IEICE TRANSACTIONS

on Information and Systems

VOL. E101-D NO. 4
APRIL 2018

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER

Hardware Accelerated Marking for Mark & Sweep Garbage Collection

Shinji KAWAMURA[†], *Nonmember and* **Tomoaki TSUMURA**[†a)], *Member*

SUMMARY Many mobile systems need 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 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 implemented on DalvikVM, which is 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 redundant marking for marked objects can be omitted by referring these tables. The result of the simulation experiment shows that the percentage of redundant marking is reduced by more than 50%.

key words: *garbage collection, hardware acceleration, mark & sweep, energy efficient implementation*

1. Introduction

In many of mobile systems such as smart phones, virtual machine environments, or Web APIs are commonly adopted in terms of effectiveness for developing applications. However, *Garbage Collection (GC)* routines occupy a large part of their running time in these environments. Alongside, it is well known that GC can degrade the response time of a system, because any other processes generally have to halt while GC routines are running. Therefore, the effectiveness of GC now can affect the total performance of the wide range of platforms. Furthermore, performance degradation has a negative effect on energy consumption, and it is a serious problem for mobile devices. To address such a problem, many GC algorithms have been studied from the view of their software. However, most of the studies are for reducing the frequency of collections by complicated and dedicated tuning for the systems and the applications, or for reducing the response time of the system at the cost of the throughput. Thus, neither algorithm can be the way for fundamental solution.

For this reason, we have focused on the basic routines that many GC algorithms commonly have, and aim to achieve high performance in many GC algorithms with hardware supports. To find which routine can be improved with hardware supports, we firstly investigated the GC implementation of 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 GC routines tracing references between the objects. Additionally, we also found that many objects are marked redundantly. In this paper, we propose a hardware support technique [1] to omit such redundant marking for marked objects, and evaluate it by estimating the performance and energy consumption. 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 that 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 disclose that GC includes redundant marking, and some objects are marked several times per collection.
2. We propose a hardware support for common GCs. We install dedicated tables to a processor, and redundant marking can be omitted by using the tables.
3. We evaluate the proposed method. The results show that the execution cycles can be reduced by 22.4% at a maximum, and 13.8% on average.

2. Background

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

2.1 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 that are located in the heap area may have pointers to some other objects. Those objects that are referred by another object can be reached from the set of roots, and are called *live objects*. On the other hand, the

Manuscript received May 18, 2017.
Manuscript revised October 20, 2017.
Manuscript publicized January 15, 2018.

[†]The authors are with Nagoya Institute of Technology, Nagoya-shi, 466–8555 Japan.
a) E-mail: tsumura@nitech.ac.jp

DOI: 10.1587/transinf.2017EDP7163

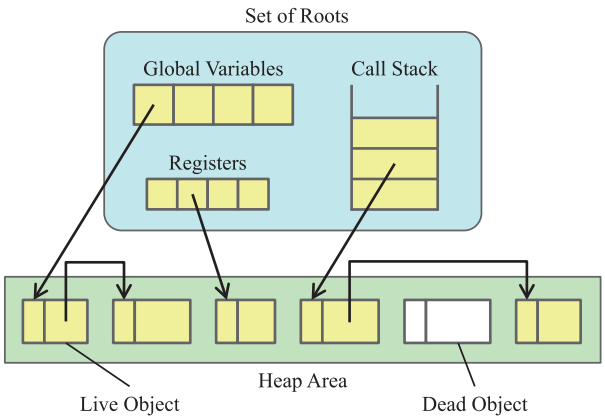


Fig. 1 Heap Area and References during program execution.

unused objects that cannot 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 [2] is one of the representative GC algorithms. It consists of two phases. The first phase is defined as a Mark phase which marks encountered live objects, and the second phase is defined as a Sweep phase which reclaims the space of unmarked, or dead objects. These two phases are executed repeatedly.

Copying [3] and Reference Counting [4] are also the representatives of GC algorithms. Although many other GC algorithms have been studied, all of them are just combinations or improvements of those three algorithms [5]. Especially, Mark & Sweep is widely used as a base of many algorithms because it is comparatively easy to be implemented.

