This is the accepted manuscript of a paper published in
Proc. 4th Int'l Symp. on Computing and Networking (CANDAR'16)
Copyright (C) 2016 IEEE

Hardware Supported Marking for Common Garbage Collections

Shinji KAWAMURA* and Tomoaki TSUMURA*
*Nagoya Institute of Technology
Gokiso, Showa, Nagoya, Japan
Email: camp@matlab.nitech.ac.jp

***Abstract*—Many mobile systems have to achieve both high performance and low memory usage, and the total performance of such the systems can be largely affected by the effectiveness of GC. Hence, the recent popularization of mobile devices makes the GC performance play one of the important roles on the wide range of platforms. The response performance degradation which is caused by suspending all processes for GC has been a well known potential problem. Therefore, GC algorithms have been actively studied and improved, but they still have not reached any fundamental solution. In this paper, we focus on the point that the same objects are redundantly marked during the GC procedure that is implemented on DalvikVM, one of the famous runtime environments for the mobile devices. Then we propose a hardware support technique for improving marking routine of GC. We installed a set of tables to a processor for managing marked objects, and by referring these tables, redundant marking for marked objects can be omitted. The result of the simulation experiment shows that the execution cycles of GC are reduced by 20.7% at a maximum, and 10.2% on average.**

I. Introduction

Many mobile systems have to achieve both high performance and low memory usage, and the total performance of the wide range of platforms now can be affected by the effectiveness of *Garbage Collection (GC)*. It has been well known that GC has a big effect on the total performance of the systems such as server-side Java runtime environment. In addition, the response performance degradation which is caused by suspending all processes for GC has been a well known potential problem. To address this problem, algorithmic solutions have been mainly explored in many studies for GC. However, most of the studies are for reducing the frequency of collections by using a complicated tuning for the systems and the applications, or for reducing the response time of the system at the cost of the throughput. Thus, the problem of GC has not reached any basic resolutions.

For this reason, we have focused on the basic routines that many GC algorithms commonly have, and we aim to improve the performance of GC by supporting the routines with hardware. First of all, we investigated the operation of GC which is implemented on DalvikVM; it is famous as a runtime environment on the mobile devices. As a result, we found that it takes much time to mark objects one after another with tracing references between the objects. In addition, we also found that many objects are marked redundantly. In this paper, we propose a hardware support technique to omit such redundant marking for marked objects. In our proposal, for managing marked objects, we install dedicated tables to the processor. By referring these tables, redundant marking for marked objects can be omitted. Thereby, the tracing routine which consumes much time in GC can be improved, and high performance GC can be achieved.

In this paper, we aim to make the following contributions:

1) We found that GC includes redundant marking, that is, some objects are marked several times per collection.
2) We propose a hardware supported marking for common GCs. In our proposal, we install dedicated tables to a processor. By using these tables, redundant marking can be omitted.
3) We evaluate the proposed method. The results show that the execution cycles can be reduced by 20.7% at a maximum, and 10.2% on average.

II. Background

In this section, we explain GC, the representative GC algorithms, and related work.

A. Garbage Collection

GC is a routine for automatic memory management. It automatically frees a part of heap area occupied by unused objects. Figure 1 shows a state of the heap area and references between objects. A pointer to an object located in the heap area is stored into a global variable or an area such as call stack or registers, which are directly accessible from applications. These areas are called the *set of roots*, and by tracing pointers from here, all the objects in the heap area can be referred. The objects which are located in the heap area may have pointers to some other objects. Those objects which are referred by another object can be reached from the set of roots, and are called *live objects*. On the other hand, the unused objects which can not be reached from the set of roots are called *dead objects*. Because GC frees a part of the heap area occupied by dead objects, the memory area where the objects have been located can be reused.

Mark & Sweep [1] is one of the representative GC algorithms. This algorithm consists of two phases. The first phase is called Mark phase when all live objects are marked, and

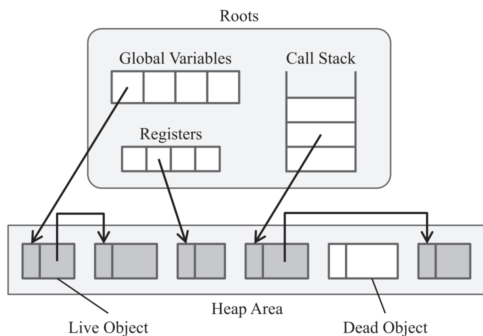

Fig. 1. Heap Area and References during program execution.

the second phase is called Sweep phase when dead objects are collected. These two phases are executed repeatedly.

