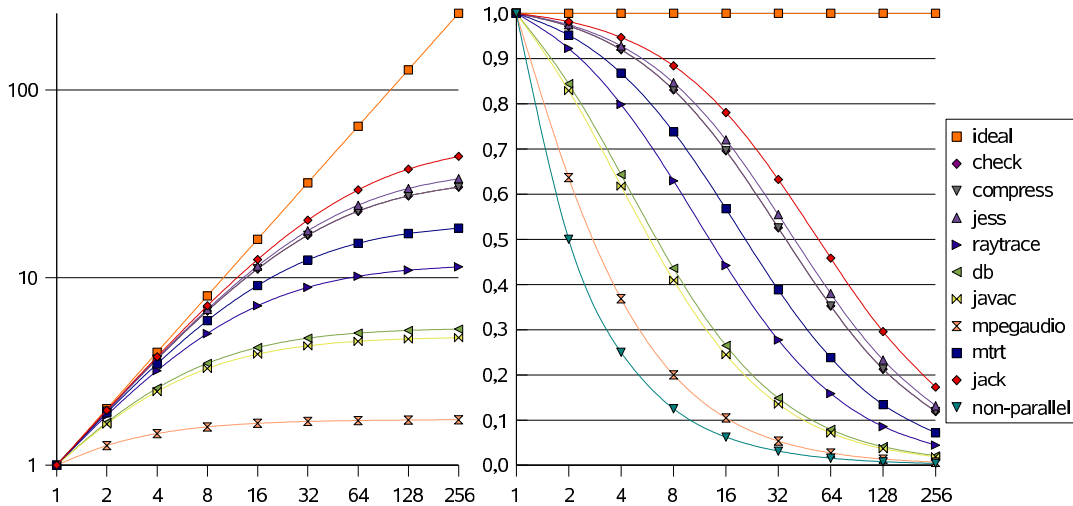


Figure 10. The same graphs as in Figure 7 for the expected worst-case scalability of the mark phase using the depth measured for 32 byte blocks.

4.4 Enhancing Uniform Mark Steps for Parallel Systems

The reason for the poor scalability shown in the previous section lies in the fact that contiguous arrays are essentially treated as singly-linked lists of blocks, which causes a high maximum depth and results in poor scalability.

To improve this behaviour, the mark code was changed such that a mark step that scans the first block in a contiguous array marks several blocks in the array as *grey*. For the measurements presented here, the array was split into eight sub-arrays, such that for an array that occupies n blocks, blocks number 1, $n/8$, $n * 2/8$, $n * 3/8$, ..., $n * 7/8$ were added to the *grey* list. This ensures that the subsequent marking of the array elements could be performed in parallel on up to eight processors.

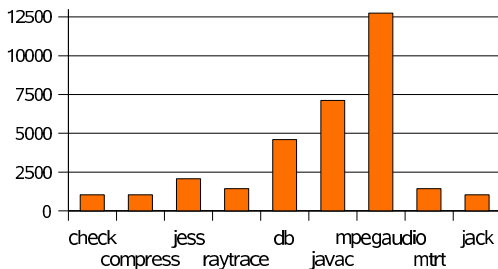


Figure 11. Maximum depth at the granularity of 32 byte blocks used by JamaicaVM with modified mark for contiguous arrays.

The resulting maximum depth is shown in Figures 11 and 12. The resulting scalability has improved significantly (Figure 13): Now, all applications except mpegaudio scale well up to about 64 processors. Only mpegaudio, which is dominated by one large array, scales well only up to about 16 processors. The bad scalability for mpegaudio with more processors is due to the fact that this large array is split up into only 8 parts that can be processed in parallel. A better scalability for a larger number of processors could be achieved by marking more inner blocks within sub-arrays of a large array when scanning the first block of such an array. However, this would come at the cost of a higher worst-case execution time of the mark step that handles the array header block.