2.2 Related Work

Concurrent GC [6] is a representative algorithm of GC. This algorithm aims to shorten suspension period of the system by running in parallel with other applications. While GC is being executed in parallel with other applications, some objects cannot be marked properly because some pointers may be modified by the applications. To prevent this issue, barrier synchronization is used in Concurrent GC for detecting the modifications on pointers and starting marking again from the pointers. This allows Concurrent GC to run in parallel with other applications. Whereas, the throughput will be decreased because of the overhead that is caused by synchronization between GC and other applications.

Although most of the previous work focus on the software algorithm as mentioned above, a few studies such as SILENT [7] and Network Attached Processing (NAP) [8] have proposed hardware support techniques. They let GC run in parallel with other applications, as well as Concurrent GC, and achieve high performance with hardware acceleration for barrier synchronization.

3. Bottleneck Analysis of GC

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

3.1 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 of GC. We used gem5 simulator system [9], and executed DalvikVM on this simulator to measure the execution cycles of each routine of GC. We measured the execution cycles of AOBench [10], GCBench [11], and four benchmarks in SPECjvm2008 suite [12]. 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.

3.2 Object Tracing in DalvikVM

In order to search for the live objects in the heap area, the references that belong to marked objects need to be recursively traced. *Mark Stack* 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 referred by this object are searched for, and they are marked and pushed into the stack. By repeating this routine until Mark Stack becomes empty, all live objects in the heap area are marked.

Now, pushing every marked object onto Mark Stack causes a problem if there are circular references among objects. Therefore, when marking an object, GC routine inquires Mark Bitmap; each bit of it represents the corresponding object is already marked or not. If the bit is set, the object is already marked and should not be pushed onto Mark Stack. With this mechanism, each object can avoid being pushed redundantly. However, to know whether an object is already pushed or not, it is inevitable to calculate the bit position corresponding to the object. If an object is marked repeatedly, this redundant calculation for the same object can largely degrade the GC performance.

We estimated the ratio of such redundant calculation, or redundant marking in the same environment mentioned in Sect. 3.1. For the estimation, we measured how many times each object had been marked every time when GC that ran in each benchmark program is completed. The ratio of redundant marking is summarized in Table 1.

According to the result, the average ratio is over 60%

Table 1 The ratio of redundant marking for the same objects.

	maximum	average
AOBench	82.17 %	60.67 %
GCBench	99.99 %	68.03 %
crypto.aes	73.00 %	61.12 %
crypto.signverify	66.25 %	61.28 %
compress	64.05 %	61.51 %
serial	67.88 %	61.83 %

with all benchmark programs, and the maximum ratio exceeds even 70% with many programs. This should be one of the causes of the low performance of marking routine described in Sect. 3.1. Thereby, high performance of GC will be achieved by omitting the redundant marking.

4. Reducing Redundant Marking Calculation with Additional Hardware

In this section, we propose a speed-up technique for omitting redundant marking calculation described in Sect. 3, and then we explain its overview and running model.

4.1 Overview of Proposal

We propose a hardware support technique to omit redundant calculation for 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, we install a dedicated table in which marked objects are managed. We also extend the existing marking phase of GC, and make objects marked after the dedicated table is checked. Thus, when an object which is being marked is on the table, in other words, when the object has been already marked, the marking calculation for the object can be omitted. On the other hand, if the target object to be marked is not on the table, the object is registered with the table for omitting future redundant calculation for the object.

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

4.2 Table Management

As explained in Sect. 4.1, marked objects are managed in the dedicated table to improve the object tracing which can incur a significant overhead in existing Mark & Sweep on DalvikVM. To eliminate all the redundant marking, a dedicated table which can manage all marked objects is required.

