Military Technical College
Kobry El-Kobbah
Cairo, Egypt

10th International Conference
On Aerospace Sciences&
Aviation Technology

Parallel Generational Copying Garbage Collection Schemes for Shared-Memory Multiprocessors

♦Prof. Khayri A. M. Ali ♣Assoc. Prof. Fatma A. Omara ♠Eng. Osama A. Elshakankiry

Abstract

♦In this paper, an improved parallel generational copying real-time garbage collection scheme for shared-memory multiprocessors, which supports load balancing among workers, has been proposed, implemented, and evaluated. The basic idea of improvement is developed from Ali's two papers [2,3]. The scheme proposed here is a form of copying collectors that attempt to eliminate the drawback of frequently copying long-lived (stable) objects. This class of schemes is called generational-based schemes, which is based on concentrating the collection efforts on small areas of memory, so-called young generation. They reduce the need for collecting the remaining large areas of memory, old generation. A modified scheme, without real-time response, has also been implemented and compared to the real-time one. A comparative study has been done for the two schemes with other two parallel non-generational copying garbage collection schemes founded by the author in [1]. According to this comparative study, we proved that the performance of generational schemes is better than the performance of non-generational schemes. Also, the overheads occurred due to real-time response have been calculated.

1. Introduction

Storage Management in the programming environment refers to how to control the use of memory that holds representations of objects/information. It is responsible for allocating space for the object when it is created and subsequently reusing the allocated space when it is no longer needed. An efficient storage management scheme plays a crucial role in enhancing the

♦ **Prof. Khayri A. M. Ali**
Faculty of Computer Science
October University for Modern Sciences & Arts, MSA.
♣ **Assoc. Prof. Fatma A. Omara**
Computer Science Dept.
Faculty of Computers & Information
Cairo University.
♠ **Eng. Osama A. Elshakankiry**
Computer Science & Engineering Dept. Faculty of Electronic Engineering,
Minufiya University.

efficiency of a programming system. Programming languages that rely on garbage collection have existed since the late 1950's [7]. Though the benefits of garbage collection for program simplicity and robustness are well known and accepted, most software developers have continued to rely on traditional explicit memory management, largely because of performance concerns. Only recently garbage collection has been allowed to enter the mainstream and be used in large systems. Developers have been skeptical about garbage collection for two reasons: throughput and latency. That is, they fear that collection will either slow down the end-to-end performance of their systems, or induce long collection pauses, or both. Large increases in computing power have not eliminated these concerns, since they are typically offset by corresponding increases in memory requirement [6]. Lieberman and Hewitt [5] first proposed generational garbage collection techniques, but Ungar reported the first implementation [8]. Garbage collection techniques can address both performance concerns. They split the heap into generations according to object age. Concentrating collection activity on the "young" generation increases throughput, because (in most programs) young objects are more likely to be garbage, so more free spaces is recovered per unit of collection work. Since the young generation is typically small relative to the total heap size, young-generation collections are usually brief, addressing the latency concern.

The memory space, which is not accessible, is referred to as garbage and the reclamation of garbage is referred to as garbage collection (GC). The runtime system must be capable of recognizing memory shortages and reclaiming unused memory for reallocation. The following three steps are either implicitly or explicitly performed in all GC algorithms:

(1) Identification of accessible objects,
(2) Reclamation of the inaccessible memory objects,
and (3) Compacting the memory to improve locality [9].

Advanced programming environments have been implemented on different parallel architectures and it is found that the shared-memory multiprocessor is the most used architecture as a test bed. Then, it is important to develop a memory management system such that it efficiently manages the allocated storage by the parallel program. Therefore the underlying architecture would be used efficiently.

The aim of this work is to implement and compare the two generational schemes with each other and with the two non-generational copying schemes found in [1]. The same number of workers, number of memory blocks that construct the global shared heap space, data structures, locking mechanism, and synchronization mechanism have been used in our implementation. The used

simulator is a shared-memory multiprocessor simulation toolkit for Intel x86 architectures that called Augmint [4]. This simulator has been used with number of workers up to 32 to implement the concerned GC schemes. Our improved generational scheme is based on a simple parallel generational real-time garbage collection scheme for Shared-memory Multiprocessors [3], which is described in Section 2. In Section 3 our modification to improve this simple scheme by adding the load balancing to produce the new scheme is explained. In section 4 the new scheme is described. Section 5 explains how to modify the new scheme such that it becomes parallel generational load balancing garbage collection scheme without real-time response. The performance measurements and the comparisons of the schemes as well as the method used to calculate speedup are illustrated in Section 6. Finally section 7 is for conclusions and future work.

2. A Simple Parallel Generational Real-time GC Scheme

Our generational scheme is based on a simple parallel generational real-time garbage collection scheme for Shared-memory Multiprocessors [3]. It is a two generations scheme in which garbage collection is interleaved with normal program execution. In this scheme, the space devoted to the heap is dynamically allocated to workers according to their demands for free space. This space is subdivided into two generations, NEWER and OLDER. Each generation is subdivided into two semispaces of equal size, the semispaces are OLDER_OLD, OLDER_NEW, NEWER_OLD, and NEWER_NEW. Each semispace is a list of non-contiguous memory blocks. During program execution, new objects are allocated in the NEWER_NEW semispace and each accessed object in the OLDER_OLD, and NEWER_OLD is moved to the OLDER_NEW semispace. Each time a new object is allocated, an increment of scanning and copying is done. When a worker assigns a pointer to the NEWER_NEW space into a memory cell in the OLDER_NEW space, a pointer to that cell is stored in a local reference list. Elements of this list will be used as roots in the next minor cycle.