Copying [2] and Reference Counting [3] are also the representatives of GC algorithms. In addition, although many other GC algorithms have been studied, all of them are just combinations or improvements of those three algorithms [4]. Especially, Mark & Sweep is widely used as a base of many algorithms because it is comparatively easy to be implemented.

B. Related Work

Concurrent GC [5] is a representative algorithm of GC. This algorithm aims to shorten suspension period of the system by running in parallel with other applications. However, while GC is being executed in parallel with other applications, some objects can not be marked properly because some pointer modifications may be caused by the applications. Thereby, Concurrent GC makes it possible to prevent this issue and run in parallel with other applications, by using the barrier synchronization and starting mark routine again from those pointers. The barrier synchronization is used for detecting the modifications of pointers. Whereas, this algorithm decreases the throughput because of the overhead which is caused by synchronization between GC and other applications.

Although most of the improvements of GC focus on the software as mentioned above, a few studies have proposed hardware support techniques, for example, SILENT [6] and Network Attached Processing (NAP) [7]. These techniques use the algorithms which let GC run in parallel with other applications as well as Concurrent GC, and achieve high performance with hardware supports for barrier synchronization.

III. Bottleneck Analysis of GC

In this paper, we cancel the performance bottleneck of the representative GC algorithms with a hardware support, and then we aim to achieve significantly high performance GC.

A. The Major Routine in GC

We analyzed a representative GC algorithm for investigating the bottleneck of GC. In this analysis, the target is Mark & Sweep implemented in DalvikVM. First of all, we estimated the breakdown of the execution time which GC spent. We used gem5 simulator system [8], and executed DalvikVM on this simulator to measure the execution cycles of each routine of GC. We measured the execution cycles of GCBench [9], AOBench [10], and four benchmarks in SPECjvm2008 suite [11]. As a result, we found that Mark phase occupies about 80% of the whole GC cycles at a maximum, and about 46% of them on average.

Based on this result, we investigate the reason why Mark phase occupies much execution time of GC, and we aim to improve many GC algorithms by supporting this phase with additional hardware.

TABLE I
THE AVERAGE NUMBER OF MARKING FOR THE SAME OBJECTS.

X	1	10	50	100	200
AOBench	345	151	37	20	11
GCBench	61,717	6,312	1,269	636	319
crypto.aes	2,804	854	218	115	61
crypto.signverify	2,801	1,412	349	180	94
compress	627	321	84	45	25
serial	4,269	1,156	313	165	87

B. Object Tracing in DalvikVM

In order to search for the live objects in the heap area, the references which belong to marked objects need to be recursively traced. *Mark Stack*, which manages marked objects, is used in DalvikVM in order to trace references recursively.

After objects are marked, they are pushed into Mark Stack in sequence. When an object is popped from this stack, all objects which are referred by this object are searched for, and they are marked and pushed into the stack. Repeating this routine until Mark Stack becomes empty makes all live objects in the heap area be marked.

However, a simple implementation such that all of marked objects are pushed into Mark Stack can not manage circular references among objects. Therefore, Mark Bitmap, which is a bit string for managing whether each object in the heap area is marked or not, is inspected when the object is marked. Then, if the bit corresponding to the object has been already set, it is inhibited from being pushed into Mark Stack.

Whereas, the determination that the object should be pushed or should not be pushed into Mark Stack is made just before the object is pushed. Thereby, duplicated calculation of the bit position corresponding to a previously traced object can not be omitted, while it is essentially unnecessary, and the calculation is the major part of making routine of GC.

We estimated the frequency of such redundant calculation, or redundant marking, in the same environment mentioned in Section III-A. Therefore, we measured how many times each object had been marked when GC which ran in each benchmark program is completed. We calculated how many times each object is marked, and we summarized the result of the average number of the high-ranking objects in TABLE I. In this figure, X means the number of objects which are used for calculating the average. For example, the column of $X = 10$ shows the average of marks for the top 10 objects.

According to the result, some of the benchmark programs include the objects which are marked more than 1000 times.

