H-0207 November 3, 2003
Computer Science

IBM Research Report

Parallel Copying Garbage Collection Using Delayed Allocation

Erez Petrank
Department of Computer Science
Technion - Israel Institute of Technology
Haifa 32000
Israel

Elliot K. Kolodner
IBM Research Division
Haifa Research Laboratory
Haifa 31905, Israel

Parallel Copying Garbage Collection using Delayed Allocation

Erez Petrank[*] Elliot K. Kolodner[†]

Abstract

We present a new approach to parallel copying garbage collection on symmetric multiprocessor (SMP) machines appropriate for Java and other object-oriented languages. Parallel, in this setting, means that the collector runs in several parallel threads.

Our collector is based on a new idea called *delayed allocation*, which completely eliminates the fragmentation problem of previous parallel copying collectors while still keeping low synchronization, high efficiency, and simplicity of collection. In addition to this main idea, we also discuss several other ideas such as improving termination detection, balancing the distribution of work, and dealing with contention during work distribution.

Keywords: Language design and implementation, Parallel garbage collection, Memory management.

1 Introduction

Java is an important new technology, especially as the language of internet programming. This popularity is attributed to Java being a simple, object oriented, secure, portable, and platform independent language. High performance is a crucial property of any Java Virtual Machine (JVM), and since

[*]**Contact author.** Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel. Most of this work was done while the author was at the IBM Haifa Research Lab. Research supported by the Bar-Nir Bergreen Software Technology Center of Excellence. E-mail: `erez@cs.technion.ac.il.`

[†]IBM Haifa Research Laboratory, Mount Carmel, Haifa 31905, ISRAEL. `Email: kolodner@il.ibm.com`

Java provides automatic memory management and garbage collection, one of the first candidates for performance improvements is to incorporate an efficient allocator and garbage collector into the runtime.

Initially, Java was introduced as a technology for client machines on the desktop. Recently, it has also gained popularity for server machines, mainly because of its platform independence. Today's characteristic server platforms employ symmetric multiprocessors in order to increase their computing power. The use of multiprocessors is also increasing for desktop machines. Thus, taking full advantage of the multiprocessor is essential for good Java performance on these platforms.

Many garbage collection algorithms, including advanced algorithms first designed for uniprocessors, do not take advantage of a multiprocessor. In these algorithms, all application threads are stopped while a single thread executes the collector on a single processor and all other processors are idle. Thus, these collectors are not appropriate for use on a multiprocessor. A parallel garbage collector keeps all processors busy doing useful work even during collection.

In this paper we present a design for a parallel collector. Our parallel collector is appropriate both for Java and for other object oriented programming languages on a multiprocessor.

Another way to use multiprocessors efficiently is to employ a concurrent garbage collector. In such a collector, a single garbage collection thread runs concurrently with the program threads (see for example [2, 21, 22, 7, 6, 9, 8, 17]). Potentially, all processors can be kept busy during collection. However, as the number of processors and program threads increase, a single garbage collector thread may not keep up with the allocation demands of the many program threads (see for example [1, 9]), and the system may end up being single threaded as it waits for the garbage collector to free space. The scalability of the system depends on the collector being able to collect as fast as the application allocates. Thus, a concurrent collector can also benefit from the parallelization of the collector thread.

1.1 Contribution of this work

Before going on, let us define terminology for the rest of this paper. We denote by *parallel collection* a collection which is run while the application program is stopped and several parallel collectors perform the collection. We denote by *concurrent collection* a collection performed by one or more collector threads that run concurrently together with the application threads.

The main contribution of this work is the design of a parallel garbage collector appropriate for Java and other object oriented languages on SMP server machines. We present a parallel version (for SMP machines) of the well known copying garbage collector introduced in [20, 11, 3]. The advantages of the copying garbage collection are the fact that the heap is compacted in each collection, the low complexity of the algorithm which touches only the live objects (rather than touching all heap as a mark & sweep algorithm does), and the simplicity of allocation (controlled increase of a pointer).

Our main new idea is delayed allocation during the parallel collection (presented in Section 4.2). Using delayed allocation, a collector thread does not copy an object immediately; rather, it waits until it has a group of objects, and then allocates memory for all objects in the group at once and copies those objects. Delayed allocation completely eliminates the fragmentation problem of previous parallel collectors [13, 5, 19, 14, 15]. This method incurs low synchronization (as low as in previous work), it is simple (even simpler than some of the previous solutions), and it is as efficient as previous solutions.