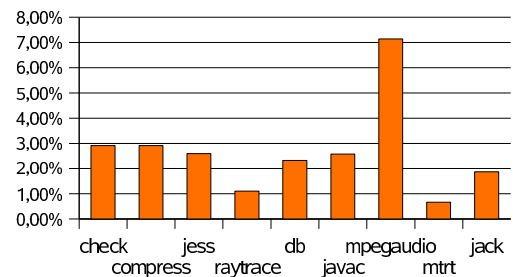


Figure 12. Relative maximum depth at the granularity of 32 byte blocks used by JamaicaVM with modified mark for contiguous arrays: maximum depth divided by number of reachable blocks.

5. Related Work

Endo, Taura and Yonezawa [14] presented an approach to predict the scalability of parallel garbage collection in the average case. Their approach takes the memory graph and specifics of the hardware such as cache misses into account. In contrast to this work, their result is an estimate of the scalability, while this paper presents an upper bound for the worst-case scalability.

Early work on concurrent and incremental mark and sweep garbage collection dates back to the seventies with important contributions from Dijkstra [7] and Steele [29]. The first incremental copying collector was presented by Baker [3].

One of the earliest implementations of an incremental and parallel garbage collector is the Multilisp implementation by Halstead [16]. This is a parallel version of Baker's copying garbage collector that uses processor-local oldspaces and newspaces to enable parallel garbage collection work and parallel memory allocation. However, this approach did not address the problem of load balancing at all, while I assume perfect load balancing in this paper, i.e., as long as the *grey* set is non-empty, any processor could obtain an element from this set to perform mark phase work. Load balancing between different mutator processors was proposed by Endo [13]. In Endo's approach, each process maintains a local subset of the *grey* set. These local subsets are divided into two subsets: a local

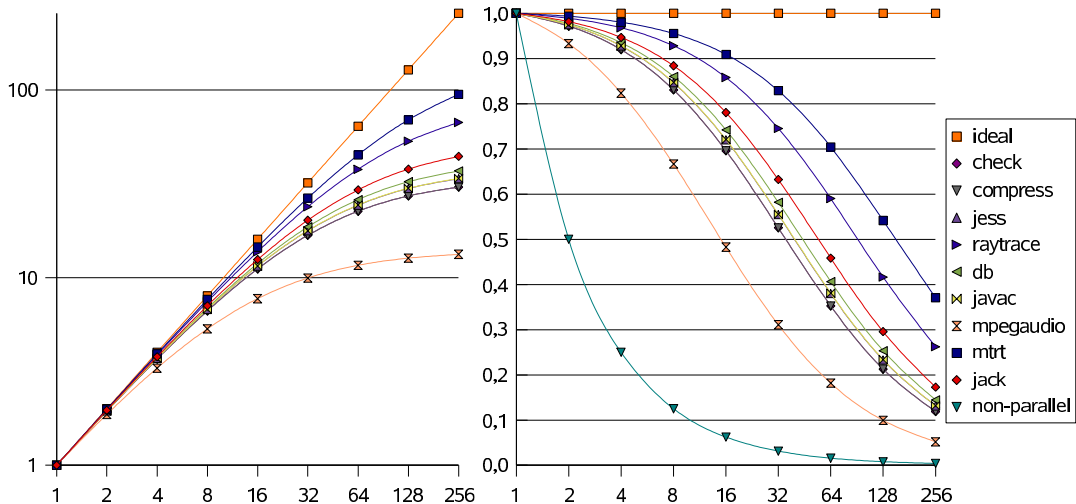


Figure 13. The same graphs as in Figure 7 for the expected worst-case scalability of the mark phase using the depth measured for Jamaica blocks with modified mark for contiguous arrays. The scalability has improved significantly, all applications see an important performance gain up to 32 CPUs, some even up to 256 CPUs.

mark stack and a stealable mark set. When a processor would stall due to empty local mark sets, it will attempt to steal *grey* objects from another processor's stealable mark set.