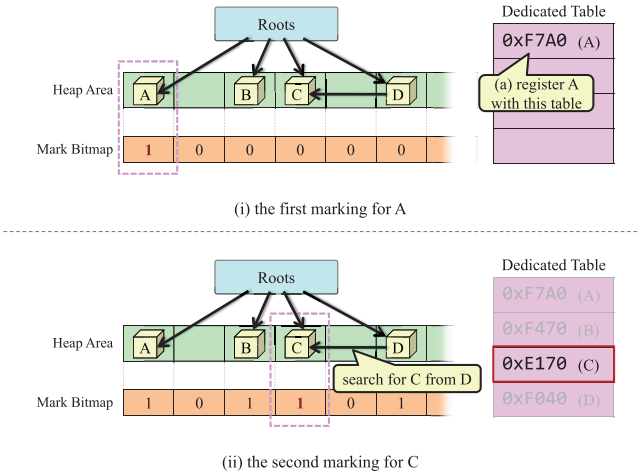


Fig. 2 Omitting a marking by using a dedicated table.

However, how many objects need to be managed is different in each program. Therefore, preparing a well huge table that can manage all objects in any programs will seriously increase area cost and power consumption, and it will become a concern.

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 that are frequently marked can be managed preferentially with a little hardware cost.

However, if the list employs a simple LRU such that a target object to be marked is inserted into the head of the list in any case, the objects that are frequently marked and should be managed may not be preferentially preserved in the list. For example, in case when many objects that are marked only once are continuously registered with the list after an object which would be marked redundantly was registered with the list, 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 tables are used depending on whether objects are marked redundantly or not. In this paper, we define each of these dedicated tables as *Primary Table* (*Pri-Table*) and *Secondary Table* (*Sec-Table*). Before an object is marked, these tables are searched in order of *Pri-Table*, *Sec-Table*, and whether the marking for the object can be omitted or not is decided. *Sec-Table* is used for managing newly marked objects, and *Pri-Table* is used for managing only objects that have been marked twice or more. Thereby, objects that are marked redundantly, or objects for which redundant marking can cause a big overhead are preferentially managed in *Pri-Table*.

The example in Fig. 3 shows how to manage objects, *A* to *D*, which are being traced in sequence. The number of entries in each table is assumed three in this example. At first, when *A* is marked for the first time, its address is registered with *Sec-Table*. Repeating this action for *B* and *C*, three ob-

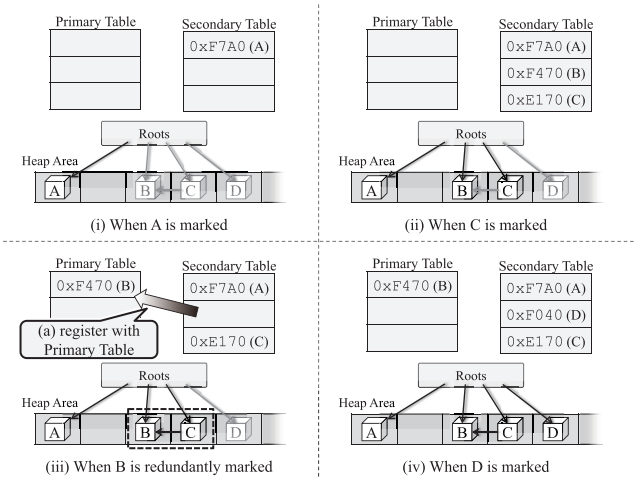


Fig. 3 Managing entries by using two tables.

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

5. Implementation and Behavior Model

In this section, we describe the implementation of the method to omit redundant marking explained in Sect. 4.

5.1 Composition of the Dedicated Tables

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

5.1.1 *Pri-Table*

In our proposal, objects are managed with two tables that employ the LRU-based algorithm mentioned in Sect. 4.2. Thereby, before an entry spills from the table, the oldest object is evicted from the tables. Frequently marked objects are managed in *Pri-Table*, and the redundant marking for them 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, *Pri-Table* is implemented with a CAM (Content Addressable Memory) which is capable of fast associative search.

Each entry of *Pri-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 entry index corresponding to an object in *Pri-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 *Pri-Table*. In order to manage the number of the registered objects, one register is also installed, and this is defined as '*#Addr*.'

5.1.2 *Sec-Table*

After *Pri-Table* is searched for a target object to be marked, *Sec-Table* is referred if the object proves not to be on *Pri-Table*. If the object is on *Sec-Table*, it is moved to *Pri-Table* for omitting future redundant calculation for it. The latency for accessing *Sec-Table* barely affects the total performance of GC and can be concealed, because it can be parallelized with marking routine for the object. Therefore, *Sec-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. However, if *Sec-Table* is naively implemented with a RAM, all entries in the table need to be accessed sequentially, and the search cost for the table may become significantly large. In our proposal, we therefore employ a set-associative strategy as the structure of *Sec-Table*, and search cost can be decreased by using hash.