This should be one of the causes of the low performance of marking routine in the result which is shown in Section III-A. Additionally, crypto.signverify is proved to include more objects which are frequently and redundantly marked because the value of $X = 10$ is not so smaller than the value of $X = 1$ compared with the other benchmark programs. Thereby, high performance of GC will be achieved by omitting the redundant marking.

IV. Reducing Redundant Marking with Additional Hardware

In this section, we propose a speed-up technique for omitting redundant marking mentioned in Section III, and then we explain its overview and running model.

A. Overview of Proposal

In order to reduce the overhead of GC, which is mentioned in the previous section, we propose a hardware support technique to omit redundant marking. In this section, we introduce the abstract and execution model of our proposal.

In order to omit the redundant marking while GC is running, a new mechanism which makes it possible to judge whether the target object to be marked has already been marked or not is required. For this reason we modify the hardware of the processor, and install a dedicated table in which the marked objects are managed. Additionally, we extend the existing marking phase, and make the objects be marked after the dedicated table is checked. Thus, when the objects which are being marked have been registered with the table, in other words, when the objects have been already marked, the marking calculation for those objects can be omitted. On the other hand, if the target object to be marked has not been registered with the table, the object is registered with the table for omitting future redundant marking for the object.

Here, we show how to omit redundant marking by using the dedicated table in Fig. 2. This figure shows an example where four objects, A to D, in the heap area are marked in sequence. At first, when the object A is marked for the first time (i), it is marked as usual because it has not been registered with the table. At the same time, the address of the heap area which is allocated to the object A is registered with the table in order to omit the future redundant marking for object A (a). As these actions are repeated for all objects to be traced, the objects, A to D, are eventually registered with the table. Therefore, redundant marking for object C is, for example, omitted because it has been already registered with the table when the reference from D to C is traced (ii).

B. How to Manage Entries

As mentioned in Section IV-A, marked objects are managed in the dedicated table to improve the object tracing which can incur a significant overhead in existing DalvikVM. To eliminate all the redundant marking, a dedicated table which can manage all marked objects is required. However, the number of objects which should be managed is different in each program. Therefore, preparing a well huge table for managing all objects in most programs will seriously increase area cost and power consumption, and it will become a concern.

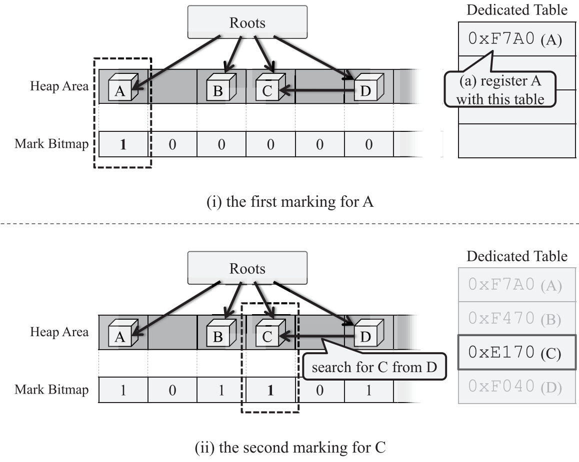

Fig. 2. Omitting a marking by using a dedicated table.

As one of ways to solve this problem, registered objects in the dedicated table should be managed in a list structure with an LRU based eviction algorithm. Thereby, the objects which are frequently marked can be managed preferentially with a little hardware cost.

However, if the list employs a simple LRU that a target object to be marked is inserted into the head of the list in any case, the objects which are frequently marked and should be managed may not be preferentially preserved in the list. For example, in case when many objects which are marked only once are continuously registered with the list after an object which would be marked redundantly was registered with the list, the information about the object may be evicted even though it should be managed.

In our proposal, objects are therefore managed in two dedicated tables which employ an LRU-based eviction algorithm. The algorithm manages objects depending on whether they have been marked redundantly or not. In this paper, we define each of these dedicated tables as *Primary Table* and *Secondary Table*. Before an object is being marked, these dedicated tables are searched in order of *Primary Table*, *Secondary Table*, and whether the marking for the object can be omitted or not is decided. *Secondary Table* is used for managing newly marked objects, and *Primary Table* is used for managing only objects which have been marked twice or more. Thereby, objects which are marked redundantly, that is, objects for which redundant marking can cause a big overhead are preferentially managed in *Primary Table*.

