Parallel Copying Garbage Collection using Delayed Allocation

Elliot K. Kolodner[*] Erez Petrank[†]

IBM Haifa Resrach Lab

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. Finally, we explain how the ideas presented here can be extended to deal with generational copying garbage collection and replication-based garbage collection. We believe that our ideas can be generalized in a similar manner for other copying-based garbage collection algorithms.

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

[*]E-mail: `kolodner@haifa.vnet.ibm.com`.

[†]**Contact author.** E-mail: `erezp@haifa.vnet.ibm.com`.

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 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, such as generational scavenging [22] and the train algorithm [18], 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 [4, 1, 27, 9, 19, 10, 12, 11]). 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, 12]), 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 [25, 14, 6]. 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 4.1). 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 [15, 8, 24, 16, 20]. 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.

Next, we extend the design to parallelize generational garbage collection [22]. A similar approach works for the train algorithm [18] as well. Also, we discuss the extension needed to incorporate our algorithm into the replication-based garbage collection of [27, 26]. 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 [13] 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 implementors 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. We go on with discussing the work distribution. The machinery is described in Section 5.1, and in Section 5.2, we discuss how the work should be partitioned to small chunks. We continue with termination detection in Section 6. In Section 7 we extend the discussion to generational collection and finally, in Section 8 we talk about combining our algorithm with the replication-based garbage collection.

2 Design goals

We present the design goals for our parallel collector. The three major goals are efficiency, scalablity and the preservation of the advantages of sequential copying garbage collection. There are two aspects to efficiency: latency and throughput. Low latency means that the application program will be stopped for short pauses for garbage collection. However, we do not want to reduce the pauses by paying an unbearable overhead cost for the application program (for example, by making it perform a costly procedure for each update). Thus, an algorithm is also judged by how much it reduces the throughput of work in the system. We aim for high throughput. Many times there is a tradeoff which we must settle between these two goals. Let us go on with the more specific design 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 [25, 14, 6].
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 (son of a root):
    (a) If this son is not yet copied then
        i. Copy son to *to-space*;
        ii. Write a forwarding pointer in (the *from-space* copy of) the son;

(b) Update the root pointer to point to the new copy of the son in *to-space*;

4. **Scan *to-space*:** For each son of an object in to-space:
   (a) If this son is not yet copied then
      i. Copy son to *to-space*;
      ii. Write a forwarding pointer in the son;
   (b) Update the pointer in the father object to point to the new replica of the son in *to-space*;
5. Reclaim from-space area; 6. Release mutator threads;

4 Delayed allocation

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.

The goal is to prevent contention on the *to-space* allocation pointer for each copy of an object. The first solution, used by Halstead [15] and Crammond [8] 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 [24] following [23, 7] 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 [20] 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.

Another reasonable enhancement, which we would like to point out, is to treat big objects separately. For example, if an object is bigger than an 1/8 of the size of an area, then allocation is performed especially for this object, and it is not copied into the current area used by the collector. As a result of this treatment, the "holes" in *to-space* cannot be bigger than 1/8 the size of an area, since all objects copied into the area are smaller than that. This parameter (the 1/8) can be optimized to trade between conflicts frequency on allocation and the amount of fragmentation we expect.

However, 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.

4.1 Our solution

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-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 around 1kb ensures good behavior on most processors available today. In any case, one must tune this parameter carefully. We suggest further means to improve locality of reference later in Section 5.1.

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

4.2 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

[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.

on *from-space* handling. This has indeed been reported as a small issue in previous works. Halstead [15] 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 son. If the done bit is set, then the collector only needs to update the given pointer according to the forwarding pointer in the son. Another possibility is that the work bit is not set. In this case, the collector has to perform the actual copy of the son 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 4.2.1 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 son 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*.

4.2.1 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 son, since this option could lead to 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 son in *from-space*. The log is global (rather than being associated with an object), and the collector threads apply the parents log when they cannot find anymore objects to scan (usually, towards the end of the collection). We will discuss the work distribution and deal with applying the parents log in Section 5.2.2 below.

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.

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

4.3 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 would like to add some words on how the heap management can be implemented. We believe that the management must be simple and with low contention. However, we must allow some fragmentation until the compaction of the next collection.