Before an object is registered with *Sec-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 which entry 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 that the value of *Victim Index* indicates is overwritten.

When *Sec-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 is on *Sec-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.

5.2 Operations for the Dedicated Tables

In this section, we explain the operations for the dedicated tables that are mentioned above. Firstly, we explain the required operations in the case when a target object to be marked is found in *Pri-Table*. Secondly, we explain the required operations in the case when the object is not found in *Pri-Table*.

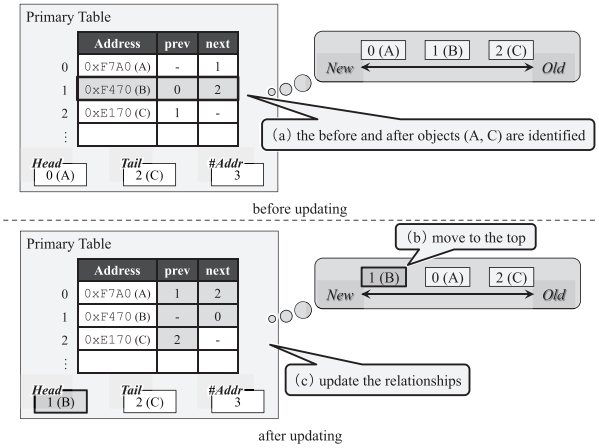


Fig. 4 Operations in case when the object has been registered with *Pri-Table*.

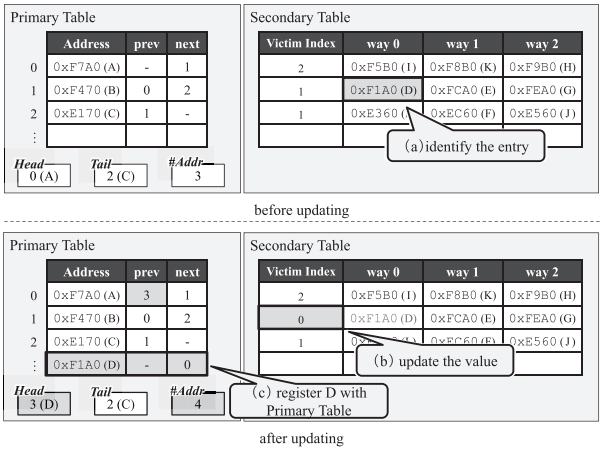


Fig. 5 Operations in case when the object has registered with *Sec-Table*.

5.2.1 When found in *Pri-Table*

When a target object to be marked has been registered with *Pri-Table*, marking this object is omitted. Then, the entry corresponding to the object is inserted into the head of the list in accordance with LRU.

Now, we concretely explain operations for *Pri-Table* by using Fig. 4. This figure shows operations in case when object *B* on *Pri-Table* is attempting to be marked again. In this case, 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 *Pri-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.

5.2.2 When not found in *Pri-Table*

When a target object to be marked has not been registered with *Pri-Table*, it is marked as usual. At the same time, hash is calculated from the address of the object, and *Sec-Table* is searched by using this hash. In case when the object has been registered with *Sec-Table*, it is registered with *Pri-Table* in order to omit the future redundant marking for the object. The operations for each dedicated table in this case are illustrated in Fig. 5. This figure shows the case when *D* which has been registered with *Sec-Table* is marked again. Firstly, hash is calculated from the address of *D*, and the set that can contain the address is determined. By comparing the address with ones in the set, the entry that has *D* is identified (a). Secondly, in order to move *D* to *Pri-Table*, *D* is removed from *Sec-Table*. Along with the deletion of *D*, the entry in which *D* was stored 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 *Pri-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 moved to *Pri-Table* and *Pri-Table* overflows, one entry will be reserved by evicting the tail object of the list. The object evicted from *Pri-Table* is moved to *Sec-Table*, and if the object is marked again, it can revert to *Pri-Table* immediately.