Now, we explain how to manage objects with the two dedicated tables by using an example shown in Fig. 3. This example shows how to manage objects, A to D, which are being traced in sequence. The number of entries in each table is assumed to be three. At first, when the object A is marked for the first time, its address is registered with *Secondary Table*. Repeating this action for the object B and C, three

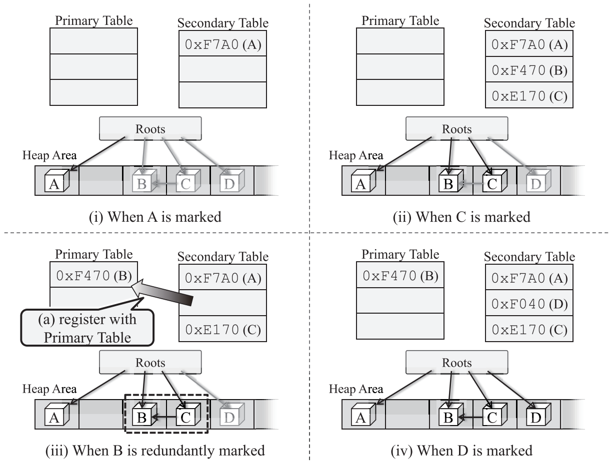

Fig. 3. Managing entries by using two tables.

objects, A to C, will be registered in course of time as shown in (ii). If the object B which has been registered with *Secondary Table* is traced again (iii), it is removed from *Secondary Table* and is registered with *Primary Table* (a). Only redundantly marked objects are managed in *Primary Table* due to such actions. If the object D is marked in this situation (iv), it is registered with *Secondary Table*, and each entry of *Secondary Table* comes to have an object. If another object becomes a new target to be marked in this situation, the oldest object, that is, the object A is evicted from *Secondary Table*.

V. Implementation and Behavior Model

In this section, we describe the implementation of the method to omit redundant marking mentioned in Section IV.

A. Composition of the Dedicated Tables

In this section, we introduce the concrete structure of each dedicated table which is for managing marked objects.

1) Primary Table: In our proposal, objects are managed with two tables which employ the LRU-based algorithm mentioned in Section IV-B. Thereby, when the entries of the table overflows, the oldest object is appropriately evicted from the tables. Because frequently marked objects are managed with *Primary Table*, the redundant marking for the objects should be omitted. In order to achieve this, it needs to be confirmed whether the target object is marked or not with as small latency as possible. Therefore, *Primary Table* is implemented with a CAM (Content Addressable Memory) which is capable of fast associative search.

Primary Table consists of three fields. One is defined as '*Address*,' and holds an address in the heap area which is assigned to a marked object. Another is defined as '*prev*,' and keeps the index of the previous object in the LRU list. The other is defined as '*next*,' and keeps the index of the next object. Each of *prev* and *next* holds the table index corresponding to an object in *Primary Table*. In order to insert an entry to the head of the LRU list and in order to evict an entry from the tail of the list, two types of registers are also installed. These two are defined as '*Head*' and '*Tail*,' and respectively hold the indices of the head and the tail of the list which are managed in *Primary Table*. In order to manage the number of the registered objects, one register is also installed, and this is defined as '*#Addr*.'

2) Secondary Table: Secondary Table is referred when a target object to be marked has not been registered in *Primary Table*. Then, if the object has been registered with *Secondary Table*, the information about it is moved to *Primary Table*. After *Primary Table* is searched for a target object to be marked, the operation for *Secondary Table* is done for the object which has not been registered with *Primary Table*. The latency of the operation itself barely affects the total performance of GC, and the latency can be concealed because this operation is done concurrently with a marking routine for the object. Therefore, *Secondary Table* is implemented with a RAM because it causes lower area overhead and lower power consumption than a CAM while it is not capable of fast associative search like a CAM. However, if *Secondary Table* is naively implemented with a RAM, all entries in the table need to be accessed sequentially, and the search cost for the table becomes significantly large. In our proposal, we therefore employ a set-associative strategy as the structure of *Secondary Table*, and search cost can be decreased by using hash.

Before an object is registered with *Secondary Table*, hash is calculated from the address of the object in order to decide a set where the object should be registered. When the number of registered objects in a set becomes larger than the number of ways along with registration of a new object, any one of entries is overwritten by a new entry. Owing to this, a field '*Victim Index*' for managing the entry which should be overwritten next is provided in every set. This field is implemented with a ring counter which can count to the maximum value of the way number. When an object is registered, the entry which the value of *Victim Index* indicates is overwritten.

