Distributed Garbage Collection for the Parallel Inference Engine PIE64

Lu Xu, Hanpei Koike, and Hidehiko Tanaka
Tanaka Lab., Dept. of Electrical Engineering, Univ. of Tokyo
Hongo 7-3-1, Bunkyo-ku, Tokyo 113, Japan

Abstract

In this paper, we will present an efficient algorithm for garbage collection of distributed heap memories. We propose that a garbage collector should be comprised by several layers of garbage collections. We refer such a garbage collector to as Garbage Collection System (GCS). Our GCS is mainly based on static analysis and a combination of Reference Counting on memory pages and Mark-Scan Scheme. The Reference Counting Scheme in our GCS concentrates on collecting Single Reference Objects (SRO). The global garbage collection, which is based on Mark-Scan Scheme, is implemented on pipelines of Unifier/Reducers (UNIRED) and is expected to be accomplished in a very short constant time. Particularly, the marking phase adopts a combination of breadth first/depth first strategies to make full use of the pipelines. In order to diminish the impact of remote references on duration of the global garbage collection, we devised a mechanism of co-operation between processors. Another characteristic is that no indirection table is used in usual computation. Indirection tables for remote references are created when the global garbage collection is required and deleted just after the global garbage collection is completed.

Our algorithm, which deals with variable size objects, is very time-efficient, partly real-time and can be implemented with very little space overhead. The sources of its efficiency are discrimination of single-reference objects, memory allocation and management according to object lifetime, and special hardware supports for the global garbage collection.

1 Introduction

So far, many methods about distributed garbage collection have been proposed. They are all based on two schemes: Reference Counting and Mark-Scan. But they lose either the advantages of Reference Counting or the advantages of Mark-Scan because they cannot combine the two schemes efficiently. How to combine the two schemes with a low overhead is a very difficult problem. The problems of these methods are:

1. In these methods, if a scheme is adopted, either one of the two, as many kinds of objects as possible are collected by the scheme, regardless of the differences of their consuming rates.
2. In these methods when Reference Counting is adopted, references to an object are always counted dynamically. It is very difficult to protect objects from becoming garbage prematurely unless Weighted Reference Counting is used. Therefore, the space overhead is increased.
3. There is no consideration about lifetime of an object when allocating it. The objects in one region may be of quite different lifetimes. The lifetime of an object is always determined dynamically at the cost of time overhead. This makes the efficiency of memory-reuse very low.
4. By now, the methods based on Mark-Scan Scheme which are proposed for distributed environments are not impracticable, because their space overheads are very high.

There must be many kinds of objects in a system. These kinds of objects can be divided into several categories according to their consuming rates. According to the characteristics of each one of these categories, we choose a garbage collection method appropriate for it. All these layers of garbage collections make up what we call the Garbage Collection System (GCS). An object can be moved between these categories, i.e., we can decide dynamically which category an object belongs to, somehow similar to [10], [12]. We can also estimate statically which category an object belongs to, just as we propose for PIE64, if it is possible. The main differences between the concept of GCS and the basic idea of [10], [12] are:

1. We recommend that for different layers of a GCS, different schemes should be adopted if it is necessary and feasible. In other words, we advocate a GCS should be based on a combination of Reference Counting Scheme and Mark Scan Scheme.
2. In a GCS, objects are divided by their consuming rates rather than only their lifetimes. There is some close relationship between the concept of *consuming rate* and that of *lifetime*, but they are not equivalents. The difference between them results that we are likely to make a greater effort for a larger object than that for

a smaller object if they are of almost the same lifetime.

As for the GCS proposed for PIE64, for those kinds of objects which are of high consuming rates, we must collect them on time or in time. But for those kinds of objects which consume memory at low rates, we need not collect them immediately after they become inaccessible. It is only to degrade the performance of a garbage collector to make a great effort to collect those kinds of objects of low consuming rates on time. Therefore, we would like to propose to concentrate a real-time garbage collection on those kinds of objects of high consuming rates and commit other kinds of objects to other layers of garbage collection which may be based on Mark-Scan Scheme. We need not be afraid that some kinds of objects of low consuming rates leak out from our real-time garbage collection, even we would let them leak out from our real-time garbage collection deliberately, as long as we can collect them eventually by another layer of our GCS. This is our basic idea.

As said above, there must be many kinds of objects and we can sort them into many categories according to their consuming rates. We can decide which category an object belongs to and some other properties of the object either dynamically or statically. In fact, in many languages, especially in the committed-choice languages like FLENG which is implemented on our PIE64, consuming rates of some kinds of objects, their lifetimes and the numbers of references to them can be estimated in advance.

