TR-354

Piling GC -Efficient Garbage Collection for AI Languages-

by

K. NakaJima

March, 1988

Piling GC

– Efficient Garbage Collection for AI Languages –

Katsuto Nakajima
CSNET: nakajima%icot.jp@relay.cs.net,
ARPA: nakajima%icot.uucp@eddie.mit.edu,
UUCP: ihnp4!kddlab!icot!nakajima

Institute for New Generation Computer Technology (ICOT)
21F Mita Kokusai Building, 1-4-28 Mita,
Minato-ku, Tokyo, 108, JAPAN
Phone: 03(456)3069 Telex:ICOT J32964

Abstract

One of the most critical points for efficient implementation of language systems is in the efficiency of allocation and reclamation of memory area for data object. As in languages such as Lisp or GHC, stack-like memory allocation as in Prolog cannot be applied and general heap-like memory management is required. Hence, some garbage collection (GC) mechanism is indispensable. Widely used copying GC is an efficient method because garbage is not accessed during GC. Its performance, however, decreases when active objects occupy a great deal of memory area. Another defect of the method is that half of the memory space must always be reserved for GC during normal processing.

This paper proposes a new GC method based on copying GC, which solves the above-described problems of original copying GC; it is efficient even when the density of active objects is high; it also solves the memory space efficiency problem. A study of combining this method with reference count GC is also presented. A scheme for executing this GC in parallel on a shared memory multiprocessor system is given.

1. Introduction

One of the most noticeable features of programming languages such as Lisp, Prolog and GHC [7] is that the system substitutes memory management for the programmers. Using these languages, AI programmers are not concerned with memory allocation and can devote themselves to encoding the algorithm itself.

Prolog and GHC have the write-once characteristic. Memory cells are allocated and used even for storing temporary results. Even in impure Lisp, once allocated data structures are not reused very often. Hence, memory cells run out very quickly unless they are collected and reused quickly after they become garbage. In Lisp and GHC systems, the memory management scheme is particularly important because there is no other efficient way for memory reclamation like the stack mechanism in a Prolog system [8].

If the system can collect garbage memory cells and reuse them repeatedly, the working set of the memory can be kept small. As a result, the execution speed can be improved on a machine which has hierarchical memories such as cache memory.

The key issues in selecting or designing a garbage collector are:

- Time efficiency (time required to collect a unit of garbage)
- Non-stop GC or stop GC (real-time GC or not)
- Memory efficiency

In a real-time system where the response might be the most important feature, non-stop GC is preferable, even at the cost of sacrificing the time efficiency to some extent. However, on a special machine for the language, it is natural to give priority to the time efficiency, although efforts should be made to shorten the stop period.

Memory efficiency may also be a major problem in copying GC [1], which wastes half of the memory space, and in reference count GC, which uses an extra counter field for all objects. If copying GC is adopted on a real memory machine, wasting half of the memory is a serious problem. Even on a virtual memory machine, although wasting half of the memory space may be a minor matter, the degradation of the performance is not small when a large program is executed because the working set is doubled.

In order to improve time efficiency, at least two issues must be considered. One is to reduce the costs of operations related to reclamation, and the other is to obtain high locality of memory access. From the view point of memory access locality, incremental garbage collectors using the reference count have the best efficiency among various garbage collectors [3]. However, the reclamation cost of the reference count is relatively high in general. Furthermore, the reference count has the serious defect of not being able to collect cyclic garbage (garbage objects which are linked to each other).

Some garbage collectors which traverse active (live) objects can collect cyclic garbage. *Sweeping GC* [6] and *copying GC* [1] are popular. As sweeping GC sweeps all the memory space once or more after marking active objects, the cost of memory reference is very high. Copying GC only accesses active objects in the condemned memory space (called *from space*) although the copied objects in the new space (called *to space*) are also accessed twice by sweeping. The number of memory accesses is almost proportional to the number of active memory cells and the locality of the memory references is much better than that of sweeping GC. However, if a large part of the memory is occupied with active objects as in a practical situation in which a large application program is running, even copying GC accesses many memory cells over the whole of memory, and its performance is no better than sweeping GC.