On the other hand, in case when the target object to be marked is on neither *Pri-Table* nor *Sec-Table*, the object should be registered with *Sec-Table*. In order to do this, the set corresponding to the address of the object is identified. Then, the object is registered with the way that is indicated by *Victim Index* of the set, and the value of *Victim Index* is increased.

5.3 Software Interface

For using the dedicated tables from software GC routines, we install two dedicated instructions shown below. Now, we assume ARM instruction set architecture [13], which uses condition flags for branch instructions.

GC_SRCH *Rn* searches *Pri-Table* for the object address stored in the operand register *Rn*. This instruction is followed by a conditional branch instruction whose branch target is the end of the marking routine. If the object address is found on *Pri-Table*, a condition flag is set for the subsequent branch instruction being taken. Which condition flag is adopted is implementation-dependent, but in this paper we assume to utilize N(negative)-flag, and use BMI (branch if minus or negative) as the branch instruction. If not found, the object address is managed on the dedicated table.

GC_CLR flushes all entries in the dedicated tables, at the end of the GC routine, to prepare for next GC.

When a GC_SRCH instruction is executed, *Pri-Table* is

firstly searched as explained in Sect. 5.2.1. If the target address is found, N-flag is set, and the subsequent BMI will be taken for omitting the following redundant marking routine. On the other hand, if the target address is not found on *Pri-Table*, *Sec-Table* is searched, and the operations explained in Sect. 5.2.2 are performed.

The dedicated tables are installed on the backend of the instruction pipeline. Therefore, some wrong-path instructions may be in-flight in the pipeline when the search result by a GC_SRCH is ascertained. Such a case appears as that the branch prediction for the BMI just after the GC_SRCH misses, and the pipeline will be flushed.

By using these dedicated instructions in the GC routine, the hardware accelerated marking can be utilized. In the marking routine, a pair of a GC_SRCH and a BMI should be located just after the instructions that are for determining the address of an object being marked, and the register where the address is stored should be used as the operand of the GC_SRCH. Alongside, a GC_CLR should be located at the end of the GC routine.

For the performance evaluation shown in Sect. 6, we modified the GC routine of DalvikVM as it uses these dedicated instructions. The instructions are not specialized for DalvikVM, and they can be used even for other GC routines other than of DalvikVM. We suppose that a GC library is provided, with which users can easily utilize the instructions, and the hardware supported GC will be available in many environments.

6. Performance Evaluation

We have evaluated the performance of the hardware support mechanism for GC, with a simulator. We also have evaluated the energy consumption because it is significantly important for mobile devices. In this section, we discuss its effectiveness, and show the estimation of the hardware cost for implementing the mechanism.

6.1 Simulation Environment

In this evaluation, we used gem5 simulator system [9]. The simulated platform system is shown in Table 2. Workloads are AOBench [10], GCBench [11], and four benchmarks in SPECjvm2008 suite [12]. Gem5 simulator is a full system simulator, and some other programs run in parallel with the benchmark program. In addition, the behavior of those programs varies every time, and the amount of available resources for the benchmark also varies every time. Thereby, for considering the performance variation, we executed each workload several times, and adopted the best result among them in each benchmark program as the result with least disturbance by the other running programs.

6.2 Evaluation Results

In order to discuss the size of dedicated tables, we firstly measured the percentage of redundant marking in the total number of marking, without and with the hardware support mechanism. Then we measured the execution cycles of GC. To evaluate how much GC affects the total performance of the system, we also measured the average suspension time. At last, we evaluate the energy consumption by using McPAT [14].

Table 2 Simulated platform.

Platform	ARM-RealView PBX
Processor	ARMv7
Clock	2.0 GHz
L1I Cache	32 KB
ways	4 ways
L1D Cache	32 KB
ways	4 ways
L2 Cache	1 MB
ways	8 ways
Memory	256 MB
OS	Linux 2.6.38.8-gem5
Execution	non-pipelined / in-order