We believe that our basic ideas can also be generalized to parallelize other copying-based garbage collection algorithms.

In addition to this main contribution, we offer several new ideas for designing a parallel copying garbage collector. First, we consider termination detection. An efficient termination detection is tricky and a previous attempt to describe a termination detection protocol [10] was faulty. We provide a correct and efficient termination detector for a parallel collector. We also discuss work distribution: first, how to break the work into small pieces to be jointly performed by the parallel collector threads, and second, what machinery should be used to incur low contention on distributing the pieces of work among the collector threads.

In addition to the aforementioned ideas, we also make several observations important for implementers of a parallel copying collectors. These include items such as dealing with modern SMP memory coherence models, and our list of design goals (in Section 2 below) that should be addressed when designing such a collector.

1.2 Organization

We start in Section 2 with the design goals. In Section 3 we review the sequential copying garbage collection. We start describing this work in Section

4 with our main idea: the delayed allocation method. A short discussions on work distribution is provided in 5. We continue with termination detection in Section 6 and we conclude in Section 7.

2 Design goals

We present the design goals for our parallel collector. The three major goals are efficiency, scalability and the preservation of the advantages of sequential copying garbage collection. These major goals are achieved via the following concrete goals.

1. **Load balancing:** Load balancing is always a crucial point in the efficiency of a parallel algorithm. Efficiency suffers if some of the processors are idle while the other processors perform the work.

2. **Scalability:** We would like the algorithm to achieve large speedup on today's SMP machine, and also to allow scalability to a bigger number of processors in future SMP's. One major consideration here is to avoid contention when accessing shared resources.

3. **Compaction:** We would like to preserve the major advantage of the sequential copying collector: the collection produces a compacted heap.

4. **Locality of reference:** An important goal in the design is to try and avoid cache misses as much as possible. A collector that incurs many cache misses cannot be considered efficient.

5. **Avoid synchronization:** The parallel threads must synchronize while distributing the work between them and while accessing mutual resources. However, it is desirable to keep the synchronization points as few as possible since performing any synchronized operation such as a compare and swap instruction (even without incurring any conflict) can be expensive.

6. **Simplicity:** Finally, we believe that the design should be simple. A very complicated collector will probably not be used in practice.

Two remarks are in place. First, in many cases, there is a tradeoff between the various goals. For example, for load balancing we will usually prefer to cut the jobs to small pieces, but for small contention we would like to let the threads work on large jobs before they have to synchronize again. In any design, we must settle these tradeoffs, and we believe that a good design leaves as many open parameters as possible so that the algorithm can be adjusted to any specific local environment.

3 Sequential Copying Garbage Collection

In order to start discussing our ideas for parallel copying collection, let us review the steps in the sequential copying collector [20, 11, 3].

1. Stop mutator threads;
2. Flip the roles of *from-space* and *to-space*;
3. Scan the roots in each mutator thread and also the global roots. For each object referenced by a root (child of a root):
    - (a) If this child is not yet copied then
        - i. Copy child to *to-space*;
        - ii. Write a forwarding pointer in (the *from-space* copy of) the child;
    - (b) Update the root pointer to point to the new copy of the child in *to-space*;
4. **Scan** *to-space***:** For each child of an object in to-space:
    - (a) If this child is not yet copied then
        - i. Copy child to *to-space*;
        - ii. Write a forwarding pointer in the child;
    - (b) Update the pointer in the father object to point to the new replica of the child in *to-space*;
5. Reclaim from-space area;
6. Release mutator threads;

4 Parallel copying collection

The basic idea of the sequential algorithm is still used and we concentrate on extending this algorithm to parallel collection. A naive parallelization of the sequential algorithm would have each collector thread do part of the scan. However, this leads to a bottleneck on the *to-space* allocation pointer. Working with a single pointer is simple and elegant, but when several collector threads perform the copies, they will heavily compete on a single resource causing unacceptable contention. Other problems also arise. For example, we don't want several collector threads to copy the same (popular) object several times, we have to distribute the parallel work carefully, etc. We start with the allocation problem and go on to the other problems in the following sections.