It is not feasible to analyze an object statically until recently. In recent years, many new logic programming languages are proposed, which have the same characteristic that they do not take allocation and memory-reuse into account. This characteristic exacerbates the anxiousness for efficient garbage collector. On the other hand, this also makes it possible to analyze an object statically. Since all allocations are regulated to the system, the system can know some information about the objects such as the usages and so on. We can estimate the consuming rates, lifetimes and references from these information.

If we can allocate according to the lifetimes of objects as well as according to their consuming rates, we can reuse memory efficiently. Our consideration about lifetimes of objects is quite different from that proposed in [10], [12]. In these papers, lifetimes are always determined dynamically, and tested by garbage collection. When an object survives a few garbage collections, it will be moved to a bank of longer lifetimes. This only means all objects in one region are qualified for the bank they reside, but does not mean that most objects in one region are of almost the same lifetime. In our case, we do mean the latter. This makes it possible to count references to pages instead of to objects and combine Reference Counting with Mark-Scan easily. In our system, page is used for memory management, but it is used mainly as the unit of reference counting. Therefore, the objects in our system are classified into several categories according to their consuming rates. Under one category for which we want to collect the garbage by Reference Counting Scheme, objects are further divided into several kinds according to their lifetimes and we allocate these kinds of objects in different places. Each kind of objects may be stored in several pages.

To realize our garbage collection based on Mark-Scan Scheme efficiently, we improved the methods [6] and [11]. Our garbage collection based on Mark-Scan Scheme is realized with a low space overhead and it increases the parallelism of marking with respect to the methods [6] and [11].

In order not to slow down the processing speed, no indirection tables for remote references are adopted in our system in usual computation. Indirection tables for remote references are created when the global garbage collection is required and deleted immediately after the global garbage collection. We combine the algorithm of marking and that of compaction naturally to implement the creation and deletion of indirection tables efficiently.

We would like to have a distributed GCS with the characteristics list in the following.

1. The GCS should reclaim all the unaccessible objects eventually, including cyclic structures and nonhomogeneous objects, despite there may be some objects spanning several processors.
2. The GCS should reclaim all unusable objects having to synchronize among processors as little as possible.
3. The GCS should complete its work with little spatial overhead.
4. The interconnection networks are usually considered to be the bottleneck of the distributed systems. The GCS should not increase the traffic of the networks.
5. The GCS should meet real-time constraints as far as possible.

Our garbage collection system consists of three stages:

1. Real-time garbage collection based on Paging Reference Counting,
2. Local garbage collection of goalframe areas, which is accomplished independently by each IU (Inference Unit) and much more quickly than the conventional Mark-Scan method,
3. Global garbage collection by the Mark-Scan Scheme.

In fact, there are only two categories of objects in PIE64. The first two stages are provided for the same one category: SRO (Single Reference Objects). The third stage is provided for the other category: UNSRO.

Figure 1: Internal Architecture of IU

In the remaining sections, we will show how we implement reference counting, memory allocation and management according to consuming rate and lifetime efficiently, and combine them with Mark-Scan method to ensure the high performance of both efficiency and real-time nature. At first, we will describe the architecture of PIE64, and then details of the three stages of our GCS. In the final section, we will compare our GCS with other related works.

2 Architecture of PIE64

In recent years, a number of new logic programming languages have been proposed with the purpose of exploiting potential parallelism. To implement these languages efficiently, many machines with parallel architecture are being made. One of them, PIE64, is being designed in our laboratory. PIE64 is a high-speed parallel machine aimed at executing the parallel knowledge information processing programs written in a committed-choice language referred to as FLENG (GHC-like) [14]. PIE64 is made from 64 IUs (Inference Units) which are connected by two high-speed and highly intelligent interconnection networks and one host workstation.

2.1 Internal Architecture of Inference Unit

Each IU consists of 7 parts. They are the Unifier/Reducer(UNIRED), Network Interface Processor(NIP), SPARC processor, Local Memory(LMEM), SPARC memory, Host interface and I/O Interface. These are shown in Fig. 1.

UNIRED is a co-processor of the SPARC processor designed to enhance inference capability. Each UNIRED has a pipeline which is used to expedite the execution of unification and reduction. In order to maintain a high throughput in the pipeline, two techniques are adopted [9]. One is multiple-context processing. It means that UNIRED will process a number of goals concurrently and the context will be switched quickly whenever a remote data access occurs. The other is data balancing mechanism. It is used to balance allocation of data among the memory modules to minimum the contention of data access. UNIRED also provides primitives for the global garbage collection.