Blelloch and Cheng have presented a theoretical bound on time and space for parallel garbage collection [4] and refined their approach [5] to become practically implementable by removing the fine granularity of the original algorithm and adding support for stack scanning, etc. The original scanning of fields one at a time prevented parallel execution. Their new approach scans one object at a time, resulting in parallel scanning of objects. Since scanning of large objects would prevent parallelism as presented in sections 4.2 and 4.3, the authors split up larger objects into segments that can be scanned in parallel, a technique that is similar to marking inner blocks of a large contiguous array *grey* to improve the parallelism as presented in section 4.4.

A parallel, non-concurrent garbage collector with load balancing via *work stealing* was presented by Flood et al. [15]. Each processor has a fixed size work-stealing queue. If a processor's work-stealing queue overflows, part of its content is dumped to a global overflow set. Processors with an empty local queue first try to obtain work from the overflow set before they resort to stealing. This technique was then applied to parallel copying and mark-compact garbage collectors resulting in a speedup factor between 4 and 5 on an 8 processor machine.

The parallel, incremental and concurrent GC presented by Ossia et al [22] employs a low-overhead work packet mechanism to ensure load balancing for parallel marking. In contrast to previous balancing work, all processors compete fairly for the marking work, i.e., there is no preference for a processor to first work on the work packets it generated.

Other fully concurrent on-the-fly mark-and-sweep collectors have been presented by Doligez and Leroy [10], Doligez and Gonthier [9], and Domani et al. [11, 12]. For all of these parallel collectors, the worst-case scalability presented in this paper can be applied and provides a worst-case bound on the scalability.

This paper did not address the problem of implementing the *grey* set such that parallel accesses are possible. In addition to the different load balancing techniques presented in this section [13, 15, 22], one possible implementation is to reserve one word per

object such that all *grey* objects could be stored in a linked list [25, 24].

A different approach to parallel garbage collection was presented by Huelsbergen and Winterbottom [18]. Their approach is basically a concurrent mark-and-sweep garbage collector in which the mark and the sweep phases run in parallel. However, the mark phase itself is not parallel in this approach, so it does not suffer from the stalls discussed in this paper.

Being able to give an upper bound of the scalability of the garbage collector enables one to give an upper bound on the total work required to perform one garbage collection cycle. This work can then be used to schedule the garbage collector such that it reclaims memory fast enough to satisfy all allocation requests. There are two main approaches to schedule the garbage collector work: work based scheduling [3], or approaches based on the allocation rate of tasks in the system [17].

6. Conclusion

With the theoretical upper bound for the number of stalls in a parallel garbage collection mark phase, an upper bound for the worst-case scalability of a parallel garbage collector's mark phase has been found. For the analysed benchmarks, this upper bound promises relatively good scalability provided that the cost of marking an object is equal for all objects.

A garbage collector that operates on fixed-size blocks and therefore has a more uniform mark step can show a significantly worse worst-case scalability when no measures are taken to ensure that large arrays can be scanned in parallel. When such measures are taken, the worst-case scalability can be improved: All benchmarks analysed show a significantly better worst-case scalability for up to 32 processors, most tests even scale well up to more than 100 processors.

The presented upper bound on the worst-case performance of a parallel marking garbage collector enables the prediction of the worst-case garbage collector behaviour on parallel systems. This is a first required step for hard realtime garbage collection on parallel systems that require good upper bounds on the time required to complete a garbage collection cycle. With these bounds available, a parallel and concurrent garbage collector can be scheduled

such that it is ensured that a garbage collection cycle completes and recycles enough memory to satisfy ongoing allocation requests. If a bounded worst-case behaviour can be found for all aspects of a parallel garbage collector, i.e., also for root scanning, load balancing, sweeping and compaction mechanisms, it becomes possible to employ hard realtime garbage collection in parallel system with correctness guarantees that are not only measured, but also proven.