When *Secondary Table* is searched by using the address of the object as a key, hash is calculated from the address in order to identify a set. To confirm whether the target object to be searched for has been registered with *Secondary Table* or not, the address of the object is compared with all addresses stored in the entries of the set. The number of comparison is the same as the number of ways at most, and so the comparison overhead can be significantly smaller than the overhead in case when all the entries are accessed sequentially.

B. Operations for the Dedicated Tables

In this section, we explain the operations for the dedicated tables which are mentioned above. Firstly, we explain the required operations in the case when a target object to be marked has been registered with *Primary Table*. Secondly, we explain the required operations in the case when the object has not been registered with *Primary Table*.

1) When the Object has been Registered in Primary Table: When a target object to be marked has been registered with *Primary Table*, that is, when the object has been marked several times, marking this object is omitted. Then, the entry

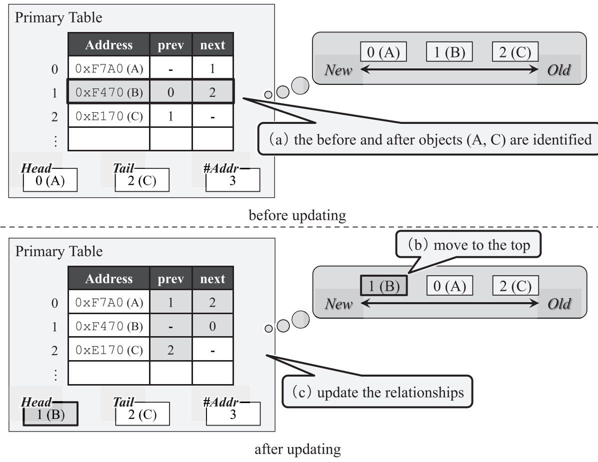

Fig. 4. Operations in case when the object has been registered with *Primary Table*.

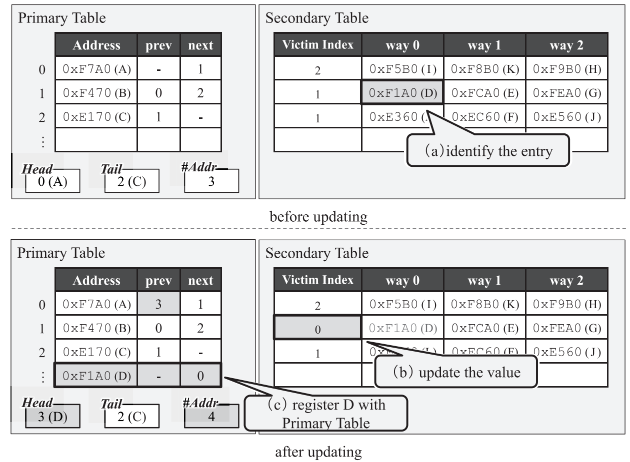

Fig. 5. Operations in case when the object has registered with *Secondary Table*.

corresponding to the object is inserted into the head of the list in accordance with LRU.

Now, we concretely explain operations for *Primary Table* by using Fig. 4. This figure shows operations in case when the object B which has been already registered with *Primary Table* is attempting to be marked again. In this case, the objects A and C, which are located just before and after B in the list respectively, are identified in order to move B to the head of the list after the marking routine for B has been omitted (a). After A and C are identified, B is moved to the head of the list (b). Along with this, *prev* and *next* of objects which are located just before and after B in the list, and registers are updated (c). In this example, because B is moved to the head of the list, the contents of *Primary Table* are updated to make A and C adjoining. In addition, the value of register *Head* which represents the head of the list, the value of *prev* and *next* which B has are updated.