4.1 Previous solutions

Several previous systems tried to prevent contention on the *to-space* allocation pointer for each copy of an object. The first solution, used by Halstead [13] and Crammond [5] was to partition *to-space* into n equal spaces, where n is the number of processors, and let each processor allocate in its own private space. This completely solves the contention on allocation but has a major drawback (reported by Halstead): the allocation requests by the different processors are not even and thus one processor gets stuck on failing allocation when other processors have big empty spaces. Halstead suggested to ameliorate the behavior of the system by letting each collector allocate a “chunk” of memory and perform allocations inside the chunk privately. Namely, when a collector needs to copy an object to *to-space*, then it actually allocates a big area (a chunk), copies the object in hand, and keeps copying subsequent objects to this private area until there is no more room and a new area should be allocated.

This method, adopted by Miller and Epstein [19] following [18, 4] solves the contention conflict problem for *to-space* allocations since these allocations become much less frequent. However, a new problem arises: the fragmentation of *to-space*. Recall that one of the major benefits of a copying garbage collection is compaction of the heap. With this solution, we do not compact the heap through the collection.

To solve the fragmentation problem, Imai and Tick [15] suggested letting each processor manage several chunks, each used for a specific size of allocation. Typical sizes are powers of two, and objects that fall in between these sizes (such as an object of size 5) are allocated on the chunk that uses the smallest power of 2 big enough to hold them (e.g., allocate 8 bytes to keep an object of 5 bytes). The waste of space in their scheme is at most half, and in practice much less. However, this scheme needs management of the chunks and it complicates the solution. Also, it does not completely overcome the fragmentation problem.

Flood et al [12] adopt the chunk allocation solution denoted *local allocation buffers*. We remark that this kind of allocation has recently become common. As applications are multithreaded, some synchronization is required for allocation and using thread local allocation seems to be the best option. However, as we will claim, when allocating for the collector, more flexibility may be assumed as allocations may be delayed.

In what follows, we present a method in which the garbage collection outputs a heap with no fragmentation at all. Our solution, the *delayed allocation* method, is simpler than the Imai and Tick solution, and does not increase the contention on allocation.

4.2 Delayed allocation

The idea is to differentiate between regular allocation performed by the mutators and the special allocation that the collector needs. When a mutator allocates, the space must be assigned immediately to avoid delaying the mutator. However, the collector's allocations may be delayed. In our scheme, a collector thread does not perform each allocation immediately when the original algorithm dictates a copy. Instead, the collector thread keeps an *allocation log* in which it records which copies should be performed. Whenever a copy of an object from *from-space* to *to-space* is needed, the collector thread adds a record to the allocation log in which it puts the *from-space* address of the object and the *to-space* (or root) address of the cell pointing to the object. Also, it updates the accumulated size of all objects mentioned in the allocation log. This single number is kept at the beginning of the log.[1]

This accumulated size, i.e., the sum of all objects to be copied, is the space needed to apply the allocation log. When the accumulated size is big enough, e.g., a page, the collector actually applies the allocation log: it allocates the exact space needed for all the objects, and then it copies the objects.

Note that there is no fragmentation at all since the allocated space in *to-space* exactly matches the space needed to copy the objects mentioned in the log. Also, the frequency of conflicts and synchronized operations does not increase. Finally , big objects do not require special care, and they fall naturally into the framework set by delayed allocation.

One may think that delayed allocation has a disadvantage in foiling locality of reference. For each object we start by looking at its header and only (somewhat) later we copy it as a whole. So if the header is evacuated from the cache, we get an additional cache miss. However, fixing reasonable parameters (similar to previous work), eliminates this problem. If the cache is big enough to hold all copied objects in a chunk twice (once for *from-*

[1]One may choose to keep all sizes of all objects in the allocation records. This is a good idea if detecting the length of an object requires a few operations, and we do not want to read this length in the *from-space* area twice.

space and once for *to-space*) and also the allocation log itself, then we get no additional cache misses. Setting the chunk size to around 1kb ensures good behavior on most processors available today. In any case, one must tune this parameter carefully.

We proceed with the next synchronization issue: the parallel access of objects in *from-space*.

4.3 Synchronizing access to *from-space*

The parallel access to *from-space* is the second obstacle that has to be properly managed. It is possible that two collector threads will try to work on the same *from-space* object, since they are scanning two different parents of this object in parallel. We would like to stress that the contention on *from-space* handling is of far lower likelihood than the contention on *to-space* allocation. For the latter, any two collectors copying any two objects cause contention on *to-space* allocation. Whereas only two collectors that try to handle the very same object at the same time will face contention on *from-space* handling. This has indeed been reported as a minor problem in previous works. Halstead [13] reports less than one conflict per second (experienced with Concert Multilisp running on eight processors). This is the reason why we don't feel there is a need for an advanced mechanism to handle these contentions. Our mechanism is simple (and standard) and allows a good distribution of work between the collectors.