NIPs are co-processors of the SPARC processor designed to enhance parallel processing capability. The elementary operations of the NIPs are to fulfill the transfer of data or goalframes between memory modules according to requests.

NIPs also provide basic primitives to support the global garbage collection. These primitives are **Remote-Mark** and **Restore**, which are used to co-operate with UNIRED to complete the global garbage collection.

The SPARC processor is the manager of an IU. It is in charge of scheduling goal execution in UNIRED and managing memory area. System predicates are also executed by the SPARC.

The SPARC, NIPs and UNIRED are connected by three fast pipeline-arbitrated synchronous buses to share local memory, and they can also exchange commands by a high-speed command bus.

2.2 Interconnection Networks

The two interconnection networks [8] are both 64×64 3-stage Omega networks. Because the performance of the two networks will have a great impact on the efficiency of PIE64, we do our best in the designing of the SU (Switching Unit) chip and are confident in assuring its high performance, moderate cost, and ease of implementation.

The two interconnection networks are referred to as PAN and DAAN and have automatic load balancing facilities. PAN uses this facility to balance processing load among IUs. DAAN uses this facility to balance the allocation of data among IUs and to reduce access contention.

3 FLENG

FLENG is a logic programming language. It is a general purpose language for all kinds of architectures. FLENG uses committed choice non-determinism and descends mainly from GHC (Guarded Horn Clauses) and Parlog.

In FLENG, a goal can be executed as usual no matter that other goals succeed or fail. This means that the result of the execution of a goal, either success or failure, has nothing to do with other goals. The logic relationship between goals is only presented by share variables. It is possible to eliminate the guard part in each clause.

FLENG has very few primitives, but these primitives are flexible to be able to be combined to form any program. They are:

1. **unify(R, X, Y)** binds **R** with true if the unification of **X** and **Y** succeeds, otherwise binds **R** with false.
2. **compute(Op, X, Y, R)** writes the result of the operation indicated by **Op** on **X** and **Y** into **R**. The possible operations described by **Op** are:
   - arithmetic operations: +, −, ∗, / .
   - bitwise logical operations: and, or, xor.
   - numerical comparison operations: <, >,= , sametype.
3. **write(T, R)** binds **R** with true if it succeeds to output **T** to a standard file (usually screen), otherwise binds **R** with false.
4. **call(G)** executes goal **G**.

Like many other parallel logic languages, FLENG tends to consume memory at much higher rate than conventional languages. It is also said that most objects are referenced only once. For example, goalframes are always referenced only once. This makes it possible to manage memory and collect garbage much more efficiently.

4 Distributed Environment of PIE64

In each IU, there is a local memory of 1M words (4M bytes). The distributed memory heaps make up the global memory of PIE64, and an object can reference any other object directly in the system.

In many parallel machines like PIE64, indirection tables for remote references are adopted. This make it easy to manipulate remote pointers when local references outnumber remote ones by far. But in PIE64 goals are often copied between IUs; there are quite a lot of remote pointers. If we adopt indirection tables in usual computation, the processing speed will be slowed down rapidly. Therefore in PIE64, no indirection table will be introduced in usual computation. But when a remote pointer is created, additional space must be kept for the use of compaction in the future.

5 Object-Management System (OMS)

Committed-choice languages like FLENG reference most objects only once. For PIE64, we categorize objects into two categories: SRO (Single Reference Objects) and UNSRO. By some experiments, we know that for FLENG, memory is mainly consumed by objects of SRO. The consuming rate of SRO objects are much higher than that of UNSRO objects. We would like to collect the garbage of SRO category by Reference Counting Scheme. Therefore, we further divide objects of SRO category into several kinds according to their lifetimes and manage them respectively.

In fact, in PIE64, we can guess statically which category an object belongs to, and if it pertains to SRO, we can also guess statically which kind it is in SRO category.

To manage memory efficiently, local memories are divided into pages. We keep a reference count for every page of SRO (Single Reference Objects) areas. No reference count is kept for objects of UNSRO and the garbage of UNSRO is collected only by the third stage of our GCS. Therefore, there are two kinds of allocations in our system. One is page-allocation. The other one is object-allocation.

The allocation scheme is shown in Fig. 2.