7. Future Work

A comparison of the theoretical number of stalls with the actual number of stalls seen running in a real system would be an interesting next step that we at aicas intend to perform when our implementation of a parallel realtime garbage collector is available. The effects on other implementation decisions, such as the implementation of grey sets, load balancing, etc., should then also be analysed, both by measuring actual performance and by finding theoretical bounds for the worst-case performance.

References

- [1] Diab Abuaiadh, Yoav Ossia, Erez Petrank, and Uri Silbershtein. An efficient parallel heap compaction algorithm. *SIGPLAN Not.*, 39(10):224–236, 2004.
- [2] David F. Bacon, Perry Cheng, and V.T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'2003)*, San Diego, CA, June 2003. ACM Press.
- [3] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.
- [4] Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 104–117, Atlanta, May 1999. ACM Press.
- [5] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 125–136, New York, NY, USA, 2001. ACM.
- [6] Sylvia Dieckman and Urs Hölzle. A study of the allocation behaviour of the SPECjvm98 Java benchmarks. In Erik Jul, editor, *Proceedings of 12th European Conference on Object-Oriented Programming, ECOOP98*, volume 1445 of *Lecture Notes in Computer Science*, pages 92–115, Brussels, July 1998. Springer-Verlag.
- [7] Edsger 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, New York, 1976.
- [8] Robert Dimpsey, Rajiv Arora, and Kean Kuiper. Java server performance: A case study of building efficient, scalable JVMs. *IBM Systems Journal*, 39(1):151–174, 2000.
- [9] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 1994.
- [10] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 113–123. ACM Press, January 1993.
- [11] Tamar Domani, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Vancouver, June 2000. ACM Press.
- [12] Tamar Domani, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. Technical Report 88.385, IBM Haifa Research Laboratory, 2000. Fuller version of [11].
- [13] Toshio Endo. A scalable mark-sweep garbage collector on large-scale shared-memory machines. Master's thesis, University of Tokyo, February 1998.
- [14] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. Predicting scalability of parallel garbage collectors on shared memory multiprocessors. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01)*, San Francisco, CA, pages 43–43. IEEE Computer Society, 2001.
- [15] Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, April 2001.
- [16] Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.
- [17] Roger Henriksson. Scheduling real-time garbage collection. In *Proceedings of NWPEN'94*, Lund, Sweden, 1994.
- [18] Lorenz Huelshbergen and Phil Winterbottom. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. In Jones [20], pages 166–175.
- [19] Jamaica Virtual Machine, aicas GmbH, Karlsruhe, Germany. www.aicas.com/jamaica, 1999-2008.
- [20] Richard Jones, editor. *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, Vancouver, October 1998. ACM Press.
- [21] Haim Kermany and Erez Petrank. The compressor: concurrent, incremental, and parallel compaction. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 354–363, New York, NY, USA, 2006. ACM.
- [22] Yoav Ossia, Ori Ben-Yitzhak, Irit Gofit, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 129–140, Berlin, June 2002. ACM Press.
- [23] Pekka P. Pirinen. Barrier techniques for incremental tracing. In Jones [20], pages 20–25.
- [24] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 159–172, New York, NY, USA, 2007. ACM.
- [25] Fridtjof Siebert. Hard real-time garbage collection in the Jamaica Virtual Machine. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, 1999.
- [26] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Compilers, Architectures and Synthesis for Embedded Systems (CASES2000)*, San Jose, November 2000.
- [27] Fridtjof Siebert. Constant-time root scanning for deterministic garbage collection. In *Tenth International Conference on Compiler Construction (CC2001)*, Genoa, April 2001.
- [28] Standard Performance Evaluation Corporation (SPEC). *SPECjvm98 Benchmarks*. <http://www.specbench.org/osg/jvm98/>.
- [29] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.
- [30] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.