2) When the Object has not been Registered with Primary Table: When a target object to be marked has not been registered with *Primary Table*, it is marked as usual. At the same time, hash is calculated from the address of the object, and *Secondary Table* is searched by using this hash. In case when the object has been registered with *Secondary Table*, it is registered with *Primary Table* in order to omit the future redundant marking for the object. Here, the operations for each dedicated table in this case are shown in Fig. 5. This figure shows the case when D which has been registered with *Secondary Table* is marked again. Firstly, hash is calculated from the address of D, and the set which can contain the address is selected. By comparing the address with ones in the set, the entry which has D is identified (a). Secondly, in order to move D to *Primary Table*, D is removed from *Secondary Table*. Along with the deletion of D, the entry which had D becomes empty, and so *Victim Index* of the set corresponding to the entry is updated to indicate the way number of the entry (b). After that, D is registered with *Primary Table* (c). Along with the registration, the *prev* and *next* of D and the *prev* of A, which was the head of the list, are updated. In addition, each value of the registers is also updated.

When an object is registered with *Primary Table* in case when the table is full, one entry will be reserved by evicting the tail object of the list which is managed by *Primary Table*. The object which is evicted from *Primary Table* is registered with *Secondary Table*. Therefore, when the object is marked again, it will be registered with *Primary Table* immediately.

On the other hand, in case when the target object to be marked has been registered with neither *Primary Table* nor *Secondary Table*, the object should be registered with *Secondary Table*. In order to do it, the set corresponding to the address of the object is identified. Then, the value of *Victim Index* of the set is acquired. Thus, the object is registered with the way indicated by the value. Then the value of *Victim Index* is increased.

TABLE II
EVALUATION ENVIRONMENT.

Platform	ARM-RealView PBX
Processor	ARMv7
Clock	2.0 GHz
Memory	256 MB
OS	Linux 2.6.38.8-gem5

VI. PERFORMANCE EVALUATION

We have evaluated the performance of the hardware support mechanism for GC, with a simulator. In this section, we discuss its effectiveness, and show the estimation of the hardware cost for implementing the mechanism.

A. Simulation Environment

In this evaluation, we used gem5 simulator system [8]. The evaluation environment is shown in TABLE II. Workloads are GCBench [9], AOBench [10], and four benchmarks in SPECjvm2008 suite [11]. Gem5 simulator is a full system simulator, so some other programs run in parallel with the benchmark program. In addition, the behavior of those programs is different every time, and the amount of resources

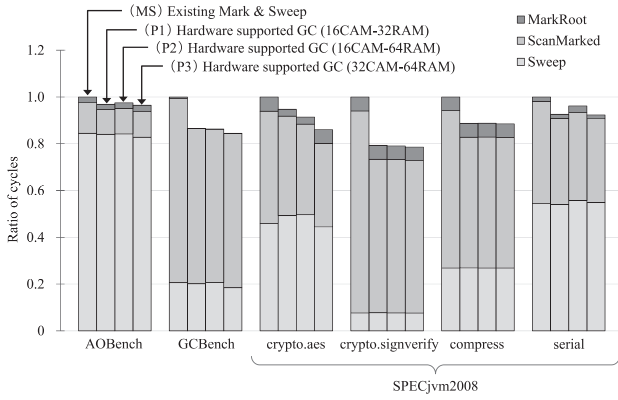

Fig. 6. Ratio of execution cycles of GC.

which are available for the benchmark are also different every time. Thereby, we have to consider the variation of the performance. Thus, we executed each workload several times, and adopt the best result among them in each benchmark program as the result with least disturbance by the other running programs.

Now, we describe the number of the entries which are used in each dedicated table. Because the marking routine for the objects which have been registered with *Primary Table* is omitted, higher performance can be achieved by using larger sized *Primary Table*. However, according to the result which is shown in TABLE I in Section III-B, redundant marking are mainly caused by top dozens of objects which have been marked so many times. In addition, the energy consumption of CAM which *Primary Table* is implemented with is comparatively large, so its size should be as small as possible. Therefore, the size of *Primary Table* in this evaluation was decided to be 16 entries, considering the size of a general TLB which is implemented with a CAM as well is generally from a dozen to several dozens entries. Additionally, it is desirable that the objects which are frequently marked are not evicted from the dedicated tables as much as possible. Therefore, the size of *Secondary Table* was assumed to be double the size of *Primary Table*, that is, the size of *Secondary Table* was assumed to be 32 entries with four ways and 8 sets configuration.

In this evaluation, to compare the differences caused by changing the size of the dedicated tables, we prepared other sized sets of tables. One set is implemented with 16-entry *Primary Table* and 64-entry *Secondary Table*. The other is implemented with 32-entry *Primary Table* and 64-entry *Secondary Table*.