This paper introduces a garbage collection method called *Piling GC*. It is based on copying GC, and is aimed at achieving great efficiency even in a system in which the density of active objects is high. Its basic idea is similar to that of generation GC [5]. The underlying assumption of Piling GC is that "An object which survived one garbage collection will probably survive the next." In garbage collection, active objects are piled in a region which will not be scavenged until all the memory is occupied with the piled objects. This method is superior in time and memory efficiencies to copying GC. Chapter 2 presents the principles of Piling GC, and Chapter 3 describes its implementation.

The combination of reference count GC and Piling GC might be one solution for efficient memory management, because it can make the most of the access locality in the reference count scheme and Piling GC can compensate for the defects of the incomplete reclamation of reference count. Chapter 4 describes a case study of combining it with reference count GC.

A shared memory multiprocessor with coherent cache memory attached to each processor is possibly a suitable architecture for the parallel execution of AI languages such as GHC. Efficient garbage collection is more difficult on such machines. Chapter 5 presents a parallel method of Piling GC.

2. Principles

Piling GC is based on the following observation.

> If a region in which most of the objects are active could be isolated from the remaining region of heap memory, and if a garbage collector could collect garbage only from the remaining region, the copying GC scheme would be very efficient in terms of the garbage collecting rate, that is, the number of collected memory cells with a unit number of memory accesses.

In Piling GC, active objects are selected from the *work area* which is used for object creation, and are piled into an isolated region called the *piling area*. If the heap memory can be split in the work area and the piling area by an arbitrary boundary, the piling area can be allowed to grow until it occupies all the heap memory. When the heap memory is filled with the piled objects, another global garbage collector condenses the piling area. As it takes a relatively long time for the piling area to occupy whole of the heap, we can use a slower garbage collector for global GC.

To make it possible to collect garbage only in the work area, reference pointers from the piling area to the work area must be memorized. These memorized pointers will become marking roots in Piling GC and be updated correctly to point to the new locations where the referenced objects are moved to. This management of the pointer is called *trailing*.

Piling GC is similar to generation GC in principle. The basic idea of generation GC came from the fact that "Newer objects easily become garbage." The objects are grouped by their birth time into several *generations*. The garbage collector visits younger generations more frequently than older generations. Some contiguous generations are occasionally merged to make one generation a reasonable size for GC. In the generation GC scheme, as the newest objects, which will live a very long time, do not move to the older generation soon, they are visited by GC more than the shorter-life objects in the older generations.

The basic idea of Piling GC comes from the assumption that "The objects which survived one GC will probably survive the next GC." It seems to be the best if the objects are grouped by the number of GCs they survived. The heap memory is divided into several generations according to the number of survivals. Active objects are shifted to their senior generations after each or several GCs. The trailed pointers should be memorized individually at each generation that they are referring to. It is as expensive as generation GC to trail and maintain the pointers between the generations.

The simplest implementation of this idea is the Piling GC, in which only two generations are used, the work area and the piling area. Active objects are piled into the piling area at each GC. All the trailing information can be discarded after each Piling GC because the objects referenced from the trailed pointers are piled and trailing is no longer necessary for them.

The following chapter explains the implementation for the simplest Piling GC.

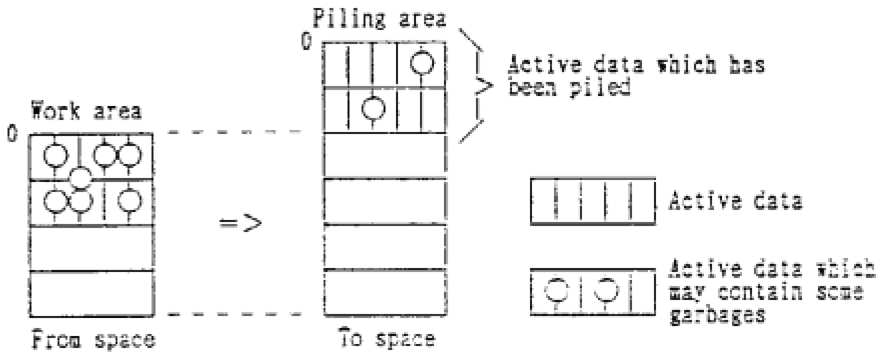
Piling area
0
Active data which has been piled
Work area
0
=>
Active data
Active data which may contain some garbages
From space
To space