2.1 The Scheme Outlines

The idea of the scheme [3] is as follows: objects in the heap storage system belongs to one of two generations: NEWER generation or OLDER generation. Objects alive after the previous garbage collection cycles belong to the OLDER generation, whereas newly allocated objects belong to the NEWER generation. Live objects in the NEWER generation will be promoted to the OLDER generation in a so-called minor collection cycle, and garbage in the OLDER generation is collected in a so-called major collection cycle. Because most of the newly created objects have short lifetimes, fewer objects will be promoted to the

OLDER generation in every minor collection. This means that the OLDER generation space fills up much slower than the NEWER generation space and the minor collection cycles will be more frequent than the major collection cycles. In minor cycles, references from the OLDER generation to the NEWER generation as well as the set of main roots in the system will be used to identify live objects, whereas in major cycles, the set of main roots only will be used. Since this scheme is a real time one, garbage collection and maintaining references from the OLDER generation to the NEWER generation are interleaved with normal program execution, and when a collection cycle is completed, a new one will be started.

The scheme uses two semispaces for maintaining objects in each generation. For maintaining objects in the NEWER generation, we have two semispaces called NEWER_NEW and NEWER_OLD. The NEWER_NEW semispace contains newly allocated objects, whereas the NEWER_OLD semispace contains objects allocated in the previous minor collection cycle. In each minor cycle, live objects in the NEWER_OLD semispace will be copied to the OLDER generation. When all live objects in NEWER_OLD semispace have been copied to the OLDER generation, the current minor cycle is complete and another one can be started. The roles of the two semispaces NEWER_NEW and NEWER_OLD change in every minor cycle. For maintaining objects in the OLDER generation, two other semispaces called OLDER_NEW and OLDER_OLD are used. The OLDER_NEW semispace contains copied objects from the NEWER generation in all invoked minor collection cycles after the latest major collection cycle and it also contains copied objects from the current OLDER_OLD semispace. The OLDER_OLD semispace contains all objects that were alive in the previous major collection cycle. A major collection cycle can be started when all live objects in the current NEWER_OLD and OLDER_OLD semispaces have been copied to the OLDER_NEW semispace. Here, also the roles of the two semispaces OLDER_NEW and OLDER_OLD change in every major cycle.

The idea of the scheme is shown in Figure 1. Figures 1 (A)-(C) show what happens when a minor cycle is invoked, whereas Figure 1 (D) shows a major cycle. The two semispaces on the right side represent the NEWER generation, and the two semispaces on the left side represent the OLDER generation. In minor cycle, which illustrated in Figure 1 (A), new objects are allocated in the NEWER_NEW semispace and they are assumed to be live, and live objects in NEWER_OLD semispace are copied to the OLDER_NEW semispace. Copying of live objects, as well as, scanning of the copied objects are interleaved with normal program execution.

Figure 1 (B) illustrates a situation during the next minor cycle where live objects in the new NEWER_OLD semispace are copied to the OLDER_NEW semispace. It is assumed that at the end of the minor cycle shown in Figure 1 (C). The amount of used space in OLDER_NEW semispace has reached a certain threshold and a major cycle is invoked as described in Figure 1 (D), where live objects in the new OLDER_OLD and NEWER_OLD semispaces will be copied to the new OLDER_ NEW semispace, and new objects will be allocated in the current NEWER_NEW semispace. When all live objects in the OLDER_OLD and NEWER_OLD semispaces have been copied to the OLDER_NEW semispace, the major cycle is completed and then a number of minor cycles could be performed before the next major cycle.

2.2 Locking

The data structures which can be simultaneously modified by more than one worker in this scheme are:

(1) The four lists that maintain blocks allocated to each of the four semispace,
(2) The accessible objects in the OLDER_NEW, OLDER_OLD, and NEWER_OLD semispaces.

One global lock for each global list solves the problem of controlling access to such lists. For the second data structure, we would have a lock field per object (1 bit). To avoid simultaneous update of objects in the OLDER_OLD, and NEWER_OLD semispaces, every worker wants to access an old object it first checks its lock field. If it is locked (i.e., its value equal true), then worker waits, if it isn't locked (i.e., its value equal false), the worker locks it, then access it, and unlocks it again.

2.3 Synchronization

In the scheme, all workers have to synchronize at the beginning of every collection cycle. Therefore, a new collection cycle cannot start until the previous one has been completed. The question now is how to detect termination of a collection cycle. One simple way is to use a global counter that counts how many workers have completed their current collection cycle. When all workers have completed their current collection cycle, the current GC cycle is successfully terminated because the value of counter becomes equal to the number of workers in the system, and then any worker can start a new collection cycle. Then all workers can start a new collection cycle.