We believe that Halstead's idea of memory-chunks is adequate for this case. Each mutator obtains a chunk and performs allocations for small objects in the chunk. When the chunk fills, the mutator obtains another chunk. Synchronization with other mutators is required when obtaining a chunk. Large objects are allocated separately. To make sure that the fragmentation does not exceed an 1/8 of the heap, we define large objects to be those which are bigger than 1/8 the size of a chunk. Thus, at most 1/8 of each chunk may be wasted.

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 yield the benefits of a multiprocessor machine. Imai and Tick [20] were the first to take explicit care for balancing the load of a parallel collector, and Endo et. al. [13] provided an enlightening measurements showing the strong influence of load balancing on efficiency[3]. In this section we discuss several issues related to load balancing. First, we need to split the overall job into "job chunks" to be distributed between the collector threads. Second, we must discuss the machinery needed to distribute the jobs. Namely, how does one collector thread that has "too much work", lets the other (less busy) collectors help. We start with the machinery and later, in Section 5.2, we discuss how to cut the heap scan into "job chunks" properly.

[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 [21] for a detailed description of mark & sweep algorithms.

5.1 Machinery

Let us begin by noting that "work" in this setting means scanning the heap and a "chunk of work" to be executed means an area in the heap that has to be scanned (it can be defined, for example, by providing a pointer to the beginning of the area and its size).

Imai and Tick used a global list of areas which held the areas to be scanned. Each collector worked on its private area, and when done, it searched for a new area in the global list. Areas that need to be scanned are put in the list immediately upon creation. Access to the global list was synchronized. Endo et. al. implemented the other extreme: They used n global lists, where n is the number of collector threads. Each collector had its own global list, in which it puts areas to be scanned by other collector threads.

We prefer an intermediate implementation. Keeping one list has the disadvantage of contention: several threads try to add or remove items from a single list. Thus, it is reasonable to use more than one list. However, using many lists makes it inefficient to look for jobs: One must traverse all collectors to search for a job, especially if only a handful of jobs exist in the lists. We suggest using 2-4 lists independently of the number of collectors. Note that the number of lists can change dynamically. A collector thread that notes contention on lists modification may add a list, and a thread that finds empty lists may remove a list. These actions must be carefully implemented as these resources are shared by all collector threads.

To add an area to the global lists, a collector thread chooses one of the lists at random and adds to that list synchronizing its access. The probability of a conflict between two accessing threads is proportional to the reciprocal of the number of lists. To remove an area, the thread may randomly choose a list and read it to see if there are available areas there. If not, it may proceed to the other lists and check them.

Let us say a few words on locality of reference and load balancing. During the collection, each collector scans an area and produces new areas to scan. (Recall that scanning includes copying objects into the *to-space* area and these objects themselves must be scanned to update their pointers.) Adopting the ideas of Endo et. al. [13] for copying collectors has good caching behavior. Each collector thread prefers to scan areas that it has previously created (areas in its own global list). But since we don't want to keep so many global lists, we suggest that each thread keeps the last area that it has created and scans it himself. Namely, if a thread produces more than one area, then it keeps the latest area for its future work and puts previous areas in the global lists.

5.2 Breaking the work into small pieces

We proceed with discussing how to break the collection into small pieces, which can be distributed between the collectors. We separate the two stages in the scan: we deal with scanning the roots in Section 5.2.1 and with scanning *to-space* in Section 5.2.2.

5.2.1 Scanning the roots

We assume that marking roots for a single application thread is not a long task and thus, there is no need to break this task into several small tasks and distribute them between the collector threads. In other words, each collector thread will scan the roots of a single application thread, and when done, go on to scanning the roots of the next application thread. We stress that even if the assumption is wrong, and scanning the roots of one of the threads turns out to be a long task, the load balance will not be jeopardized. This is because once a collector thread cannot find a fresh mutator thread to scan roots for, it simply moves on to the next stage in which it scans *to-space*.

We keep for each mutator (application) thread one bit: the *scan bit*, which indicates whether its roots are already being scanned. A collector which is searching for a job goes over the mutator threads and reads their scan bit. Once the collector thread makes a full pass over all mutator threads and finds that all the scan bits are set, the collector proceeds to next stage: Scanning *to-space* objects. If the collector finds a scan bit which is not set, it uses a synchronized operation (such as *compare and swap*) to set the scan bit. If setting the bit failed, it goes on to reading the scan bit of the next mutator thread. If the synchronized setting of the scan bit succeeded, then the collector thread starts scanning the roots of the mutator thread for which it has set the scan bit.