Figure 1: Piling Area and Work Area

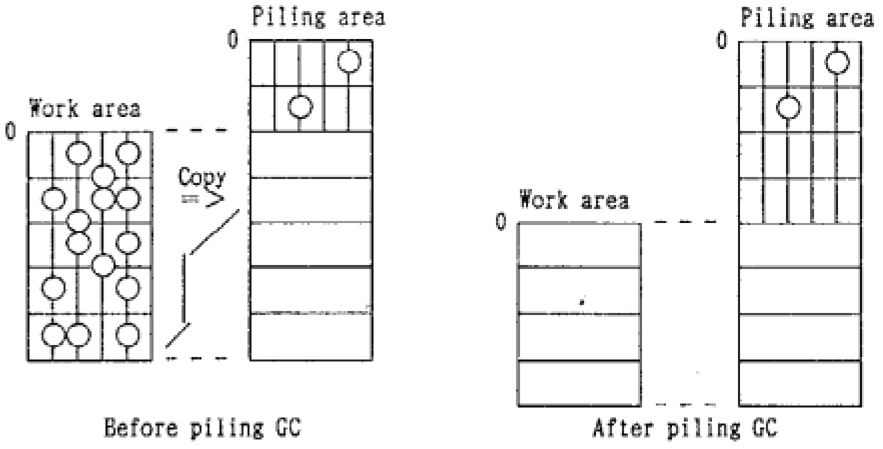

Figure 2: Piling Area and Work Area before and after Piling GC

3. Implementation

3.1 Memory Allocation

For simplicity of explanations, we assume that memory space for the heap can be divided into subspaces called *areas* and that physical pieces of memory (pages) are assigned to the contiguous address of each *area*.

First, the memory pages are divided equally into two areas, one is the work area (which corresponds to *from space* in conventional copying GC), and the other is the piling area (*to space*). Objects are created in the work area until the pages for the area is exhausted. When the work area becomes full, only the active objects are copied to the piling area. After copying, the top pointer of the piling area is set to the next to the region to which the active objects were moved. This region is excluded from the next GC. The memory pages allocated to the work area and to the region beyond the top of the piling area are divided again equally into two, one for the work area, and the other for the top of the piling area so that the piling area never overflows with copied objects in the next Piling GC. Again, new objects are created in the work area (Figure 1).

In the next GC, new active objects in the work area are piled to the top of the piling area and

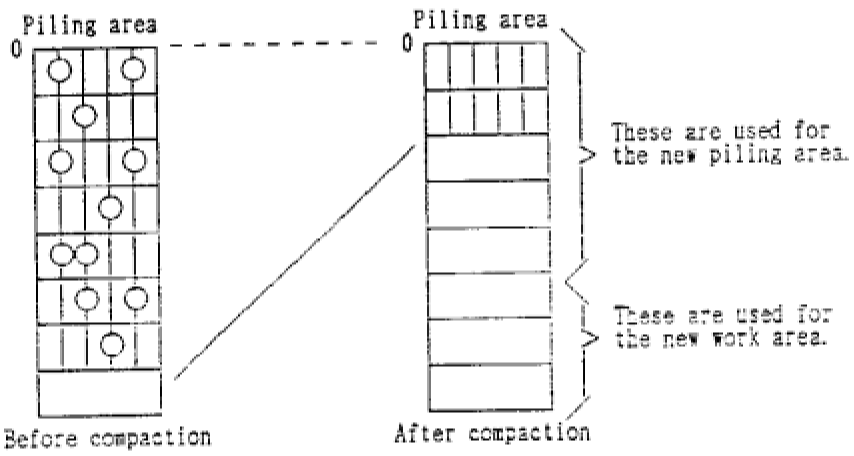
Piling area
0
Piling area
0
These are used for the new piling area.
These are used for the new work area.
Before compaction
After compaction

Figure 3: Global Compaction GC in the Piling Area

merged into the previously piled region by proceeding the top pointer of the piling area (Figure 2). This process is repeated until the number of memory pages left becomes less than some predetermined threshold number. When this state is reached, the piling area is condensed by a global garbage collector,[1] as the piling area may contain some garbage as the result of execution (Figure 3). When the work area becomes very small, most of the newly created objects will survive and be piled because Piling GC is invoked before they becomes garbage. This causes a degradation of efficiency of the reclamation in total. To prevent this, it is better not to set the threshold number too low (eg. 1%~5% of the total physical pages).[2]

3.2 Trailing

The key point of Piling GC is to avoid accessing objects in the piling area as far as possible. If the objects in the piling area are allowed to be marked, all the objects except for some possible garbage are accessed at least once for each object. Therefore, the piling area should be isolated from the work area.