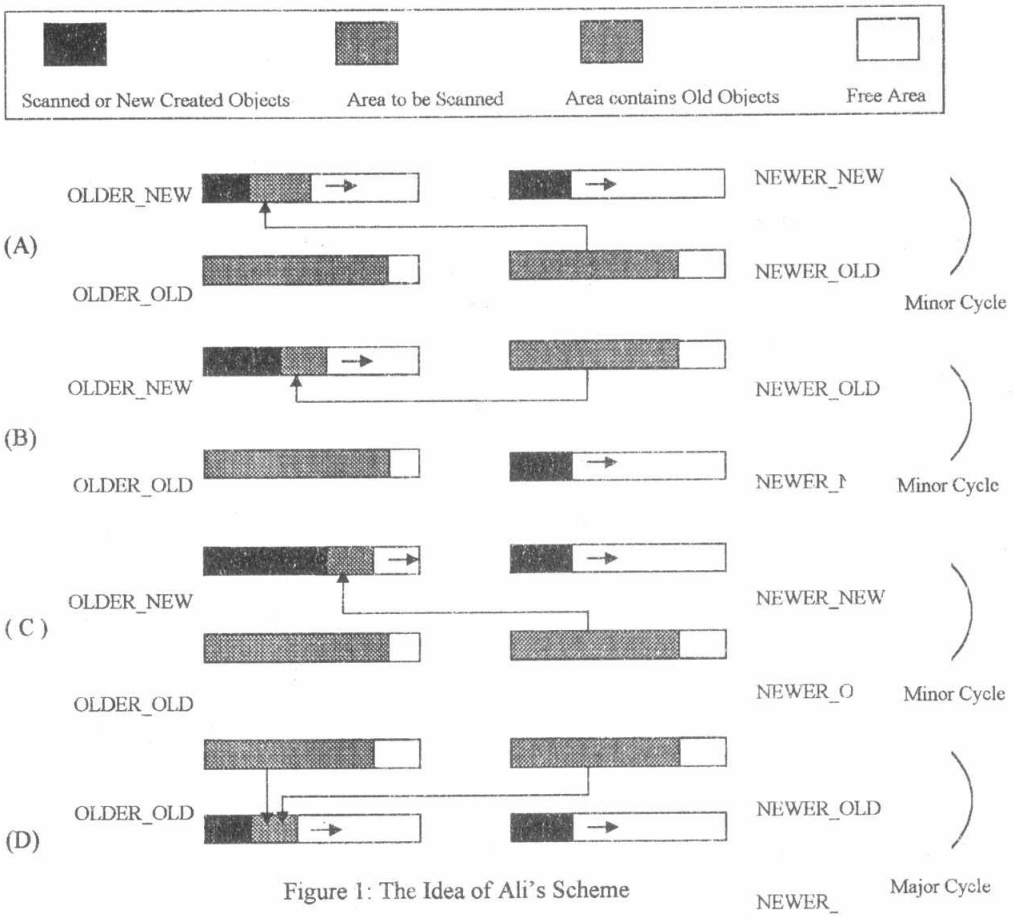


Figure 1: The Idea of Ali's Scheme

2.4 Drawback of the Scheme

In any real time garbage collection scheme (GC), garbage collection is not carried out as one atomic action while the program is halted, but instead small units of garbage collection must be interleaved with small units of program execution. The scheme interleaves collection activity with the actual work of the program. In the parallel systems where more than one worker are active at the same time, in order to guarantee each GC cycle will terminate, each creation of a new object should garbage collect some objects. In the parallel implementation, one worker could have no GC work when it creates a new object whereas

another worker has. If the former worker does not help the later, the result will be that the former worker creates a new object and does not perform any GC work. If this situation is allowed to take place, copying all accessible objects in every GC cycle could not be guaranteed. One way to solve this is that if a worker can't find any garbage collection work to do, it also stops its mutator, but this will effect the real time response of the scheme. Unfortunately, the author didn't show how this situation could be solved. When we implement his scheme, the scheme has become unstable, we proposed and implemented two solutions; the first solution, workers can allocate new object without performing any GC work, leads to not all-live objects are copied before the space is exhausted, so the GC cycle will not terminate correctly. In the second solution, workers that haven't any GC work stop their mutator also, the real time response is not as required.

3. Load Balancing

The drawback of the above scheme will be overcome by load balancing of GC work. The idea of load balancing has been taken from the parallel real-time copying garbage collection scheme with load balancing for shared-memory multiprocessor [2]. A worker with excess GC work has to make that work available to other workers and each worker that has no GC work during the creation of a new object will try to get some GC work from other worker with excess GC work. The scheme uses a unique global queue and one private queue for each worker. A worker would add work to the global queue when the queue is empty or less than a certain threshold. When a worker finishes processing all works in its own private queue, it gets some works from the global queue.
Figure 2 illustrates how a worker can maintain its excess work in its local queue in a way that simplifies sharing of that work. The worker should efficiently identify each piece of GC work, i.e., the beginning and the end of each piece of work. A pointer (S) points to the beginning of the queue of objects to be scanned and a pointer (B) points to the end of the queue. We extend the linking object, Dummy object, by another linking field to point to the end of its following unscanned area. Also, the head of the local queue, S, and B gives enough information to quickly identify each piece of local GC work. The count field indicates the number of excess pieces of GC work.

To illustrate this scheme an example is represented in Figure 3. Suppose a system having three workers W0, W1, and W2 with its global queue and with a local queue for each worker. GC work of W0 is divided into two parts: one part is global and the other part is private. W1 has all its work global; W2 has only

private GC work. The global queue has three pieces of GC work. Any worker can pick work from the global queue.

The First-B and First-E fields identify the first piece in the global queue.