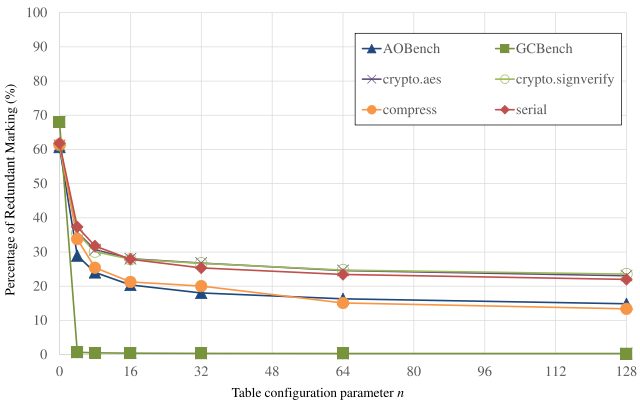


Fig. 6 Percentage of redundant marking.

6.2.1 Percentage of Redundant Marking

The precentage of redundant marking on the all objects that are used in each benchmark program is illustrated in Fig. 6. In this figure, horizontal axis represents the size of dedicated tables. The label n on horizontal axis means *Pri-Table* has n-entries and *Sec-Table* has $2n$-entries, and we have evaluated with $n = 0, 4, 8, 16, 32, 64$, and 128. Here, $n = 0$ indicates the existing Mark & Sweep, and the plotted values of $n = 0$ correspond to the average values shown in Table 1.

As we can see in Fig. 6, with all benchmark programs, the ratio of redundant marking decreases as n increases, though the decrements are very small in $16 \leq n$. Especially, with GCBench, our GC reduces more than 90% of redundant marking even with $n = 4$. In this program, many class objects and array objects are allocated all at once, and after a while, they simultaneously become dead objects. We found that such objects are commonly traced several times in a short time while they are alive. Those objects are managed with *Pri-Table* while they are marked frequently, and

redundant marking for them was omitted.

We measured how much cycles are consumed for executing marking routine once, in other words, how much cycles can be reduced by omitting one redundant marking. On DalvikVM, we found that it is around 71-cycle on average, and 54-cycle even at a minimum. It is pretty large and many redundant marking can severely decrease the total performance of the systems.

Here, we would like to discuss the ideal size of dedicated tables. Larger sized *Pri-Table* can achieve higher performance, because the marking routine for only the objects that are on *Pri-Table* can be omitted. However, the energy consumption of CAM that *Pri-Table* is implemented with is comparatively large, so its size should be as small as possible. We assumed the ideal number of entries of *Pri-Table* as 16 for the following evaluations, taking the result shown in Fig. 6 and considering that a general TLB which is implemented with a CAM as well has generally from a dozen to several dozen entries. It is also desirable that the frequently marked objects are not evicted from the dedicated tables as much as possible. Therefore, the size of *Sec-Table* was assumed to be double the size of *Pri-Table*, that is, 32 entries with four ways and eight sets configuration.

In the following, we use the set of 16-entry *Pri-Table* and 32-entry *Sec-Table* as the ideal configuration, and two reference configurations. The one is the set of 16-entry *Pri-Table* and 64-entry *Sec-Table*, and the other is the set of 32-entry *Pri-Table* and 64-entry *Sec-Table*. To sum up, we have evaluated following one existing GC and three configurations of hardware supported GC;

(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 table set consists of 16-entry *Pri-Table* and 32-entry *Sec-Table*, and so forth.

6.2.2 Execution Cycles of GC

The total execution cycles of GC is illustrated in Fig. 7. Each bar is normalized to the execution cycles of (MS) the baseline. The legend in Fig. 7 shows the breakdown items of total cycles. They represent;

MarkRoot the cycles for marking objects directly referred from roots.
ScanMarked the cycles for marking child objects
Sweep the cycles for freeing a part of the heap area where dead objects occupy.

According to the result, with all benchmark programs, the performance of (P1) is higher than (MS). This is because ScanMarked was reduced with the 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 22.4%. With all benchmark programs, the total GC cycles are reduced by 13.8% on average. The Sweep cycles are slightly different between the configurations. The variance should be a deviation caused by the disturbance from other running processes as mentioned in Sect. 6.1.