Remark 5.1 In addition to the all the local roots of all the mutator threads, we must also scan the global roots, used by the system (or compiler, or interpreter). We consider all global roots to be one task. Namely, in terms of work distribution for the collector, one may think of these roots as being the roots of some additional virtual thread.

5.2.2 Scanning to-space

How do we partition the work into small "chunks" when we scan *to-space*? The most natural choice is to use the area sizes as output by the delayed allocation scheme (see Section 4.1). Recall that when the allocation log indicates the need to allocate an area bigger than a predetermined size, then this area is actually allocated and the log is applied. Once this area is created, it needs scanning, and may be added to the global lists. These areas indeed form the basic blocks for the collection. In addition, when a collector thread cannot find an area in the global lists, it tries to apply a buffer in the parents-log (see Subsection 4.2.1).

Notice that as a benefit to the way we create areas, we ensure that an area consists of complete objects. Thus, it is easy to scan an area an object at a time and identify the pointer fields in the object.

One drawback of the above choice of area sizes is that there is no limit on the size of the area. We only know that it must be bigger than a predetermined size but the area can be very big. To solve this, we let each collector return part of the area to the global lists if the area turns out to be too big. Note the trade-off between improving the load balancing and reducing synchronization: cutting areas into smaller areas requires an additional synchronized access to the global lists.

Let us finish by pointing out an interesting phenomena: the natural dynamics of the collection should automatically improve the load balancing among the collector threads. What is the real amount of work that is done on a given area in *to-space*? It depends not only on the size of the area, but also on the number of pointers in the area, on the percentage of the sons that have been copied already, and on the size of the sons. When we reach the end of the collection, we get that the average amount of work on each area decreases, since many sons have already been copied and the collector only needs to update the reference. But this is exactly the time that we would like job chunks to be small, since at the end of the collection we don't want one collector to work for a long time while the other collectors are idle. We prefer small chunks of work. Thus, the behavior of the system naturally balances the load among the collectors.

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 area lists, and finish applying all records in the parents log.

A collector can check that the area 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 area to the area lists, and later become idle. The issue of termination detection is error prone. In fact, a previous solution ([13], 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 [13]. 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 global list. It then scans the area 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 (area 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 area 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 (area 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. The reader should first convince herself 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. We skip the details. The problem is in the check is non-atomic. Suppose that the collection is not done yet, and let us check if the collector can erroneously decides 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 ne 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 an area to the area 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 [13]

A previous termination detection protocol [13] 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 Extension to Generations

Generational garbage collectors rely on the assumption that many objects die young. Thus, if we partition the heap to an area containing the young objects and an area that contains the old objects, then it is useful to collect the garbage in the young area more frequently.

Generational collectors (introduced in [22]) divide the heap into two or more generations (or parts) and the younger the generation the more frequently it is collected. New objects are allocated in the youngest generation which is collected whenever an allocation fails. Some of the surviving objects (the older objects) are promoted to the next generation. When the promotion fails because there is not enough space, the next generation is also collected and some of the surviving objects are promoted to an older generation and so forth.

Since we collect only a part of the heap, we have to know which objects in the rest of the heap point into the part of the heap containing the generations that we are collecting. These pointers are called inter-generational pointers. Usually, when we collect a generation we also collect all the generations that are younger than that generation. The primary reason is that we only need to scan pointers from old generations to younger generations. The number of such pointers is typically small and thus, generational collections can use a data structure to maintain an (almost) updated list of these inter-generational pointers. The reader may find a detailed description of the possible options for recording inter-generational pointers and promotion policies in Jones and Lins [21].

In the following sections we generalize the ideas developed so far and construct a parallel collector for generational scavenging. We consider a simple scheme in which there are two generations (young and old), and both are collected by a copying collector. When collecting the old generation we collect the (much smaller) young generation as well, i.e., the full heap. Thus, for the old generation, we can use the standard copying collection discussed in previous sections, and we concentrate on how the young generation is collected. The avenues we use to make the scheme parallel depend on the specific way in which inter-generational pointers are recorded. In this paper, we follow a simple scheme based on [30, 17], which we describe below. Although we discuss a specific design, our ideas can be easily modified to suit other variants of generational scavenging.