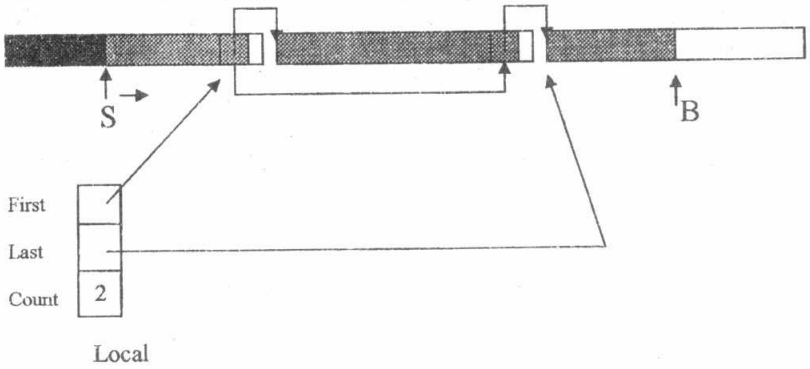


Figure 2:Managing local GC work in a queue to simplify work distribution

Linking objects at the tail of each piece of work identifies the following pieces of work. The Last field of global queue points to the end of the queue. The Count field indicates the number of pieces in the queue. The data structures used for representing GC work and the queues allow efficient moving of work from any local queue to the global queue and conversely. When a worker finishes its local work and takes work from global queue, it updates its S, B, and local queue as follows. It sets its S to the beginning of the piece of work taken. It makes the linking object of that work point to the location of B. It sets the First and Last fields of the local queue to the location of B. It increments the Count field of its local queue by 1. Then it starts scanning at S, copies at B and allocates new objects at F. First-B and First-E fields of the global queue is updated to point to the next global piece of GC work.

In the heap organization, a worker gets one free memory block at a time for allocating new objects and for copying accessible objects. Workers allocate objects in their memory blocks at different rates. The result could be a situation where some workers may have GC work and have no free space available and other workers have no GC work and have free space. In this situation the former workers should give their GC work, which requires free space, to the latter workers. W1 in Figure 3 shows a situation where it has local GC work and does not have free space to perform it. Thus, it moves its local work to the global queue. Workers with no free space available (W1) can't allocate new objects before the current GC cycle is complete. They can do any other work except allocating or copying objects. Workers with free space available should speed up

completion of the current GC cycle by concentrating on processing the available GC work, in order to minimize the idle time of the other workers.

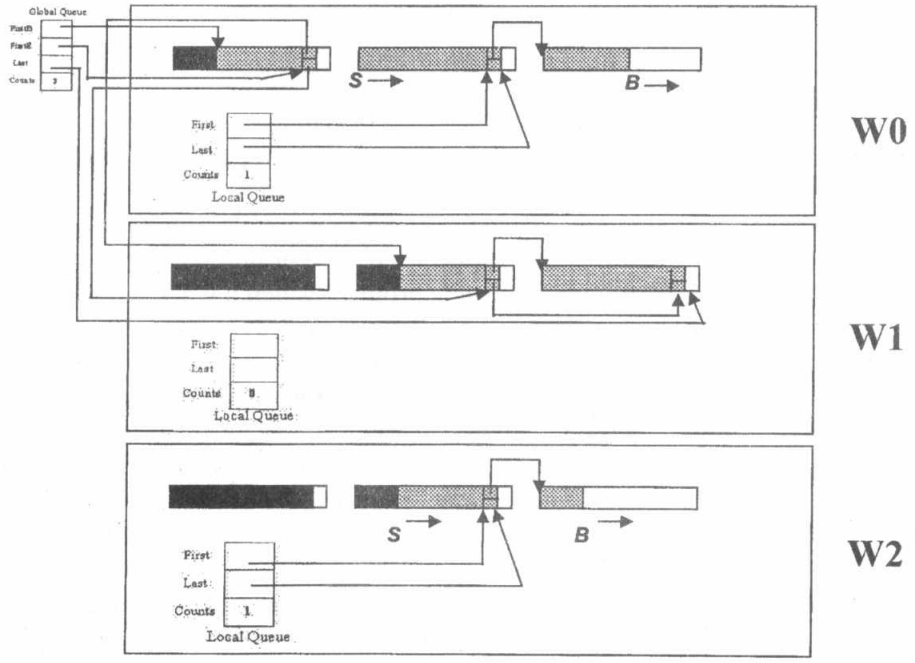


Figure 3:Local and Global GC work for a system with 3 workers W0, W1, and W2

4. A Parallel Generational Copying Real Time GC Scheme

This scheme is proposed according to the work in this paper, the idea of this scheme is exactly as the idea that has been described in the simple parallel generational real time garbage collection scheme for shared memory multiprocessors, except that the scheme supports dynamic load balancing. Dynamic load balancing has contributed by:

(1) guaranteeing garbage collection progress in every allocation of a new object,

(2) guaranteeing successful termination of every GC cycle,

and (3) improving efficiency by making workers that have no computation work due to lack of parallelism in the program, perform most of the GC work, leaving the other workers to concentrate on computation work..

Each time a new object is allocated, an increment of scanning and copying is done. The scanning and copying work is balanced between workers to guarantee garbage collection progress in every allocation of a new object. Workers that

have work and have not enough free memory space to perform the work will give their work to workers having free space.