In Piling GC, the reference pointers from the piling area to the same area can be left unchanged. The pointers referring from the work area to the same area are maintained correctly by GC as in conventional copying GC. Only the pointers referring across the two areas should be considered. In Piling GC, these pointers are treated as follows.

(a) Pointers from the work area to the piling area

As the referenced object can be treated as *marked*, the marking path is terminated. No operation or access to the referred object is done, because an address comparison is enough to determine whether the object is in the piling area or not.

(b) Pointers from the piling area to the work area

All the pointers in the piling area are considered *active*. Therefore, the pointers referring to the work area should be the marking roots. They should be memorized when the pointers are

[1]sweep compaction GC is available

[2]it can be used as marking stack in sweep compaction GC

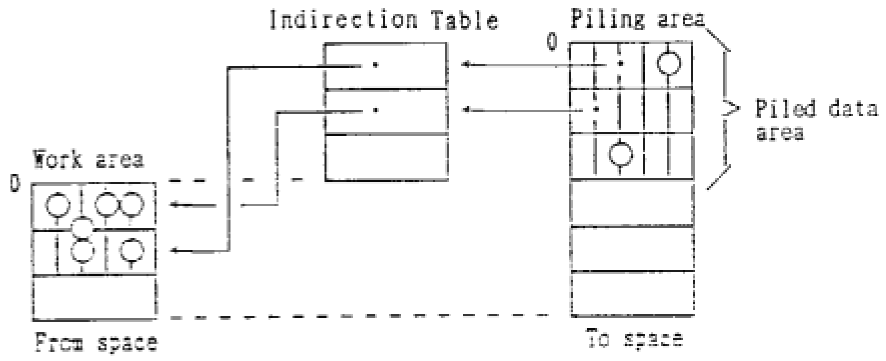

Figure 4: Trailing by *Indirection Table*

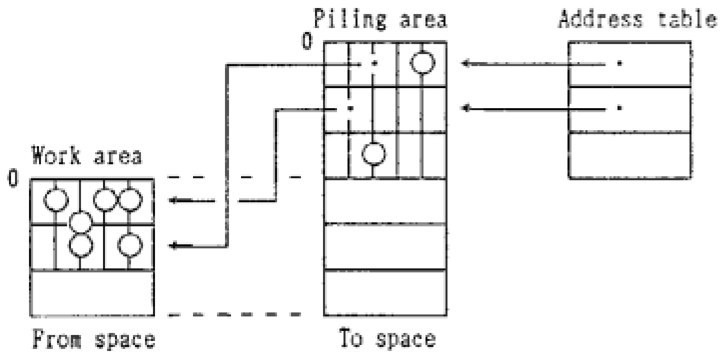

Figure 5: Trailing by *Address Table*

created, and they should be updated at GC because the objects referred to by them move to the piling area.

In Lisp or GHC, there are generally fewer pointers referring from old objects to new ones than in the opposite direction. Therefore, the number of pointers of case (b) should be very small, and the memory space for memorizing them, called ***trail table***, must be negligible in total.

There are two possible ways to trail the pointers.

(1) *Indirection Table*

The trail table keeps the indirection pointers from the piling area to the work area (Figure 4).

(2) *Address Table*

The trail table keeps the address of the pointers in the piling area (Figure 5).