Figure 2: Allocation Scheme of OMS

Figure 3: Real-Time Garbage Collection Scheme

6 Real-Time Garbage Collection

By our experience, in the logic programming languages like FLENG, memory is mainly consumed by the objects of SRO. And the average size of the objects of SRO is much larger than that of the objects of UNSRO. Therefore, we are likely to make a greater effort to collect the garbage in SRO areas. For PIE64, we collect them by Paging Reference Counting Scheme.

The base of our real-time collector is the consideration about lifetime. In the papers [10], [12], they concentrate on how long an object lives precisely. With the assumption that an object graduating from many garbage collections would live for a long time, the efforts are concentrated on the newer objects. In our case, we would like to allocate all objects being of the same lifetime to the same area, rather than pay attention to how long an object lives exactly.

According to our OMS, we allocate and manage SRO objects according to their lifetimes. Therefore, we expect most objects in one page of SRO to be of almost the same lifetime. In the ideal case, all objects in one page of SRO become inaccessible at the same time. This makes it possible to treat all objects in one page of SRO as one object, and only have to keep reference count for pages of SRO. We can reclaim a page when its reference count becomes zero.

The real-time garbage collection scheme is shown in Fig. 3.

7 Local Garbage Collection

There is a problem with the real-time garbage collection. If we make a mistake when estimating the lifetime of an object, the mistake may largely affect reuse of the page in which the object resides. Therefore, there may be many pages in which only a small part of them is still in use, but they cannot be reclaimed completely by the real-time garbage collection. Even if this kind of mistake is scarce, the efficiency of memory reuse may be degraded severly. To solve the problem, we introduce the second stage of garbage collection.

In PIE64, there are various kinds of objects of SRO. Among these kinds of objects, the problem presented above is most serious for goalframes [5] since they are usually suspended and activated. Goalframes [5] are representations of goals in real memory and can be seemed as process records in conventional machines. Consequently, we do the local garbage collection only for goalframes. Thus we only need mark the first cell of each goalframe and

compact by one scan, since in PIE64 the first cell of each goalframe is an integer which represents the length of the goalframe itself. Accordingly, we can simplify the local garbage collection greatly.

In each local memory of PIE64, there is two local root queues. One is active goal queue, the other is suspended goal queue.

When the local garbage collector is started, it will mark all the goalframes accessible from the active goal queue and the suspended goal queue. This mark phase is different from the conventional one, because there is no need of marking all the cells in the goalframes. It is enough only to mark the first cell of the goalframes. At the same time, the technique of pointer reversal is used. The pointer reversed will not be recovered until compaction phase.

Secondly, we will compact garbage. Logically all the pages of goal frames can be seemed as consecutive, therefore we can accomplish compaction by sliding each goalframe sequentially, since we can know the length of the goalframe by tracing the pointer reversed, and revising the pointer reversed.

Because for each goalframe only the first cell is needed to be marked and the compaction can be completed with only one scan, the local garbage collection can be fulfilled much more quickly than the conventional Mark-Scan method.

8 Global Garbage Collection

When free memory of an IU is run out and both the real-time garbage collection and the local garbage collection cannot reclaim any free space, PIE64 will have to come to a halt and the global garbage collection is triggered.

In our proposal of global garbage collection for PIE64, we divide the GC into two phases, one is marking phase, the other is compaction phase. In the marking phase, the system-wide accessible cells will be marked using a combination of parallel breadth-first/depth-first strategies. In the compaction phase, the marked cells will be compacted to one end of local memories as much as possible. In order to diminish the overhead of operation of remote reference in usual computation, indirection tables are not introduced in usual computation. Indirection tables are created when the global garbage collection happens and deleted after the global garbage collection is accomplished. In fact, this stage is mainly provided for UNSRO objects.

8.1 Marking Phase

Now, we present the idea of the marking scheme. Its space requirements are fixed and can be guaranteed in advance. The original idea is from [6] and [11].

The key idea is:

In the conventional collector used before, a stack is used to execute marking. But if a tree structure is used instead of stack, we can get the idea suitable for parallel marking. This tree is referred to as the Marking Tree. So we can imagine a marking task for the root is spawned when GC is called. Therefore, a number of sub-mark-tasks are spawned in accordance with its children with the limitation of definite space. In addition, the marking tree is simultaneously built for termination. Termination is detected since each marking task eventually spawns an uptree task, which propagates upward in the marking tree. When an uptree task arrives at a cell, it will test whether marking tasks are spawned on all the children of the cell. If there are the children on which marking tasks have not been spawned, a number of marking tasks will be spawned on these children. Otherwise, if the cell is root, no task is spawned. If the cell is not root, a new uptree task will be spawned from this cell and pass over upward the Marking Tree. When all tasks about marking have been executed and no new task is spawned, it indicates that marking is complete.