4.1 Synchronization

In our scheme, all workers have to synchronize at the beginning of every collection cycle. A new collection cycle cannot start until the previous one has been completed. The question is how to detect termination of a collection cycle. One simple way is to use a global counter that counts how many workers have completed their current collection cycle. A worker that wishes to start a new GC cycle signals all other workers to synchronize. When a worker detects the signal, it will try to finish all available GC work as long as it has free space available. When a worker has no more free space available and has private GC work, it puts its work in the global queue and then it replies to the synchronizer. When all workers have replied to the GC invoker and there is no remaining work in any of the local queues and the global queue, the current GC cycle is successfully terminated. The counter value will be as the number of workers in the system, and then any worker can start a new collection cycle.
The successful termination could fail when we reach a situation in which there is GC work in the global queue and none of the workers has enough free space to process it. To avoid this situation we must estimate the amount of GC work that should be done in every allocation of a new object at run time to guarantee successful termination.

4.2 GC Work per New Allocation

The amount of GC work that should be done in every allocation of a new object to guarantee successful termination of each GC cycle will be estimated at run time as follows:
Assuming that **Omax** is the maximum object size of the allocated objects, and **Omin** is the minimum object size of the allocated objects. Recall our assumption that the heap space contains **(N+M)** memory blocks where NEWER generation contains **N** memory blocks, OLDER generation contains **M** memory blocks, where M is much larger than N, and the two semispaces of each generation are equal in size. At termination of a minor GC cycle, **L1** is the total amount of free space in the current NEWER_NEW semispace. After flipping of the NEWER generation semispaces, in the worst case **((N/2)-L1)/Omin** objects should be copied and scanned when **(N/2)/ Omax** new objects are allocated. That is **(Omax / Omin)*(((N/2)-L1)/(N/2))** objects should be scanned in every new objects allocation. Upon doing this, the new GC cycle will successfully terminate. In the real system, a high percentage of allocated objects are garbage and not all old objects have minimum object size **(Omin)** and not all new

objects have maximum size **(Omax)**. Thus, using this estimation, it is guaranteed that every minor GC cycle will successfully terminate.
By the same way, at termination of a major GC cycle, **L1** is the total amount of free space in the current NEWER_NEW semispace and **L2** is the total amount of free space current OLDER_NEW semispace. After flipping of the NEWER and OLDER generations semispaces, in the worst case, **(((N/2)-L1)+ ((M/2)-L2))/Omin** objects should be copied and scanned when **(N/2)/ Omax** new objects are allocated. That is **(Omax / Omin)*((((N/2)-L1)+((M/2)-L2))/(N/2))** objects should be scanned in every new objects allocation.

4.3 Outline of The Scheme

The space devoted to the heap is virtually subdivided into four semispaces: OLDER_OLD, OLDER_NEW, NEWER_OLD and NEWER_NEW. The space of each semispace is dynamically allocated to workers according to their demands for free space. During program execution, new objects are allocated in the NEWER_NEW semispace and each accessed object in the OLDER_OLD, and NEWER_OLD is copied to the OLDER_NEW semispace, and references from the OLDER generation to the NEWER generation are maintained. Each time a new object is allocated, an increment of scanning and copying is done.
Each worker allocates new objects in its memory blocks in the NEWER_NEW semispace and copies uncopied objects to its memory blocks in the OLDER_NEW semispace. Each worker maintains its private work using its **S** and **B** pointers, and its local queue. This queue represents unprocessed local GC work. When a worker assigns a pointer to the NEWER_NEW semispace into a memory cell in the OLDER_NEW semispace, a pointer to the cell is stored in a local reference list. Elements of this list will be used as roots in the next minor collection cycle. When a worker allocates a new object, it scans some elements of its private queue and reference list created in the previous cycle, if there is any.

Whenever a worker having excess work discovers that the global queue is empty (or less than a certain threshold), it moves some of its excess work to the global queue. Each worker could check the global queue at the end of each allocation of a new object. When a worker allocates a new object and cannot find private GC work to do, it will try to take work from the global queue, if there is any. When a worker demands a new memory block and there is private work to be scanned in the worker's current memory block, that work is saved in either the global queue or in the worker's private queue. When a worker cannot get a new memory block, i.e., memory space in the current NEWER_NEW semispace has been exhausted, it will try to start a new GC cycle. If a worker has private GC work, it will move that work to the global queue. Then the worker will decide

whether it is a major or minor one and automatically signal all workers to synchronize for flipping, if another worker has not done that yet. When a worker detects the flipping signal, and finds GC work to be done, it will first try to finish its private GC work and then the global queue. Any worker that cannot process its private GC work due to shortage of free space will save the remaining work in the global queue. Then the worker replies to the GC invoker and waits.

When all workers have acknowledged the flipping, the GC invoker atomically reverses the roles of the two semispaces and then signals all workers to start a new collection cycle. When a worker detects a start signal, it copies objects directly reachable from its roots to its newly allocated memory blocks replies back, and resumes its normal program execution.

4.4 Idle Workers

The GC work in the system will be dynamically moved from busy workers to idle workers. Workers performing memory operations and having excess GC work move some of their local work to the global queue and each idle worker that does not have GC work will take from the global queue.

5. Parallel Generational GC with Load Balancing

Because some systems don't need real-time or even interactive response, disruptive behavior is not important, Therefore, our scheme has been changed such that the real time response is eliminated (i.e., it becomes a stop-the-world collector). According to this modification, GC carried out as one atomic action while the mutator is suspended. When a worker cannot get a new memory block (i.e., memory space in the current NEW semispace has been exhausted), this worker will start a new GC cycle by signaling all other workers, suspending its mutation, and waiting. When a worker finds the signal, it suspends mutation and waits, this will be happen for all workers in the systems. After all workers suspend, the first worker, which wishes to start a new GC cycle reverses the role of the two semispaces. Then, all workers including the first one copy the live objects starting from their root set. When all live objects have been copied (i.e., Global queue and all Local queues become empty), all workers are allowed to resume their mutation. Because the mutator is halted during the GC cycle, then it leads to long and unpredictable delays. In the opposite, the overheads that could be occurred due to real-time response are eliminated, so that the total execution time of the program is decreased. Therefor, the modified scheme (without real time response) is preferred if the mutator allows long delays without making any effect on it.