There has been no previous work on parallel generational collection. The closest is the work of Miller et al. [24, 23, 7], which divide the heap into a static area and a dynamic area. However, there was no promotion of objects nor recording of intergenerational pointers. See [24] for details.

7.1 Maintaining inter-generational pointers

We use two generations and the inter-generational pointers are kept using a combination of card marking and a remembered set as in [17]. The remembered set contains the locations of all inter-generational pointers as recorded in the last collection. In addition, the heap is divided into cards (e.g., 1kb in size), and we keep a table indicating which cards were updated since the last collection. Thus, the inter-generational pointers appear either in the remembered set or in cards that are marked. It is the mutator responsibility to mark each card while the card is updated.

In order to find inter-generational pointers, the collector checks the dirty cards and the remembered set in order to remove entries in the remembered set that are outdated (i.e., those that have been changed by the mutators) and add entries for pointers that were modified and are now inter-generational. We refer the reader to a forthcoming paper [3] for a method to efficiently combine card marking with remembered sets when more than two generations are used.

7.2 Scanning the roots of the young generation

Scanning the roots for the young generation includes both scanning the mutators roots and the inter-generational pointers. In the non-generational case, we have suggested in Section 5.2.1 not to spend much effort on maintaining the load balance while scanning the roots, since collector threads that cannot find work on the roots scan can continue with the scan of *to-space* and do not become idle. This approach may still be good for generational collection. Thus, we may let one collector go over the card marking and over the remembered set and scan the relevant pointers. The other collectors scan the mutators roots and then move on to scan *to-space*. This is probably the first solution to be implemented.

However, a problem may come up in this special case of the young generation collection. First, the scanned heap (the young generation) is small, and so, scanning it may end quickly. Second, the work on scanning the marked cards and the remembered set may turn out long. Thus, the load may become unbalanced eventually: all threads wait for the special collector thread, which scans the inter-generational pointers. This collector thread becomes the bottle neck. Thus, We would still like to suggest some simple ways to partition the work of marking the inter-generational roots among several collectors.

Decrease delayed allocation limit: One very simple change is to decrease the limit on the delayed allocation (see Subsection 4.1 above). In this way, the special collector will produce work more often and this may be enough to keep the other collectors busy. This fix is trivial and this parameter can be modified dynamically by unemployed collectors. Note that here we trade the frequency of synchronization with the load balance.

Partition the work on delayed allocation: A more significant change (although still simple), is to partition the operation associated with the delayed allocation into two. The collector that produces the allocation log does not apply the log once the actual allocation has to be performed. Namely, we let the special collector that scans the inter-generational pointers fill an allocation log and then another collector takes the log and applies the records in the log while the special collector goes on in producing the next

allocation log. Here we trade the frequency of synchronization and locality of reference with the load balancing.
A real partition of work: To go further in partitioning the work of the special collector, we suggest to partition the cards between the collectors such that each collector scans its assigned cards and updates the relevant part of the remembered set.

7.3 Scanning the heap

To actually scan the roots and the heap we use the same techniques as in Sections 3 – 6 above. Let us concentrate on the differences. The major difference is that the objects are not copied to a single space anymore, but they are either copied to the *to-space* part of the young generation or promoted. In the more general case of several generations, there are even more than two possible destinations. We adopt the following policy. Each collector keeps an allocation log for each of the destinations into which it actually needs to copy objects. So if a copy to a specific generation has to take place, the collector appends a record to its allocation log that is associated with the corresponding generation. If the collector does not currently hold an allocation log for that generation, then it creates a new log for that generation. In general, we expect less conflicts in this case since the collectors compete on more than one pointer. Thus, the limit on the size needed to actually allocate could be made smaller.

7.4 The train algorithm

We would like to point out that our scheme also fits the train algorithm [18, 28]. There, the young generation may be collected together with a car from the old generation. Again, remembered sets and card marking can be used to record inter-generational pointers (a detailed discussion on this combination for the train algorithm appears in [3]). Almost the same algorithm (as above) can be used for a parallel version of the train algorithm: Delayed allocation with separate logs for each new location and work distribution as described in this section.