(1) is same as the *entry table* proposed in [5]. The advantage of this method is that if an entry of the table is no longer in use, the entry can be nullified to avoid marking unnecessary objects through the trailed pointer. One of the defects is that the reference through the pointer must pay one indirection overhead. Another and possibly more serious defect in normal execution is that during the reference through a pointer which may be a trailed one, the check is always necessary whether the reference is

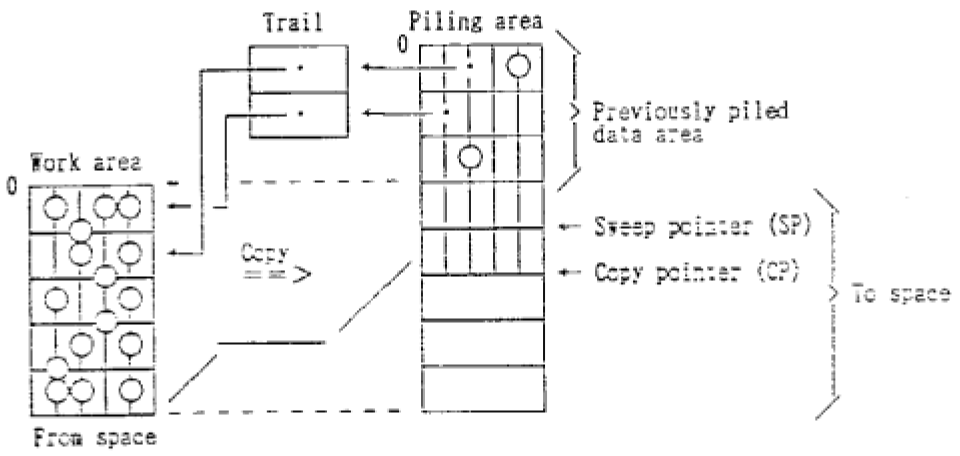

Figure 6: CP and SP during Piling GC

an indirection path or not. However, if an "invisible pointer" mechanism is already used in the system and it is also available in the trail table, no extra overhead will arise.

The advantage of (2) is that there is no overhead at all on referring through the trailed pointers in normal execution. However, it is difficult to nullify a trail table entry even when a trailed pointer is found to be no longer necessary.

If we can use the invisible pointer in the trail table, we will choose method (1), although it is necessary to consider the machine architecture, the programming language, its implementation and the dynamic characteristics of the program (see Section 4.1).

In either method, possibly the heaviest overhead in normal execution for trailing is the cost of checking whether a pointer stored into an memory object should be trailed or not when it is doubtful. The condition to trail is:

- If and only if a pointer referring to the work area is stored in the piling area.

The physical memory pages for the trail table are allocated dynamically. Any address space (area) will work well if it can be identified as the marking root. The table entries are assigned one by one at each trailed pointer creation. If there are no more memory pages for a new trail entry, we split the same number of pages from already allocated memory pages in both the work area and the piling area. Trail entries are never accumulated because they are discarded after each Piling GC.

3.3 Procedure

The Piling GC procedure is almost the same as that of copying GC. Some working pointers for marking process and two pointers, copy pointer (CP) and sweep pointer (SP), are used. The CP is the top pointer of the piling area (Figure 6).

The following is the procedure of Piling GC.

(1) After initializing the SP to the top of the piling area, one marking root is picked up from a possibly fixed region in the work area and the marking phase starts.

(2) If the root is a pointer referring to an object in the work area, the referenced object is copied to the top of the piling area indicated by the CP. The CP is incremented by the size of the

copied object. The root pointer itself is updated to point to the new location for the object. The old location is also modified to be a pointer pointing to the new location for the visitors to it afterward.

(3) If the root is a pointer referring to the work area and the location referenced by the pointer contains another pointer pointing to copied object in the piling area, the root pointer is updated to point to the new location of the object.

(4) If the root is a pointer referring to the piling area, no operation is done.

(5) After marking with one of the roots, the next marking root is picked up:

  (i) From the location indicated by the SP,

  or, if the SP has already reached the same position as the CP,

  (ii) From the trail table,

  or, if the trail table has already been marked out,

  (iii) From the fixed region in the work area, if any.

(6) Garbage collection finishes if there is no marking root.

(7) The physical memory pages for the trail table, the work area and the region beyond the CP (the top of the piling area) are deallocated. The deallocated pages are divided equally into two, one for the work area and the other for the top of the *new* piling area.

The steps above are almost same as that of copying GC except for (4), (7) and (ii) of (5).