5.1 Load Balancing

This modified scheme uses a unique global queue and one private queue for each worker. This is exactly what is happenning in the real time scheme, such that the load balancing can be implemented. A worker would add work to the global queue when the queue is empty or less than a certain threshold. When a worker finishes processing all works in its own private queue, it gets some work from the global queue. Figure 5 illustrates how a worker can maintain its excess work in its local queue such that, this work can easily be shared among other workers. The worker should efficiently identify each piece of GC work (i.e., the beginning and end of each piece of work), where **S** points to the beginning of the queue of objects to be scanned and **B** points to the end of the queue. The head of the local queue, **S**, and **B** gives enough information to quickly identify each piece of local GC work. The count field indicates the number of excess pieces of GC work. If a worker has finished processing its local GC works, and there is GC works in the global queue, then it takes work from global queue and processes it. When all live objects have been copied (i.e., Global queue and all Local queues are become empty), the mutator is allowed to resume its work. Because of the load balancing among workers, no worker will be idle waiting other workers to finish their GC works, instead they will help each other by distributing GC work among them.

5.2 Synchronization and Locking

As in the real-time scheme; all workers have to synchronize at the beginning and the ending of every GC cycle. When a worker has no more free space available to complete its mutation, it wishes to start a new GC cycle, it stops its mutation, signals all other workers to synchronize. When a worker detects the signal, it also suspends its mutation and replies to the GC invoker. When all workers have replied, they start copying live objects staring from their root sets. After all live objects have been copied, all workers are allowed to resume the mutation. In seek of locking problem, one global lock for each global list solves the problem of controlling access to such lists, also a lock field per objects (1 bit) is used to control multiple simultaneous accesses to objects by more than one worker at the same time.

6. Performance Measurements

A comparison among the two proposed schemes and the two non-generational schemes has been made to measure the total execution time, as well as, the speedup.

6.1 Total Execution Time

The simulator's results of the schemes with respect to the total execution time are given in Table 1. All schemes are supported with load balancing. These results have been measured up to 32 workers.

Table 1.

processors	Non-generational copying GC	Non-generational RT copying GC	Generational copying GC	Generational RT copying GC
1	78.788807	152.664245	74.946954	123.415635
2	56.302677	91.331063	47.320733	72.111341
4	29.013456	46.282691	26.431811	34.942237
8	15.589865	24.519489	12.974917	17.259232
16	10.139065	16.683999	6.580165	9.038633
32	6.981663	14.901922	3.466478	7.479359

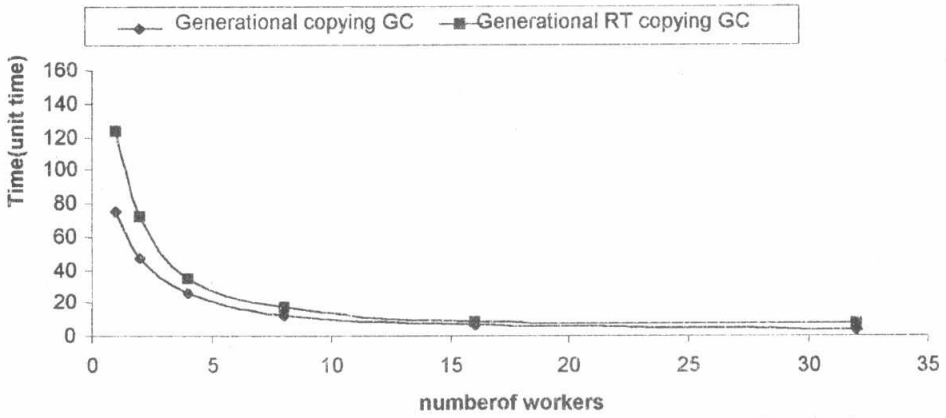


Figure 4:Real-time and non real-time parallel generational copying GC with load balancing

According to the comparison results of real-time and non real-time parallel generational copying GC schemes with load balancing, which are shown in Figure 4, the non real-time scheme shows better performance than the one with real-time. This result is natural since supporting real-time response incur more

run-time overhead. So in the systems those don't necessary need interactive or real-time response the non real-time scheme could be used.

According to Figure 5 and Figure 6, it has been found that generational copying garbage collection has a better performance than non-generational copying garbage collection. The major inefficiency in non-generational copying garbage collection is that the system must copy all live data at a collection. Most objects die before a collection and never need to be copied. Objects that do survive are copied at every collection, over and over, and the garbage collector spends most of its time copying the same old objects repeatedly. Generational copying collection avoids much of this repeated copying by segregating objects into multiple areas by age, and collecting areas containing older objects less often than the younger ones.