If we pay close attention to the algorithm, we can find that it is quite different from that described in [6], [11]. In [6] and [11], there must be a counter kept for each object. The first defect of them is that the overhead due to counters is very high. And if we consider the effect that we keep a counter for each object, we can find it is only to guarantee that we would not spawn an uptree task from the object until all uptree tasks to the object have been received. This will, in turn, decrease the parallelism of marking rapidly. To exclude the overhead of counters, we propose not to keep a counter for each object and spawn an uptree task from an object just after that the uptree task have been received from the child on which the marking task is last spawned instead of after all uptree tasks have been received by the object. This modification subtlely increases the parallelism of marking as well as eliminates the spatial overhead. On the other hand, we must also modify the condition of completion to that no task about garbage collection is left.

Therefore, marking is performed by two kinds of messages:

- **Mark(o,p)**: which means that o is to be marked as one child of p.
- **Backward(o,p)**: which means that o has been marked as one child of p.

Figure 4: Co-Operations of the Processors

We will use these two basic primitives for the global garbage collection.

In fact, there is no need of keeping pointers from parents to their children, but pointers from children to their parents are required. In PIE64, because we have no other space to store them, we have to use the pointer reversal technique.

As for PIE64, we made two important improvements. The first one is to implement this algorithm on the pipeline of each UNIRED.

It is obvious that this algorithm is a parallel algorithm. The two primitives are basic primitives provided by each UNIRED. They are able to be implemented simply on the pipeline of each UNIRED. Therefore, we can make full use of the pipeline of each UNIRED and expect that the performance of pipelines is so well that the duration of marking phase can be reduced severely.

The other improvement is to design a co-operation between processors to diminish the lengthy delay owing to remote references. In a distributed environment like PIE64, an object can reference any other object on another processor directly. Owing to remote references, the duration taken by global garbage collection may be many times that taken by local garbage collection with the same size of local memory. This implies that remote references are likely to enlong the entire cycle of global garbage collection.

To reduce the effect of remote references on the global garbage collection, we designed a co-operation among processors and some hardware supports. These are described in the following.

In the marking phase, the active goal queue, the suspended goal queue, and **Remote-Mark** requests are treated as local roots. When mark phase is started, SPARC processor in each IU marks all cells of the local roots and writes them into the queue prepared for the pipeline in UNIRED. UNIRED reads the roots from the queue and starts marking according to the scheme described above. When a remote pointer is found, UNIRED will check whether there are spare buffers in NIPs. If there are spare buffers, a **Remote-Mark** request will be sent to the NIPs. Otherwise, UNIRED will stop its work and wait until there are spare buffers to store the message. When a NIP receives a **Remote-Mark** request from UNIRED, it will store it in the destination IU with the help of the corresponding NIP, and rewrite the original cell with the pointer to the new address.

The co-operations of these processors are shown in Fig. 4.

8.2 Compaction Phase One

This phase operates locally, so there is no need of using NIPs and the interconnection networks. At first, UNIRED starts compaction phase one from the high end of memory, and operates according to the scheme of Morris phase one [13]. When a remote pointer is found, UNIRED will do nothing.

8.3 Compaction Phase Two

UNIRED will start this phase from the low end of the memory and do the same operations as that described in Morris phase two [13] until the boundary between local cells and **Remote-Mark** requests is reached. But when a remote pointer is found, a **Restore** request will be sent to the NIPs. The corresponding NIP will read the message from the destination, and restore the original cell.

The marking and restoring of a remote reference in the global garbage collection are shown in Fig. 5.

Figure 5: Marking/Restoring of A Remote Reference

8.4 Supports from NIPs

The NIPs provide supports for the global garbage collection. They provide two primitives to help UNIRED fulfill the mark phase and the compaction phases. The one of the primitives is **Remote-Mark**. When a NIP receives a command like this, the elementary operations are :

- to send a connecting request with the IU in which the destination is.
- to have the request stored in the proper place P (by the help of its partner NIP).
- to rewrite the original cell with the address P.

The other one of the primitives is **Restore**. When it is received by a NIP, the basic operations are:

- to send a connecting request with the IU in which the destination is.
- to read the content of the destination (by the help of its partner NIP).
- to restore original cell with content read.

With the help of these two primitives, UNIREDs can concentrate on marking and compacting local objects. This will expedite the mark and compaction phase largely.