4. Combination with Reference Count GC

The overall performance of the system employing Piling GC depends on the speed at which the work area is consumed. Slower the consumption speed, the better the performance. Thus the introduction of an incremental GC on top of Piling GC can make the overall performance better. Moreover, an optimized but incomplete incremental GC such as one using the Multiple Reference Bit technique [2] can be employed, since Piling GC can collect garbage cells which the incomplete incremental GC has left unreclaimed.

This chapter discusses the problems and solutions for adopting Piling GC in systems with reference count GC.

4.1 Trail

Trailing has been considered as a cheap operation and to use only a little of memory because it is supposed that few pointers are created which might be trailed. However, if reference count GC reclaims and allows the objects in the piling area to be reused, the trailing might become very frequent.

If the memory cells in the piling area are reused, the following problems arise:

(1) Frequent trail check:

If a memory cell is reused to store a *new* object, the object must be checked every time when it is a pointer.

(2) High possibility of trailing:

If a memory cell is reused to store a pointer, there is a high possibility that the pointer is trailed.

(3) Large memory consumption for the trail table:

In addition to (2), more than one trailing is done for the same memory cell in the piled region if the cell is reused again and again. If the trail entry itself cannot be reused for the same cell, the trail table may grow infinitely.

To reuse the trail entry, hashing or some other technique must be adopted, and it raises the cost of the trailing operation. Compaction for the trail table may be more realistic than avoiding double entries for the same cell. Compaction can be done at a cost of about 2·N times of memory access, where N is the number of entries in the trail table, because it is possible to mark the location pointed from the trail table.

One drastic solution for this problem is, of course, not to reuse the object cells in the piling area. As the abolished cells are not reclaimed until the next global GC, the piling area will grow more quickly. Even so, the speed of growth must be much lower than in a system without reference count GC.

4.2 Invalidation of a Trail Entry

If a trailed pointer is the last reference path to an object in the work area and the object is reclaimed by reference count GC because it consumes the path, the trail entry may become *dangling*. It may cause useless (or, in some cases, erroneous) copying in Piling GC.

If the trail table consists of the indirection pointers, the trail entry can be nullified by clearing the indirection pointer when the pointers are used. However, the indirection path through the trail entry should be distinguished from the normal indirection path.

If the trail table memorizes the address of the trailed pointers, the trail entry itself cannot be found from the trailed pointer. Hence, it is very difficult to nullify the trailing information. There are two alternatives:

(a) When a trailed pointer is created, the reference count is incremented by two instead of incrementing by one to ensure that the count will never reach zero (Figure 7).

(b) When a trailed pointer is consumed, the location where the pointer was stored is cleared. Whether this method is possible at a low cost or not depends on the implementation. The address of the pointer must be memorized during the operation in which a trailed pointer might be consumed.

As (b) needs several dynamic checks during normal execution, it is not generally efficient. On the other hand, (a) has a serious defect in that all the garbage cells, which may be large structures, reachable from the trailed pointer are left unreclaimed and never reused until the next global GC. To conclude, the indirection table is more suitable for trailing than the address table.

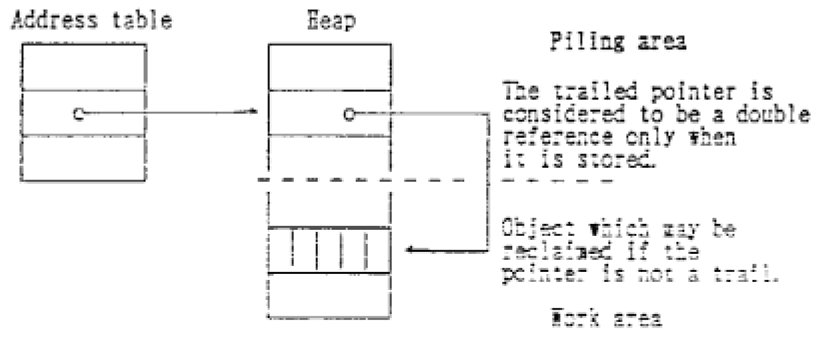
Address table
Heap
Piling area
The trailed pointer is considered to be a double reference only when it is stored.
Object which may be reclaimed if the pointer is not a trail.
Work area

Figure 7: Staving off the Reclamation of an Object Referred to from Trailing

5. Parallel Piling GC on a Multiprocessor

This chapter describes a parallel method for Piling GC on a shared memory multiprocessor. Piling GC can be done in cooperation with several processing elements (PEs).