8 Replication-based collection

In a concurrent collector, one or more collector threads run concurrently with the mutator threads. Most concurrent collectors require a synchronization point, where all mutators are stopped (hopefully) briefly initiate and/or finish the collection cycle. This kind of collector was first presented by Baker [4] for copying collectors and were further studied in several subsequent papers (see for example [1, 27, 9, 19]). A similar approach for mark & sweep collection was also well studied (see for example [10, 12, 11]). For a complete survey on this line of research see [21]. Let us explain how our parallel collector can be combined with a concurrent copying collector.

As an example, we choose to extend the replication-based garbage collection of [27, 26]. Two aspects of this extension are interesting. First, making the collector run in parallel when the application is stopped to finish a collection cycle. Using a single collector thread would be wasteful: all processors but one would be idle during the pauses. Second, we would like the concurrent collector, which runs concurrently with the application, to run in several parallel threads. This is useful when the allocation rate of the parallel application is high and outpaces the work of a single collector thread. Increasing the efficiency of the concurrent collector (by making it parallel) avoids long interruptions to the application by collecting unused objects faster.

We choose replication-based garbage collection as an example only. The same ideas can be implemented for other concurrent copying garbage collectors such as [4, 1, 5].

8.1 Replication-based collection

The replication-based collector [27, 26] starts a collection cycle by switching the names of the semi-spaces *from-space* and *to-space* without stopping the mutator threads. While the mutators keep running and operating on *from-space*, the collector replicates the live objects from the *from-space* area into the *to-space* area. Finally, the mutator threads are stopped and their roots are updated to point to the replicated objects in the *to-space* area.

The problem is that while the replication is executed, objects in *from-space* keep on changing and this has to be reflected in the to-space replica. In order to make the replica consistent, the mutators log all modifications to a *mutation-log*. The collector updates the replica according to the mutation log. In case a pointer is modified, its sons are scanned as well. Once the mutation log is processed (i.e., all its records were applied by the collector on the replica), the collector may stop the mutator threads for a pause in which the collector processes the mutation log again (additional entries may have been appended until the mutators stopped) and updates the mutator roots. The pause for the final update (flip) is supposed to be short. Note that when a pointer is modified by an application of the mutation log, its descendants must be traced (in the same manner that other live objects are traced).

8.2 Enhanced collector for multiprocessor machine

We first concentrate on the part of the collection in which the mutators are stopped and the collector finishes the cycle. This step is sensitive to the implementation of the mutation log: do we record the details of the mutation or just its location. More specifically, one option is to record the *from-space* address, which was updated, together with the new value to be put in the *to-space* replica. To apply the log, the collector writes the recorded value to the corresponding address in the *to-space* area. The second option, is to only record the *from-space* address in the log. Later, the collector uses this address to read the updated value from the *from-space* area and copies it to the *to-space* area. Memory coherence considerations, which we do not discuss here, dictate using the second option. See [2] for details. Thus, we stick to the second option. Furthermore, it is suggested in [2] to use a buffering method to implement the log. Namely, the mutators add mutation-records to a private buffer and only upon filling a buffer, a mutator adds the full buffer to the mutation log. Later, the collector takes full buffers from the log and applies the records in the buffer. Adopting this method yields another advantage of less synchronization overhead: the collector must only synchronize once per buffer and not for each record in the log. Furthermore, since the values are not stored in the mutation record, no further synchronization is required for updating the values. It is possible that two collectors compete on writing to the same *to-space* address (since the log may hold more than one record with updates to this address), but this is no cause for concern, since the two collectors are guaranteed to write the same value to that address: the value that appears at the given *from-space* address.

It remains to take care of scanning modified pointers. Here we need to synchronize the processors to do the scanning in parallel. The simplest option is to let only one of the collectors do the scans. This assumes that these scans are quick. So we let all collectors but one apply the mutation log in parallel, while the special collector accepts requests for scanning and performs the scans. However, in case the scannings are long and the single collector method yields an imbalanced work distribution, one must go back to implementing the full parallel copying collection as described in Sections 4 – 6 above.