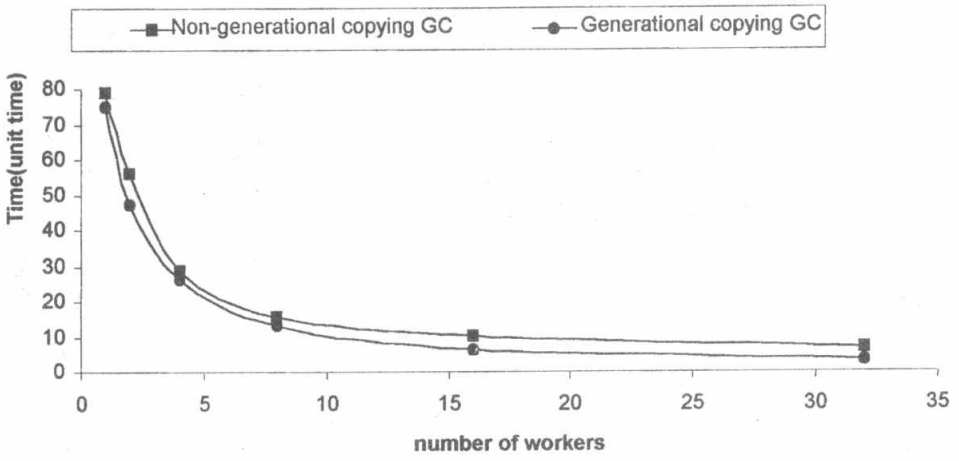


Figure 5:Non-generational and generational parallel copying GC with load balancing.

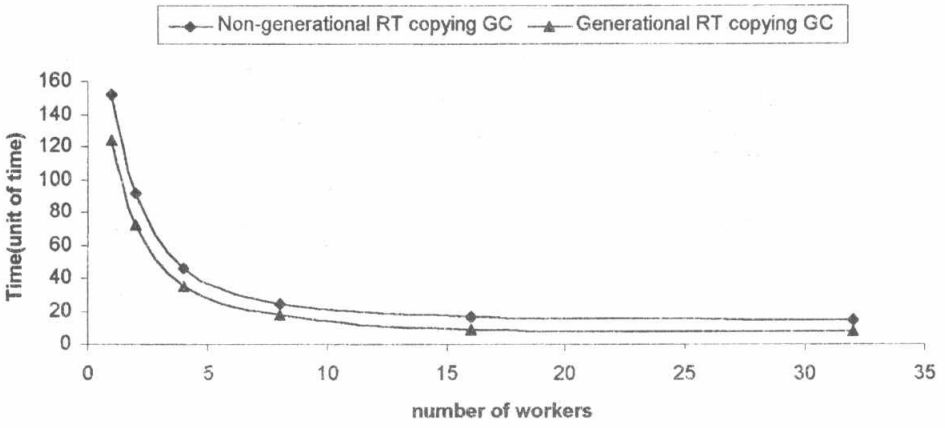


Figure 6:Non-generational and generational RT parallel copying GC with load balancing.

Once objects have survived a small number of collections, they are moved to a less frequently collected area. Areas containing younger objects are collected quite frequently, because most objects will generally die quickly, freeing up space; copying the few that survive does not cost much. These survivors are advanced to older status after a few collections, to keep copying costs down.

6.2 Speedup

The speedup is measured by the following equation.

$$\text{Speedup} = \frac{sgctime}{pgctime}$$

Where:

sgctime is the sequential garbage collection scheme time , and
pgctime is the parallel garbage collection scheme time

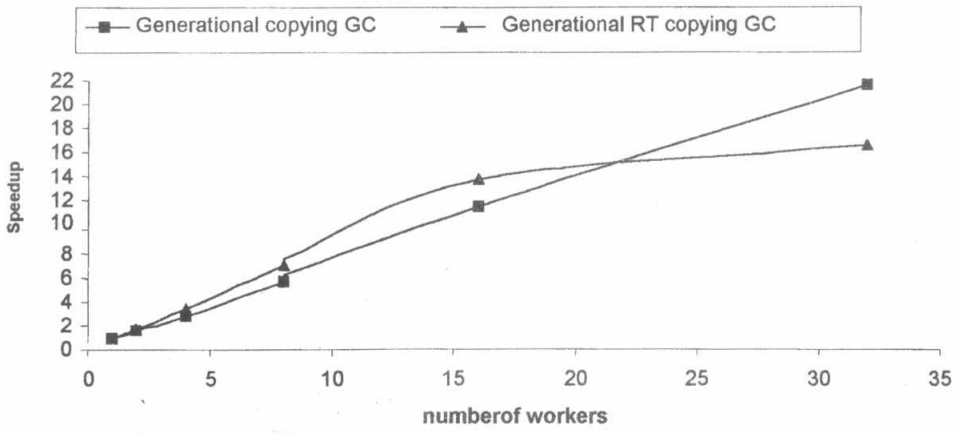
The calculated speedup of the four schemes are given in Table 2

Table 2.

processors	Non-generational copying GC	Non-generational RT copying GC	Generational copying GC	Generational RT copying GC
1	1	1	1	1
2	1.399379411	1.671547883	1.583807968	1.711459436
4	2.715595378	3.298517042	2.835483123	3.531990096
8	5.053847933	6.226240889	5.776295448	7.150702592
16	7.770815849	9.150338897	11.38982898	13.65423676
32	11.285106	10.244601	21.62049031	16.50083049

The speedup results of the real-time and non real-time parallel generational GC schemes with load balancing are shown in Figure 7. According to the results, it has been found that the real time scheme satisfy better results than the non real-time scheme for small number of workers. The overheads have been occurred in the real time scheme when the number of workers increases. This is because of the synchronization overheads.

Figure 7:Real-time and non real-time parallel generational copying GC with load balancing.