9 Performance Evaluation

We have implemented our GCS on a simulator of PIE64 and executed some programs on this simulator. We would evaluate our GCS on the basis of this simulator. In our simulator, the page size is assumed to be 1K words.

9.1 Space Overhead

At first, we would like to analyze the spatial overhead of our GCS. In our GCS, there are two kinds of spatial overhead. One is that used for the real-time garbage collection. The other part is that used for the global garbage collection. Accordingly, we only need to estimate the space overhead for the real-time garbage collection and the global garbage collection.

We can know that the space overhead for the real-time garbage collection is proportional to the number of pages. We can simply consider it to be $o(N)$. Here, N is the size of memory.

As for the second part, we know it is $2N$ bits, because two bits are dedicated to the global garbage collection in each cell.

Therefore, the total amount of space overhead is $2N + o(N)$ bits.

As far as we know, the Mark-Scan Scheme is usually advocated for its low spatial cost. Now, our three stage garbage collection is achieved at almost the same cost as Mark-Scan method, we think our collector may be the cheapest method in space requirement suitable for PIE64.

9.2 Real-Time Nature

Because we only offer the real-time garbage collection for SRO, therefore, the real-time nature of our GCS relies on

- the percentage of the pages consumed by SRO over that consumed totally,

Program	Total Pages Consumed	SRO			UNSRO		
		SRO Pages Percentage	Allocation Times	Average Size	UNSRO Pages Percentage	Allocation Times	Average Size
6-Queen	237.14	96.67	56859	4.03	3.33	6433	1.23
10-Append	1.91	95.20	566	3.21	4.80	81	1.13
13-Nreverse	1.81	89.87	522	3.12	10.13	133	1.38
10-Merge	2.13	93.92	627	3.20	6.08	107	1.21

Table 1: Consuming Rates

Programs	IU Number and LMS[a]	Percent (RGC[b])	Percent (LGC[c])	Percent (GGC[d])
6-Queen	4 IUs, LMS = 23	54.65	25.29	20.07
7-Queen	4 IUs, LMS = 50	55.78	31.66	12.56
13-Append	4 IUs, LMS = 13	100.0	0.00	0.00
10-Merge	4 IUs, LMS = 13	100.0	0.00	0.00

[a]means Local Memory Size and the unit is page.
[b]Real-Time Garbage Collector
[c]Local Garbage Collector
[d]Global Garbage Collector

Table 2: Performance of the GCS

- the percentage of the pages collected by real-time garbage collection over that collected our GCS.

We will discuss the two below.

Intuitively, the first percentage represents the nature of FLENG language. FLENG is often said that most of objects are referenced only once. From this view, our collector is close in spirit to the collectors proposed recently, most of which try to make use of nature of languages they serve as much as possible.

The second percentage can reflect the real-time nature of our GCS precisely. In our real-time collector, the local garbage collection is not included. In fact, our local garbage collector always works at a very high speed so that we think it can satisfy the real-time constraints perfectly. We also give the total effect of real-time collector and local collector in the tables.

We executed some programs on our simulator, and the results are shown in Table 1 and Table 2.

From Table 1, we can know immediately that the storage consumed by SRO is always above ninety percent. This nature is the base of our collector.

From Table 2, we can know that the garbage collected by the real-time garbage collector is over fifty percent. If we include the local garbage collector, we can see the garbage collected "in time" is about eighty percent. From the results, we can imagine that the real-time nature of our collector is quite good. Of course, we must admit that our collector is not a truly real-time collector and cannot guarantee it meets the constraint at any time. Sometimes we must make a choice either to meet real-time constraints but slow down computations, or to expedite the computations but neglect the real-time constraints. For PIE64, we choose the latter alternative.

Here, we would like to consider the average delay period from the time an object is released to that the object is collected. In our GCS, it is evident that the average delay period for SRO objects is dependent on their consuming rates, because this parameter is just proportional to the average time by which the programs of FLENG consume one SRO page memory. We can also express the parameter in the following.

$$ADP_{SRO} \propto PageSize/ACR_{SRO} \quad (1)$$

where, ADP_{SRO} means that the average delay period of SRO garbage, ACR_{SRO} means that the average consuming rate of SRO objects.

This formula is very important because it implies that our GCS is able modify the ADP_{SRO} automatically according to the average consuming rate of SRO. When the average consuming rate of SRO is very high, ADP_{SRO} becomes small, i.e., the garbage of SRO is collected quickly. When the average consuming rate of SRO is low, therefore, there is no need of collecting garbage immediately. At this time, ADP_{SRO} gets large, this means the garbage of SRO is collected slowly. Consequently, our GCS can work efficiently and keep constant real-time nature by modifying the collecting rate automatically. From this view, our collector is likely to collect garbages "in time" while other Reference Counting collectors are likely to collect garbage "on time".

