5U-7

# Distributed Garbage Collection for the Parallel Inference Machine: PIE64

Lu Xu, Kentaro Shimada, Takeshi Shimizu, Hanpei Koike, and Hidehiko Tanaka
Tanaka Lab., Dept. of Electrical Engineering, Univ. of Tokyo*

**Abstract**

In this paper, we will present an elegant algorithm for garbage collection of distributed heap memories. Our method mainly combines reference counting on memory page with a global Mark-Scan scheme. This algorithm 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 support for global Mark-Scan GC.

## 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 supplied by Reference Counting or the advantages supplied by Mark-Scan because they cannot combine the two schemes efficiently.

In the following, we will show how we implement reference count, memory allocation and management according to lifetime efficiently with low overhead in our system, and combine it with mark-scan method to ensure the high performance of both efficiency and real-time property.

Our garbage collector has three stages:

1. The real-time garbage collection based on Paging Reference Counting,
2. The local garbage collection of goal frame area,
3. The global garbage collection by mark-scan method.

The third stage has been described in [5], and here we will only describe the other two stages.

## 2 The Features of FLENG

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

*Hongo 7-3-1, Bunkyo-ku, Tokyo 113, Japan

```
claim form: (type, size)

if   (the page of the type is still enough
          for this allocation)
  then allocate for the object in the page
  else  1. to allocate a new page for the type
        2. to allocate for the object in the page

if (the type is one kind of SRO)
  then  to increase the reference count
          of the corresponding page
```

Figure 1: The allocation scheme of OSS

## 3 Object-Storage System (OSS)

In order to collect the garbages efficiently, we do allocations according to the lifetimes. We can divide each local memory into pages. Therefore, there are two kinds of allocations in our system. One is page-allocation. The other one is object-allocation. We keep a reference count for every page of SRO (Single Reference Objects) areas.

The committed-choice languages like FLENG are often said that most objects are referenced only once. We will assort such objects into several categories according to their lifetimes and manage them respectively.

The allocation scheme is shown in Fig.1.

## 4 The Real-time GC Based on Paging Reference Counting

The base of our real-time collector is the consideration about lifetime. In the papers [2] [3], they concentrate on how long an object has lived. With the assumption that an object graduated 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 OSS, we allocate and manage objects according to their lifetimes. Therefore, we can expect most objects in one page are of almost the same lifetime. In the perfect case, all objects in one page can be expected to be dereferenced at the same time. This makes it possible that we treat all objects in one page

```
dereference form : (type, size)

if (the type is one kind of SRO)
  then  1. to decrease the reference count
           of the corresponding page
        2. if (the reference count became zero)
           then   to reclaim the page
```

Figure 2: The real-time garbage collection scheme

as one object. We only have to keep reference count for pages. We can reclaim a page when its reference count became to zero.

In order to avoid the problem of premature, we would propose to keep reference count for the pages of the objects that the references to them can be determined statically. For PIE64, we will only keep reference counts for the pages of SRO.

As described above, we assort the objects referenced only once into several categories and keep the reference count when allocations or deallocations happen on each page of SRO. When the count of a page is decreased to zero, we can reclaim and re-use the page. The dereference scheme is shown in Fig.2.

## 5 The Local Garbage Collection of SRO Areas

There is a problem in the real-time garbage collection. There may be many pages in which only a small part of them are still in use, but they cannot be reclaimed completely. To solve the problem, we introduce the second stage of garbage collection.

When free pages of a local memory have been exhausted, we start the second stage garbage collection. It will mark all the accessible objects in SRO areas from the local roots and compact them into as few pages as possible.

For PIE64, we will do this garbage collection only for goal frames. In each local memory of PIE64, there is two local root queues. One is active goal queue, the other is suspended goal queues.

When this garbage collector is started, it will mark all the goal frames 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 goal frames. It is enough only to mark the first cell of the goal frames.

Secondly, we will do the compaction. All the pages of goal frames can be seen as consecutive logically, therefore we can do compaction only to slide all the goal frames to as few pages as possible.

Because in each goal frame only the first cell is needed to be marked and the compaction can be fulfilled with only one scan, this garbage collection can be fulfilled much more quickly than the conventional method.

## 6 The Co-operations for the Global GC

When global garbage collection is required, the co-operations between processors are needed. In the mark phase, the active goal queue, the suspended goal queue, and the remote mark requirements are treated as local roots. When mark phase is started,

- the SPARC processor in each IU will mark all cells of the local roots and writes them into the queue prepared for the pipeline in UNIRED.
- UNIRED reads the roots from queue and starts marking according to the scheme described above. When a remote pointer is found, a remote-mark-requirement primitive will be sent to the NIPs.
- when a NIP receives a remote mark requirement from UNIRED, it will store it in the destination IU with the help of the corresponding NIP, and rewrite the original cell.

The compaction phase can be fulfilled in the same way as described in [1].

## 7 Conclusion

As described above, we know that the garbage collector can implemented with a little overhead. We will evalute the method in the near future.

## References


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

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

[3] Moon, D.A. *Garbage Collection in a Large LISP System*, In: ACM Sysposium on LISP and Functional Programming. (1984) 235–246.

[4] Shimizu, T., Tanaka, H. *The Network Interface Processors for Parallel Inference Machine: PIE64* Parallel Processing Symposium Feb. 1989

[5] Xu, L., Shimada, K., Koike, H. and Tanaka, H. *A Study of Garbage Collection for PIE64*, Proc. 34th Annual Convention IPS Japan 1987.