From Figure 8 and Figure 9 we can see that generational copying garbage collection schemes satisfy better results than non-generational copying garbage collection.

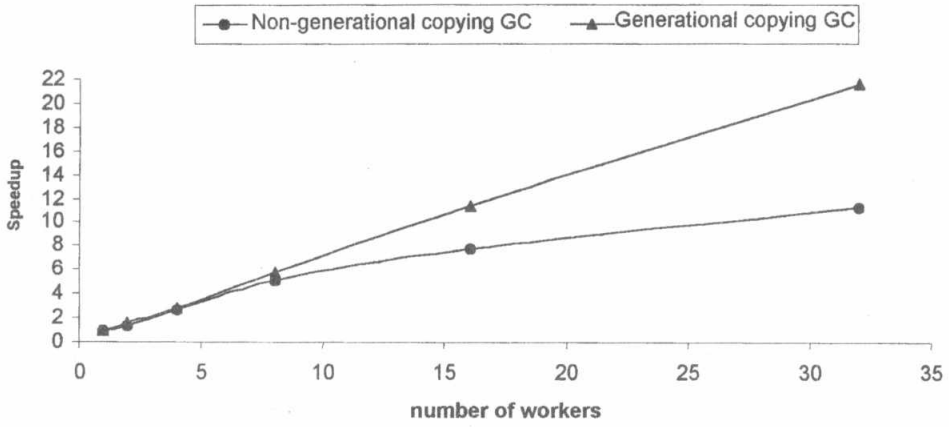


Figure 8:Non_generational and generational parallel copying GC with load balancing.

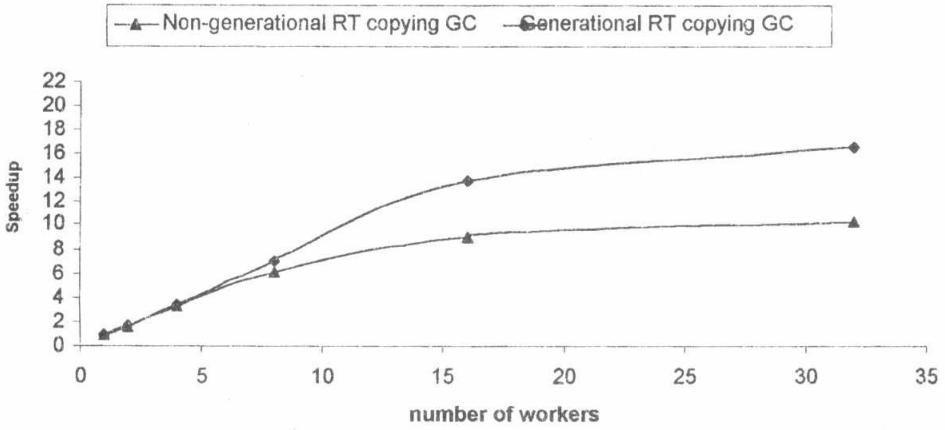


Figure 9:Non_generational and generational RT parallel copying GC with load balancing.

7. Conclusion

In this paper we proposed an improved parallel generational real-time garbage collection scheme for shared memory multiprocessors. This scheme as will as, a parallel generational garbage collection scheme for shared memory multiprocessors have been implemented and evaluated. The basic idea of improvement is developed from Ali's two papers "A simple Parallel Generational Real-time Garbage Collection Scheme for Shared-memory Multiprocessors" [3], and "A Parallel Real-time Garbage Collection Scheme for Shared-memory Multiprocessors" [2]. A comparison between the first two schemes has been made. Because the real-time response is not always required, so if we use a system that doesn't need interactive or real-time response, the modified scheme that does not support a real-time response can be used. A comparison between the two generational schemes and with non-generational schemes is also made. We prove that generational schemes have better performance than those non-generational schemes.

References

1. Khayri A. M. Ali, et al., " A Comparative Study of Parallel Copying Garbage Collection Schemes for Shared-memory Multiprocessors", to be published.

2. Khayri A. M. Ali, "A parallel Real-Time Garbage Collection Scheme for Shared-Memory Multiprocessors," In the Proceedings of the 31st Annual Conferences on Statistics, Computer Science, and Operating Research, Cairo, December 1996.

3. Khayri A. M. Ali, "A Simple Parallel Generational Real-Time Garbage Collection Scheme for Shared-Memory Multiprocessors, " The Egyptian Computer Journal, ISSR, Cairo, JNIV, Vol.24, No.2, 1996.

4. Augmint: A multiprocessor Simulation Toolkit for Intel x86 Architectures. Source + executable + docs: http://www.csrd.uiuc.edu/iacoma.

5. H. Lieberman and C. E. Hewitt, "A Real-Time Garbage Collector based on the Lifetimes of Objects," Communications of the ACM, Vol.26, No.6, PP 419-429, 1983.

6. Tony Printezis and David Detlefs, "A Generational Mostly-concurrent Garbage Collector," SMLI TR-2000-88, June 2000.

7. Herbert Stoyan, "Early Lisp History (1956-1959)," Web version: http://www8.informatik.uni-erlangen.de/ html/lisp/histlit1.html.

8. D. M. Ungar, "Generation scavenging A non-disruptive high performance storage reclamation algorithm," ACM SIGPLAN Notices, Vol.19, No.5, PP 157-167, April 1984.

9. Paul R. Wilson, "Uniprocessor Garbage Collection Techniques," Tech. Rep. University of Texas, January 1994.