Our second interest is in making the concurrent collector parallel. It is sometimes useful to run the concurrent collector (that runs concurrently with the application) on several collector threads so that it can compete with (parallel) applications that make extensive allocations. The algorithm we suggest in this paper is suitable for this purpose. Note that the concurrent collector of [27, 26] does the copying collection under the assumption that there is no application running concurrently. Only later, the mutation log is used to fix the collection with the modifications that were applied on the heap during the collection. Thus, our algorithm can be applied as is. One additional operation is applying the mutation log, which is a simple operation as discussed above, and which requires little synchronization. The scans that are implied by pointer modifications can be done by the collectors that discovers them or be added to the global lists for load balancing.

9 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. We extended the collector to generational collection (including the train algorithm) and replication-based algorithm. We plan to prototype this collector as part of a JVM on a multiprocessor platform.

10 Acknowledgment

We thank Alon Adir for explaining to us the memory coherence model on the IBM Power-PC.

References

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

[2] A. Azagury, E. K. Kolodner, and E. Petrank. A Note on the Implementation of Replication-Based Garbage Collection for Multithreaded Applications and Multiprocessor Environments. Preprint, January 1998.

[3] A. Azagury, E. K. Kolodner, E. Petrank, and Z. Yehudai. Combining Card Marking with Remmebered Sets: How to Save Scanning Time. Preprint, March 1998.

[4] 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, MIT, 1977.

[5] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157-164, 1991.

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

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

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

[9] D. L. Detlefs. Concurrent, atomic garbage collection. In *Topics in Advanced Language Implementation* chapter 5, pages 101-134. Th MIT Press 1991.

[10] 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.

[11] D. Doligez and G. 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, 1994, pages 113-123.

[12] D. Doligez and X. 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. ACM Press, January 1993.

[13] T. Endo, K. Taura, and k. Yonezawa. A Scaleable Mark-Sweep Garbage Collector on Large-Scale Shared-Memory Machines. Proceedings of the *SC97: High Performance Networking and Computing*, Nov. 1997. Web access: `http://www.supercomp.org/sc97/proceedings/TECH/ENDO/INDEX.HTM`.

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

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

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

[17] A. L. Hosking and J. E. B. Moss. Remembered Sets Can Also Play Cards. In *OOPSLA'93 Workshop on Garbage Collection and Memory Management.* Washington, DC, September 1993.

[18] R. L. Hudson and J. E. B. Moss. Incremental garbage collection for mature objects. In Yves Bekkers and Jacques Cohen, editors. Proceedings of International Workshop on Memory Management, volume 637 of Lecture Notes in Computer Science, 1992. Springer-Verlag.

[19] L. Huelsbergen and J. R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In the *Fourth Annual ACM Symposium on Principles and Practice of Parallel Programming*, volume 28(7) of ACM SIGPLAN Notices, 1993. pages 73-82.

[20] A. Imai and E. Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, vol 4, no. 9, September 1993.

[21] R. E. Jones and R. D. Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons, July 1996.

[22] H. Lieberman and C. E. Hewitt. A Real Time Garbage Collector Based on the Lifetimes of Objects. *Communicaitons of the ACM*, 26(6), pages 419-429, 1983.

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

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

[25] M. L. Minski. A Lisp Garbage Collector Algorithm Using Serial Secondary Storage. Technical Report Memo 58 (rev.), Project MAC, MIT, Cambridge 1963.

[26] S. Nettles and J. O'Toole. Real-time replication-based garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of ACM SIGPLAN Notices, Albuquerque, New Mexico, June 1993. ACM Press.

[27] S. Nettles, J. O'Toole, D. Pierce and N. Haines. Replication-Based Incremental Copying Collection. In Bekkers and Cohen, editors. Proceedings of *International Workshop on Memory Management*, volume 637 of Lecture Notes in Computer Science, St Malo, France, 16-18 September 1992. Springer-Verlag.

[28] J. Seligmann and S. Grarup. Incremental mature garbage collection using the train algorithm. In O. Nierstras, editor. Proceedings of 1995 *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science. Springer-Verlag, August 1995.

[29] G. L. Steele. Multiprocessing Compactifying Garbage Collection. *Communications of the ACM* 18(9): 495-508, 1975.

[30] D. Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. Proceedings of the *ACM Symposium on Practical Software Development Environments*, ACM SIGPLAN Notices Vol. 19, No. 5, May 1984, pp. 157-167.