The data structure we keep consists of two bits per object, the *work bit* and the *done bit*, and also a separate list called the *parents-log*. The done bit indicates that the object was copied to *to-space*. This bit must also be used in the sequential version of the algorithm. In some systems, it is possible to tell whether a forwarding pointer was written in the header of the object, and in this case, the done bit is not needed. In addition to the done bit (or the ability to tell whether a forwarding pointer has been written), we need an additional bit for our parallel version of the algorithm: the work bit. This bit indicates that the object is now being copied to *to-space* by some collector and there is no need to copy it again. At the start of a collection, the work bit and the done bit are cleared at all objects.

The parents-log contains records of parents whose pointers reference *from-space* and should be updated to reference the *to-space* copies. We will explain the need for the parents log later. Let us proceed with the algorithm.

Consider a collector thread that is scanning a pointer that references a *from-space* object. Either the pointer resides in *to-space* or it is a mutator root. The collector has to copy the referenced object into *to-space* if it has not yet been copied, and then update the pointer. The collector reads the work and done bits in the child. If the done bit is set, then the collector only needs to update the given pointer according to the forwarding pointer in the child. Another possibility is that the work bit is not set. In this case, the collector has to perform the actual copy of the child into *to-space*. To do this, the collector uses a synchronized operation (such as compare and swap) to set the work bit. We begin with describing the case that this operation succeeded and the collector is now responsible for copying the object. We will deal later (in Subsection 4.3.1 below) with the two similar cases that remain: The case that the collector failed to set the work bit (i.e., another processor won and is doing the copy) and the case that upon reading the bits of the child object, the collector found that the work bit was set but the done bit was not set.

So suppose the collector did set the work bit of the object. It then checks the size of the object and adds a record to the allocation log containing the location of the pointer and the address of the *from-space* object. Also, the collector adds the object length to the accumulated size of the objects registered in the allocation log and checks if it is time to do the actual allocation, i.e., if the total size of objects in the allocation log has grown big enough. If it is, the collector actually allocates the needed space and applies the records in the allocation log.[2] Applying a record means: Copying the relevant object, setting the done-bit in the *from-space* copy, clearing the work-bit and done-bit in the *to-space* copy, and updating the parent pointer to reference the new copy in *to-space*.

4.3.1 The parents log

We now return to the case that the collector has a pointer to update, but the pointed object is being handled by another collector thread. One cannot let the collector wait till the other collector finishes the update of the child, since this option is not efficient and could lead to a deadlock. Instead, we use a global structure called the *parents log* in which the collector writes a request to later update the pointer. A record in the log contains the address of the pointer which should be updated and the address of the child in *from-space*. The log is global (rather than being associated with an object), and

[2]We remark that locality considerations dictate that the log should be applied from least recently written record and back to the beginning.

the collector threads apply the parents log when they cannot find anymore objects to scan (usually, towards the end of the collection).

Synchronization to the parents log can be made negligible using buffering. Instead of updating the parents log each time a problematic pointer is traversed, the collector stores the parents-log-record in a local private buffer. When several records have been accumulated, it adds the buffer to the parents log in a synchronized manner. Thus, the parents-log becomes a list of buffers, each of which, contains actual records of the parents log. Later, a collector applies the records in the log by removing a full buffer from the log and applying the records in the buffer. Synchronization is minimal since it occurs only when buffers are added or removed from the log. The size of the buffers can be set as a parameter, tuned by the behavior of the applications.

4.4 Heap management for the application

Garbage collection is tightly coupled with the heap manager. Note that our method for *to-space* allocations during garbage collection is inappropriate for managing the heap allocation by the mutators. Mutator allocations cannot be delayed without delaying the mutator. Thus, we adopt Halstead's idea of memory-chunks (or thread local allocation) for application allocation.

5 Work distribution

Load balancing is one of the more important issues in making parallel implementations run faster. Letting one processor do the work while other processors are idle does not fully utilize a multiprocessor machine. Imai and Tick [15] were the first to take explicit care for balancing the load of a parallel collector, and Endo et. al. [10] provided an enlightening measurements showing the strong influence of load balancing on efficiency[3]. Flood et al [12] have adopted the queue stealing method of [10] for load balancing. We believe the methods discussed in these two works are good and should be adopted. The idea is that list of tasks are kept locally and may be stolen by other collector threads.