B. Evaluation Results

In this evaluation, we measured the execution cycles of GC, without and with the hardware support mechanism. Additionally, to evaluate how much GC affects the total performance of the system, we measured the average suspension time.

1) Executions Cycles of GC: The total execution cycles of GC is shown in Fig. 6. We have evaluated following one existing GC and three types of hardware supported GCs;

(MS) Existing Mark & Sweep. (baseline)
(P1) Hardware supported GC (16CAM-32RAM)
(P2) Hardware supported GC (16CAM-64RAM)
(P3) Hardware supported GC (32CAM-64RAM)

where '16CAM-32RAM' means that the dedicated tables consist of 16-entry CAM and 32-entry RAM. Each bar is normalized to the execution cycles of (MS). The legend in Fig. 6 shows the breakdown items of total cycles. They represent the cycles for marking objects referred from roots directly ('MarkRoot'), the cycles for marking child objects ('ScanMarked') and the cycles for freeing a part of the heap area where dead objects occupy ('Sweep').

According to the result, with all benchmark programs, the performance of (P1) is higher than (MS). This is because ScanMarked was reduced with hardware support. Especially, with crypto.signverify, ScanMarked accounts for large ratio in the total cycles of (MS). For this reason, (P1) can provide a good performance, and the total GC cycles are reduced by 20.7%. With all benchmark programs, the total GC cycles are reduced by 10.2% on average.

Here, comparing (P1), (P2), and (P3), the results are almost same. This is because most of frequently marked objects can be well managed even with small-sized tables of (P1). However, with crypto.aes, the larger the size of tables is, the better the result is. This is because, in this benchmark program, the number of objects which are marked frequently is much higher than the number of entries which can be managed. Therefore, with some programs, larger-sized tables may be required.

Now, the access latency for the dedicated tables should be considered while the execution cycles of GC are reduced. In this evaluation, we assumed that the access latency for *Primary Table* is two cycles and for *Secondary Table* is one cycle, and estimated the overhead of the hardware supported GC by multiplying the access counts for those two tables by the access latency of each table. As a result, we confirmed that the ratio of the overhead to the cycles for GC with our proposal is about 1.8%, and this is sufficiently small. In addition, operations for *Secondary Table* can be done in parallel with marking routine as described in Section V-A2. Therefore, some of the overhead for operating *Secondary Table* can be concealed, and the substantial overhead is smaller than 1.8%.

2) Average Suspension Time caused by GC: Next, the average suspension time caused by GC is shown in Fig. 7. We

TABLE III
THE FREQUENCY OF SUSPENSION IN EACH PROGRAM.

	(MS)	(CO)	(P1)
AOBench	3,363	8,937	3,364
GCBench	235	467	235
crypto.aes	198	279	198
crypto.signverify	40	42	20
compress	20	29	20
serial	357	693	356

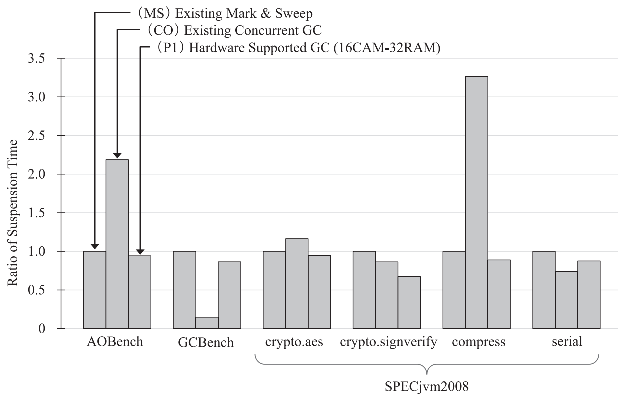

Fig. 7. The average suspension time caused by GC.

calculated the average of suspension time per GC execution by dividing the total sum of suspended time by the frequency of suspension shown in TABLE III. The evaluation results with following three GCs are shown in Fig. 7.

(MS) Existing Mark & Sweep. (baseline)
(CO) Existing Concurrent GC
(P1) Hardware supported GC (16CAM-32RAM)

Each bar in Fig. 7 is normalized to the result of (MS). According to the result, with many benchmark programs, the suspension time is reduced with (P1). This is because the cycles per GC execution are reduced by hardware support. Although Concurrent GC (CO) aims to shorten the suspension time at the cost of the throughput as mentioned in Section II-B, the suspension time significantly increases with AOBench and compress. This is because GC consumes relatively small cycles with these programs and synchronization overhead appears large. Even with these programs, the suspension time is restrained with (P1). With all the benchmark programs, we confirmed that the average suspension time is reduced by about 32.7% at a maximum, and about 13.5% on average.