IU Number and LMS	Percentage of Remote References	The Time Taken by a global GC (clock)	The Ratio to 1 IU
1 IU, LMS = 20	0.00	122743.13	1.00
4 IUs, LMS = 20	21.86	118211.38	0.96
16 IUs, LMS = 20	27.54	119997.48	0.98
64 IUs, LMS = 20	30.74	124390.21	1.01

Table 3: Performance of the Cooperations between Processors

9.3 Performance of Co-operations between Processors

In general, the length of time interval taken by one time of global garbage collection may be much longer that taken by one time of local garbage collection with the same size of local memory. The remote references play a major role in this situation. Especially for PIE64, there are so a lot of remote references that the lengthy delay of the global garbage collection owing to remote references may be unacceptable if we do not design special co-operations between processors.

To solve the problem, we design the cooperation between processers and expect this will alleviate this problem to some extent. The results are list in Table 3.

When the number of IUs increases to sixty-four, the time taken by the global garbage collection only increases about one percent compared to the case of uniprocessor.

From the results, we can derive that through the co-operations between processors, our global garbage collector can work so well that the duration of one time of the global garbage collection is almost kept to a constant when the number of IUs increases.

There are two other potential problems. We implement our collector with two assumptions:

1. In languages like FLENG, memory is mainly consumed by SRO.
2. We can approximately classify the objects of SRO according to their lifetimes.

Fortunately, these two assumptions seem to be true. This can be demonstrated by Table 1 and Table 2. Especially, from Table 2, we know the most of garbage are collected by the real-time garbage collector. Consequently, it is turned out indirectly that we can approximately estimate the lifetimes of objects.

The another advantage of our collector is that it would not increase the traffic of interconnection networks.

From the evaluations above, we can claim our GCS is able to satisfy the aims proposed in the first section. We think it is the cheapest and is sufficient for PIE64.

10 Comparisons with other Related Works

So far, there are many methods which have been proposed for distributed garbage collection. We would do a comparison with each of them.

10.1 Weighted Reference Counting

This algorithm [2], [16] is proposed with the base of Reference Counting. It decreases the traffic of the interconnection networks and obviates objects from premature compared with pure Reference Counting. But in turn, it increases the spatial and time overhead. Sometimes indirection cells are needed, this may cut down the system performance.

The shortcomings are:

1. its space overhead are very high,
2. it cannot reclaim cyclic structures,
3. it increases the traffic of networks absolutely. The interconnection networks are usually the bottleneck of the systems like PIE64. This may degrade the system performance rapidly,
4. it is likely to divide memory into small fragments. It is difficult to combine objects reclaimed into a lagerer area although they may be consecutive.

Its advantages are real-time, simplicity and high efficiency of memory reuse.

But its real-time nature is gained at the cost of time and space. Our purpose is to get high both efficiency and real-time property, so we must make a compromise. For the environment of PIE64, local memories are mainly consumed by SRO objects, especially by goalframes, so if we reclaim the areas of SRO in time, we can obtain real-time nature to a great extent.

Its high efficiency of memory reuse is gained from reclaiming all unaccessible objects. As for the SRO area.

our method may keep reuse efficiency almost at the same level with it. In fact, there is no need of reclaiming all objects immediately when they become unaccessible and this may only degrade the system performance. This is true not only for UNSRO objects but also for SRO objects when their consuming rate is not very high in some applications. To overcome this disadvantage, Lazy Reference Counting may be introduced [4]. At this point, our Paging Reference Counting is similar to Lazy Reference Counting. Just as described in the previous section, our real-time garbage collector is able to modify the average delay period of SRO garbage according to the average consuming rate of SRO. In other words, it can modify the collecting speed according to the average consuming rate of SRO objects. Therefore, our collector realize this capability automatically and more efficiently than Lazy Reference Counting Scheme, because no additional overhead is incurred.

10.2 Copying Algorithm

This kind of method [15] is based on the algorithm proposed in [1]. It has many advantages, such as:

1. it meets the real-time constraints,
2. it can reclaim all unaccessible objects eventually,
3. it combines mark and scan naturally, therefore, it would not divide memory into many fragments.