[3]Endo et. al. implemented a mark & sweep algorithm. In a mark and sweep algorithm the collector *marks* all live objects, and later scans the whole heap and reclaims (sweeps) unmarked objects. Note that although this is not a copying algorithm, this algorithm also scans all live objects, and thus has similar behavior. See [16] for a detailed description of mark & sweep algorithms.

6 Terminating the collection

When do the collectors know that the collection has terminated? Termination occurs when all the heap has been scanned, all live objects have been copied and all pointers have been updated to point into the *to-space* area. In practice, this means that the collectors finish all jobs in the job lists, and finish applying all records in the parents log.

A collector can check that the job lists are empty and that the parents log is empty, but it must also check that all the other collector threads are idle and not producing more work to be done. Furthermore, the check must be atomic since another collector thread may write a new entry to the job lists, and later become idle. The issue of termination detection is error prone. In fact, a previous solution ([10], Section 4.2 there), for detecting termination in a parallel mark & sweep collector, has a flaw which we shortly describe in Subsection 6.3 below.

We present a modification to the previously suggested termination detection [10]. For simplicity of presentation, we describe the algorithm assuming strong memory coherency and then (in Section 6.2 below) we discuss how to fix it for weak coherency.

The data structure we use consists of

1. One global flag called the *detection flag* initially cleared,
2. A global word called the *detector-id* initially set to 0,
3. A flag for each collector thread called the *idle bit* initially cleared,
4. and one global flag called the *global termination flag* initially cleared.

The detector-id should be big enough to contain any collector thread identity and one additional value that cannot be an identity (we denote this value by 0).

To support termination detection the collectors maintain their idle bit as follows. Whenever the thread is not working, its idle bit is set. In particular, a thread sets its idle bit when it finishes scanning its own areas, and has to look for a new area to scan in a tasks list. It then scans the lists and the parents log to look for a job. Once it detects a job candidate, it clears the idle bit and then it "competes" on the job by performing a synchronized operation (e.g., compare and swap) trying to remove the job from its list (tasks list or parents log). If the collector fails to obtain the job, it sets the idle bit again and continues the search. Finally, to support the termination detection, the collector threads also perform the following operation: whenever a collector thread adds a record (or buffer) to the tasks list or to the parents log then before the add operation, it sets the detection flag. Intuitively, the detection flag is set to indicate that there is activity in the system and termination has not been reached yet.

A collector starts termination detection if the job market is empty. To check termination, the thread checks the global *detector id.* If it is not set (i.e., equals 0), the thread competes (compare and swap) on writing its id to the detector id. If it succeeds, it clears the *detection flag.* It then goes over all lists to verify that they are empty (tasks lists and parents log) and goes over all other threads to check that they are idle. Next, it checks that the detection flag is still cleared, and if all the above hold then it decides that termination was detected. In this case, it sets the global termination flag, clears the detector id to 0 and halts.

When a thread wants to check termination and the detector id has another thread id, the thread waits until the detector id is reset to zero. When it is, the collector thread checks the global termination flag. If the flag is set, the thread halts. Otherwise, it competes on the detector id to start its own termination detection.

6.1 A few words on correctness

Let us say a few words on why this termination detection is correct. Note the course of detection. The detector thread starts by verifying that all job lists are empty and afterwards it verifies that all collector threads are idle. Clearly, if the collection indeed terminated then a detecting thread will detect it: collector threads cannot find jobs so they will all remain idle, and the lists of jobs will remain empty. Thus, any detector will detect termination and halt.

It remains to show that no thread will ever halt if the collection is not yet over. A property of this algorithm is that if the collection is not yet over, then at any point in time there must be some non-idle collector thread or some job hanging on some list. The problem is threads check termination in a non-atomic manner. Suppose that the collection is not done yet, and let us check if the collector can erroneously decide to terminate. If the collector finds any non-empty job list or any non-idle collector thread, then it does not terminate. We will argue that if the collection is not over when the detector thread finishes the test and the detection flag is not raised, then it is not possible that the detector will find all lists empty and all idle bits set.

To show this claim we stress again the order of the checks. The emptiness of the lists is checked before the idleness of the collectors is checked. Consider the time between these two checks. If at that time one of the lists is not empty then we are done: this list was empty when the detector thread checked it and now it is not. Therefore, an action of adding to the lists was taken, and the detection flag must be also set and the detection will fail. So when the detector thread starts to check the idle bits we may assume that all lists are empty. If during the check of the idle bits a job is added to the lists by any of the collector threads then again the detection flag is set and the detection fails. So we may also assume that while the detector checks the idle bits of all collector threads the job lists remain empty.