C. Estimation of the Hardware Cost

In this section, we discuss the hardware cost which appears in the hardware supported GC. As mentioned in Section VI-A, *Primary Table* with 16 entries and *Secondary Table* with four ways eight sets configuration are used. On one hand, 32 bits are required as the width of each field in *Primary Table* for managing the addresses of marked objects, and two four-bit fields are required for managing indices of objects which are registered just before and after an object. In addition, four bits are required for each register, which manages the head of the list, the tail of the list, or the number of objects in the *Primary Table*. Hence, *Primary Table* can be configured in 80 Bytes CAM and three four-bit registers.

On the other hand, *Secondary Table* has *Victim Index* and the field which holds the addresses of the objects, and the number of addresses which can be held in a set of *Secondary Table* is the same as the number of ways. In this evaluation, the number of ways is defined as four, and *Victim Index* can be implemented with a 2-bit counter. Therefore, each set of *Secondary Table* requires two bits for the counter and $32 \times 4 = 128$ bits for managing the addresses of the objects. Hence, *Secondary Table* can be configured in 130 Bytes RAM. Thus, the total hardware cost is only 210 Bytes, and we confirmed that the hardware cost of proposed method is quite small.

VII. Conclusion

In this paper, we proposed a hardware supported technique to improve the object tracing routine which is required in many GC algorithms. In this technique, we added dedicated tables to a processor for managing marked objects. Thereby, the overhead caused by conventional redundant marking was reduced by referring these tables while GC is running, and we achieved high performance GC.

To validate the effectiveness of the hardware supported GC, we evaluated the performance with a simulator. As a result, we confirmed that the total GC cycles were reduced by 20.7% at a maximum compared with previous Mark & Sweep. Additionally, while the throughput decreases and the suspension time increases with some benchmark programs with Concurrent GC, we confirmed that the hardware supported GC can suppress such performance degradation.

One of our future work is to study a new GC algorithm which is not ridden by conventional algorithms, that is, which can highly cooperate with some assistant hardware. In this paper, we proposed a hardware supported GC, but only object tracing routine of whole GC execution is improved. Therefore, in case when the marking routine occupies little part of GC in a program, this technique can not improve the performance effectively. Hence, we will study a new GC algorithm which is optimized for using additional hardware, and would like to find the way which leads to the dramatic improvement of GC performance.

References

[1] J. McCarthy, “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I,” *Communications of the ACM*, vol. 3, pp. 184–195, Apr. 1960.

[2] M. Minsky, “A LISP Garbage Collector Algorithm Using Serial Secondary Storage,” Massachusetts Institute of Technology, Tech. Rep., 1963.

[3] G. E. Collins, “A Method for Overlapping and Erasure of Lists,” *Communications of the ACM*, vol. 3, pp. 655–657, Dec. 1960.

[4] S. Nakamura *et al.*, *Garbage Collection - Algorithms and Implementations (Japanese)*. SHUWA SYSTEM CO.,LTD, 2010.

[5] Y. Ossia, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, and A. Owshanko, “A Parallel, Incremental and Concurrent GC for Servers,” in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI’02)*, May. 2002, pp. 129–140.

[6] I. Takeuchi *et al.*, “Lisp can be “Hard” Real Time,” in *Proc. Japan Lisp User Group Meeting (JLUGM)*, May. 2000.

[7] C. Click, G. Tene, and M. Wolf, “The Pauseless GC Algorithm,” in *Proc. 1st ACM/USENIX Int’l Conf. on Virtual Execution Environments (VEE’05)*, 2005, pp. 46–56.

[8] N. Binkert *et al.*, “The gem5 Simulator,” *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 1–7, May. 2011.

[9] H. Boehm, “An artificial garbage collection benchmark,” http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench.html.

[10] S. Fujita, “Ambient occlusion benchmark,” http://code.google.com/p/aobench/.

[11] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko, “SPECjvm2008 Performance Characterization,” in *Proc. SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, 2009, pp. 17–35.