5.1 Machine Model

Every PE shares a common heap memory, and for their local heap memory, they are supplied one or several memory pages at a time from the common heap memory. If there are not enough pages to be supplied, Piling GC is invoked. Every PE can exclusively access common memory using *lock* mechanism, and can be synchronized by this.

5.2 Procedure

Each PE has its individual destination as *to space* to copy the active objects, and shares the responsibility of marking. The start and end of GC can be synchronized by using a shared counter which is exclusively accessed and indicates the number of signaling PEs.

The algorithm of the Parallel Piling GC is as follows (Figure 8).

(1) The PE which finds that the work area full is called the *GC master*. The GC master synchronizes all the other PEs to start garbage collection simultaneously.

(2) The GC master starts marking and copying from a marking root in the work area. A lock operation is necessary to mark an object. The *to space* for the GC master is reserved one page from the top of the piling area. The top of the piling area is incremented by one page. Although this also needs a lock operation, the copying operation does not need a lock.

(3) PEs other than the GC master signal a request for marking root(s) to be provided and wait.

(4) The busy PE senses the signal of the requests from other PEs (*client PEs*) at every period of a unit operation such as marking with a single root. If the busy PE (*server PE*) finds a client PE, it provides the client PE one or more marking roots from the location pointed to by its own SP. The SP of the server PE is incremented by the number of the giving root(s).

(5) The client PE which is provided with the marking root begins marking and copying with it. If its own *to space* is full or has not yet been assigned a memory page, another page at the top of

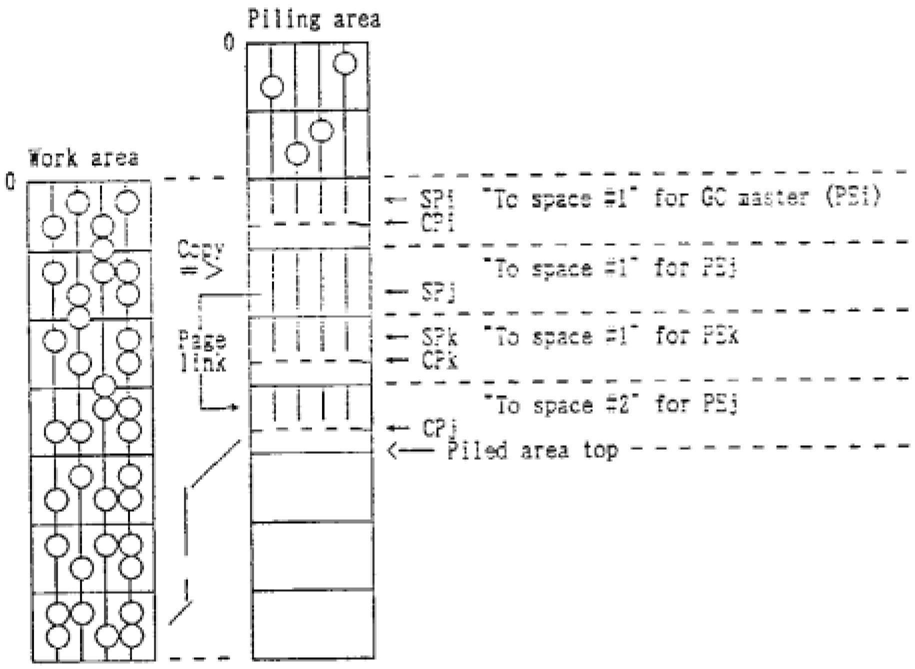

Figure 8: Parallel Piling GC

the piling area is assigned. If the new page is not adjacent to the previous page, a special linkage (*page link*) should be put at the last location of the previous page so that the SP can follow and sweep the new page.

(6) If a PE (including the GC master) finishes marking with a root, it picks up the next root from the location pointed by its own SP without a lock. If it has no root, it signals a request for marking root(s) to be provided and waits.