Now, if the lists are empty and remain empty, then no collector thread can clear its idle bit: a collector clears its idle bit only when attempting to get a new job from the job lists. So each collector may either be idle now or become idle. But no collector can stop being idle and become active. But we also assumed that the collection is not over, and since all job lists are empty, then there must be a collector thread that is not idle throughout the detection. this collector will be noted by the detector thread, which will not detect termination.

6.2 The memory coherence model

Let us say a few words on the behavior of the detection algorithm on modern multiprocessors, e.g., Power-PC, Sparc, Alpha, and Pentium. these architectures typically do not provide strong memory coherency. Namely, the order of updates executed by Processor P_1 is not necessarily the order viewed by Processor P_2. Thus, the solution outlined above does not work without modification. For example, think of a thread that raises the detection flag, adds a job (an area to scan) to the task list, and later becomes idle. It is possible that although the setting of the idle bit of the thread is visible to other processors, the setting of the detection flag is not yet visible, making detectors on other processors erroneously terminate.

Thus, in a multiprocessor environment with a weak memory coherence model, a modification is needed in the algorithm. On all such multiprocessors, there is a synchronization instruction (such as *sync* on the Power-PC, *membar* on SPARC, and *wbinvd* on the Pentium.) These instructions typically provide the following guarantee: all updates in the instruction stream before the execution of the sync operation, will appear in the view of all processors before all updates that appear after the execution of the sync operation. Such an operation is expensive (as all synchronization operations are).

Returning to our termination detector, note that we have to take care of the following course of events: A collector sets the detection flag, it puts a job in some list, and may later become idle. We make the collector perform a sync operation after setting the detection flag and just before putting a job in the list. This makes sure that any thread that detects termination may find a collector thread idle only after his view contains the setting of the detection flag performed by that collector.

6.3 A flaw in a previous termination detection protocol [10]

A previous termination detection protocol [10] relies only on a detection flag, without the detector id. We argue here that this detection is not correct. In their scheme, a detecting thread (or process) clears the detection flag, and starts checking for idleness of the system. Any activity in the system implies setting the flag. After the detector observes no activity in the system, the thread verifies that the detection flag was not set and then halts.

The problem is that even if there is an activity in the system which causes the flag to be set, at a later time, another collector thread may start detecting termination and clear the flag just before the first detector looks at the flag again. Thus, the second detector misleads the first detector to think that the flag was not set throughout the detection, and the first collector terminates erroneously.

7 Conclusions

We introduced a design for a parallel copying garbage collector, which completely eliminates fragmentation, and is nevertheless efficient, low on synchronization, and simple. Our collector distributes the work with low synchronization overhead and has an efficient termination detection mechanism.

References

[1] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[2] 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.

[3] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[4] Anthony J. Cortemanche. MultiTrash, a parallel garbage collector for MultiScheme. Bachelor's thesis, MIT Press, January 1986.

[5] Jim Crammond. A garbage collection algorithm for shared memory parallel processors. *International Journal Of Parallel Programming*, 17(6):497–522, 1988.

[6] David L. Detlefs. *Concurrent, Atomic Garbage Collection*. PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 15213, November 1991.

[7] 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. *Communications of the ACM*, 21(11):965–975, November 1978.

[8] 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.

[9] 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.

[10] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of High Performance Computing and Networking (SC'97)*, 1997.

[11] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[12] 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.

[13] Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[14] Maurice Herlihy and J. Eliot B Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3), May 1992.

[15] A. Imai and E. Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 4(9), September 1993.

[16] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[17] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In *OOPSLA'01 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 36(10) of *ACM SIGPLAN Notices*, Tampa, FL, October 2001. ACM Press.

[18] James S. Miller. *MultiScheme: A Parallel Processing System Based on MIT Scheme*. PhD thesis, MIT Press, 1987. Also Technical Report MIT/LCS/402.

[19] James S. Miller and B. Epstein. Garbage collection in MultiScheme. In *US/Japan Workshop on Parallel Lisp, LNCS 441*, pages 138–160, June 1990.

[20] Marvin L. Minsky. A Lisp garbage collector algorithm using serial secondary storage. Technical Report Memo 58 (rev.), Project MAC, MIT, Cambridge, MA, December 1963.

[21] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[22] Guy L. Steele. Corrigendum: Multiprocessing compactifying garbage collection. *Communications of the ACM*, 19(6):354, June 1976.