But its time-efficiency is not very good. It also need co-operate with the inference processor. Indirection tables are used for remote references in processing. Another problem is that the amount of floating garbage[3] may be increased rapidly when it is implemented for the logic languages like FLENG. All these maybe, in turn, degrade the system performance largely.

The main problem, we think, is its high spatial overhead. When one half of the memory is used for inference and the other half is used for garbage collection, we could not help thinking it is too expensive. Especially in the case of FLENG, the consuming rate of memory is very high, if garbage is collected only by copying method, the efficiency may be too lower that we cannot accept [12].

10.3 Mark-Compaction Algorithm

This method [11] is of very good efficiency both in time and in space. But it loses real-time property completely. One difference between our global garbage collector and the method is that we use indirection tables only when the global garbage collection is required while the method [11] relies on indirection tables all the time, especially when local garbage collection in introduced. This difference is important for PIE64 since the operation of copying pointers between IUs are manipulated frequently. Therefore, it is convenient for PIE64 to copy pointers between IUs, since we need not check whether they are local references or remote ones and overhead of transformations between local references and remote ones can be eliminated. The other difference is that there is no counter in each object of our GCS, therefore, the space overhead of our global garbage collection is much lower than that of [11]. And we also realize our global garbage collection with subtlely higher parallelism.

Acknowledgements

The authors would like to thank many people who helped us. We got many incisive comments from the members of the Special Interest Group of the Inference Engine (SIGIE) of our laboratory, especially Mr. Kentaro Shimada, from whom we got the interpreter of FLENG and many valuable advices about our algorithm, and Mr. Takeshi Shimizu, who gave us many benifical discussions about the NIPs' supports for GC. This work is supported by Grant-in-Aid for Specially Promoted Research of the Ministry of Education, Science and Culture.

References

[1] Baker, H.G.Jr. List Processing in Real Time on a Serial Computer. *Comm. ACM 21, 4*, 280–294, April 1978.

[2] Bevan, D.I. Distributed Garbage Collection using Reference Counting. *PARLE*, 1987.

[3] Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S. and Steffens, E.F.M. On-the-Fly Garbage Collection : An Exercise in Cooperation. *Comm. ACM Vol. 21, No. 11,* 966–975, 1978.

[4] Glaser, H. On Minimal Overhead Reference Count Garbage Collection in Distributed Systems. *Imperial College of Science and Technology*, 1987.

[5] Goto, A. A Highly Parallel Inference Engine Based on Goal-Rewriting Model:PIE. *PhD thesis, Dept. of Information Engineering, Univ. of Tokyo*, Feb. 1983.

[6] Hudak, P., and Keller, R.M. Garbage Collection and Task Deletion in Distributed Applicative Processing Systems. *ACM Symp. on Lisp and Functional Programming*, 168–178, 1982.

[7] Hughes, J. A Distributed Garbage Collection Algorithm. *Functional Programming Languages and Computer Architecture, J.-P. Jouannaud (Ed.). LNCS 201, Springer-Verlag,* 256–272, Sept. 1985.

[8] Koike, H. and Tanaka, H. The High Performance Interconnection Network of the Parallel Inference Machine PIE64. *Computer Architecture Symposium, Japan,* May, 1988.

[9] Koike, H. and Tanaka, H. Multi-Context Processing and Data Balancing Mechanism of The Parallel Inference Machine: PIE64. *Proc. of FGCS'88, ICOT, Tokyo, Japan*, 1988.

[10] Lieberman, H., and Hewitt, C. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Comm. ACM 26, 6,* 419–429, June 1983.

[11] Mohamed-Ali, K.A., and Haridi, S. Global Garbage Collection for Distributed Heap Storage Systems. *Internal IBM Rep. RC11082 (49769) and TRITA-CS-8502, Dept. of Computer Systems, Royal Institute of Technology, Stockholm*, April 1985.

[12] Moon, D.A. Garbage Collection in a Large LISP System. *ACM Symposium on LISP and Functional Programming*, 235–246, 1984.

[13] Morris, F.L. A Time- and Space- Efficient Garbage Compaction Algorithm. *Comm. ACM, Vol. 21, No. 8,* 662–665, 1978.

[14] Nilsson, M. and Tanaka H. -FLENG Prolog- The Languages which turns Supercomputers into Parallel Prolog Machines. *Proc. of LPC'86*, 1986.

[15] Rudalics, M. Distributed Copying Garbage Collection. *Proc. ACM Conf. On LISP and Functional Programming*, 364–372, 1986.

[16] Watson, P. and Watson, I. An Efficient Garbage Collection Scheme for Parallel Computer Architecture. *PARLE*, 1987.