(7) If all PEs enter the waiting status, garbage collection has finished. The GC master (or some other PE) reallocates the rest of the memory pages to the work area and the piling area and signals to all the other PEs to resume normal execution.

In this method, few lock operations are required and the period of the lock is short (during the write access to mark an object and the *to space* assignment in the piling area). One of its defects is the fragmentation of memory in the piling area. In particular, if a copied object is too large to put the rest of a page in the *to space*, a new page must be assigned and the rest of the old page is left unused unless it is possible to use the fragment by free list management or something else. The worst case might arise in that more memory pages are required than before GC if the fragments which are a little smaller than the copied objects are accumulated even when all the fragments can be used. The simplest solution is that each PE has its own address space as the piling area if fragmentation of address space is allowed. Physical memory pages are assigned contiguously in their spaces. However, global GC must compact each fragment of the space. In this solution, the fragmentation of the memory pages is negligible because it is the same number of pages as that of PEs in the worst case.

5.3 Trail Table

If a trail table is shared among all PEs, a lock operation is necessary to access the table entry. Therefore, it is better to split the table to each PE. The only defect of separation is the fragmentation of memory for the table, and it is also negligible. The allocation of memory pages to the trail table is almost the same as that on a single processor. Each trail table grows by obtaining pages equally from the allocated pages to the work area and the piling area.

6. Conclusion

This paper proposed a garbage collection scheme called Piling GC which is efficient for heap based implementation of AI languages such as Lisp and GHC. Piling GC can attain efficient collection without any loss of memory space, even when the density of active objects is very high, by piling the active objects and isolating them from the garbage collected area. The only defect of Piling GC is that another garbage collector is required as a global GC for compacting all the active objects, even though it may be rarely invoked.

This paper also examined the possibility of combining incremental reference count GC with Piling GC. On the recent machines with hierarchical memories such as cache memory, the incremental reference count scheme is most efficient from the view point of memory access locality. However, it is insufficient in the reclamation of cyclic garbage. Hence, the combination of reference count GC and Piling GC is one good solution for efficient execution, in which an optimized but incomplete incremental GC can be introduced because Piling GC can collect garbage cells which the incomplete incremental GC has left unreclaimed.

The paper also presented a method of parallel Piling GC on a shared memory multiprocessor and showed that the Piling GC scheme is so efficient and general that it can be used in many situations.

Piling GC will be implemented and evaluated on the Flat GHC system on the Parallel Inference Machine [4] which is being developed at ICOT.

Acknowledgement

I would like to thank the ICOT Director, Dr. K. Fuchi, and the chief of the fourth research section, Dr. S. Uchida, who gave us the opportunity to pursue this research. I would also like to thank the research members of the PIM and multi-PSI projects, especially Dr. T. Chikayama, Dr. A. Goto and Mr. N. Ichiyoshi for giving me good advice and for joining in discussions.

References

[1] H.G. Baker, List processing in real time on a serial computer, *Commun. ACM*, 21(4):280–294, 1978.

[2] T. Chikayama and Y. Kimura, Multiple Reference Management in Flat GHC, in *Proceedings of the Fourth International Conference on Logic Programming*, pp. 276–293, 1987. (also published as ICOT Technical Report TR-248).

[3] J. Cohen, Garbage Collection of Linked Data Structures, *ACM Computing Surveys*, 13(3):341–367, Sep. 1981.

[4] A. Goto, Parallel Inference Machine Research in FGCS Project, in *US-Japan AI Symposium 87*, pp. 21–36, Nov. 1987.

[5] H. Lieberman and C. Hewitt, A Real-Time Garbage Collector Based on the Lifetimes of Objects, *Commun. ACM*, 26(6):419–429, 1983.

[6] F.L. Morris, A time and space efficient garbage collection algorithm, *Commun. ACM*, 21(8):662–665, 1978.

[7] K. Ueda, *Guarded Horn Clauses: A Parallel Logic Programming Language with the concept of a Guard*, TR 208, ICOT, 1986. (also to appear in Programming of Future Generation Computers, North-Holland, Amsterdam, 1987.).

[8] D.H.D. Warren, *Implementing Prolog - Compiling Predicate Logic Program*, D.A.I. Research Report 39 and 40, Dept. of AI, Univ. of Edinburgh, 1977.