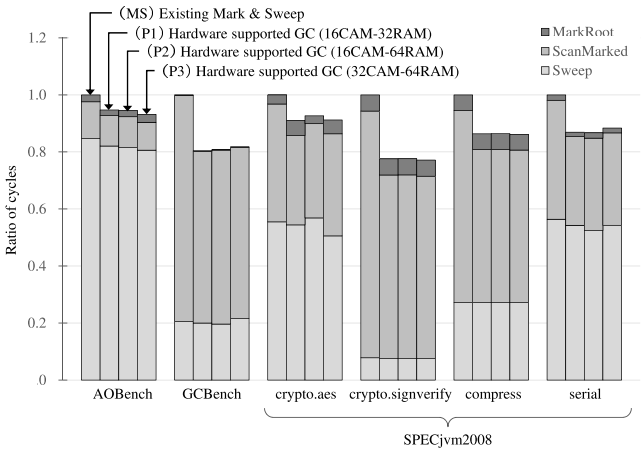


Fig. 7 Ratio of execution cycles of GC.

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 *Pri-Table* is two cycles and for *Sec-Table* is one cycle, and calculated 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, *Sec-Table* access can be overlapped with marking routine as described in Sect. 5.1.2. Therefore, the substantial overhead will be smaller than 1.8%.

The results do not much differ between (P1), (P2) and (P3). This indicates that most of frequently marked objects can be well managed even with the set of 16-entry CAM and 32-entry RAM of (P1), and is consistent with the discussion in Sect. 6.2.1. With some benchmark programs, the results of (P1) are slightly better than the others. This is because that the larger *Sec-Table* can manage many entries including the entries that do not contribute much to omit redundant marking, and even beneficial *Pri-Table* entries can be evicted to *Sec-Table* more frequently by such entries. The redundant marking on the objects corresponding to the evicted entries can be omitted again only after the entries return to *Pri-Table*. Therefore, we will show the results of existing models and only (P1) in the following evaluations.

6.2.3 Average Suspension Time caused by GC

Next, we show the average suspension time caused by GC in Fig. 8. We calculated the average of suspension time per GC execution by dividing the total length of suspended time by the frequency of suspension shown in Table 3. The results with following three GCs are shown in Fig. 8.

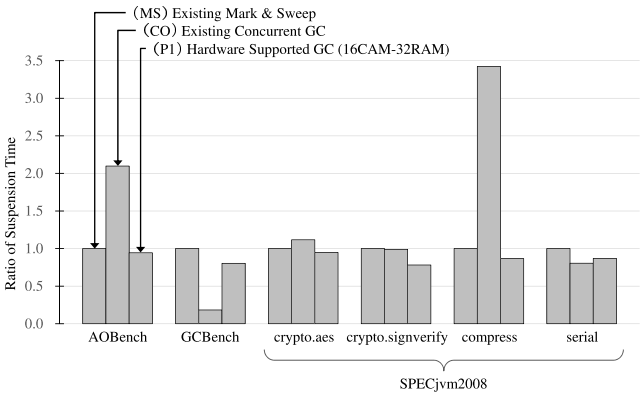


Fig. 8 Average suspension time caused by GC.

Table 3 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

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

Each bar in Fig. 8 is normalized to (MS). The results show that, with many benchmark programs, the suspension time of (P1) is smaller than (MS). 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 Sect. 2.2, its suspension time severely increases with AOBench and compress. This is because GC consumes relatively small cycles with these programs and synchronization overhead for Concurrent GC appears large. Even with these programs, the suspension time of (P1) is restrained. With all the benchmark programs, we confirmed that the average suspension time is reduced by about 21.8% at a maximum, and about 13.1% on average.

6.2.4 Estimation of the Hardware Cost

Here, we discuss the additional hardware cost for the hardware supported GC before introducing the evaluation result of energy consumption. As mentioned in Sect. 6.1, *Pri-Table* with 16 entries and *Sec-Table* with four ways eight sets configuration are used. Each entry in *Pri-Table* requires 32-bit width for *Address* field, and (2×4)-bit width for *priv/next* fields. In addition, three 4-bit width registers *Head*, *Tail*, and *#Addr* are required. Hence, *Pri-Table* can be configured in an 80-Byte CAM and three 4-bit registers.

Sec-Table has *Victim Index* and a field set for holding object addresses. The number of addresses which can be held in a set of *Sec-Table* is the same as the number of ways.

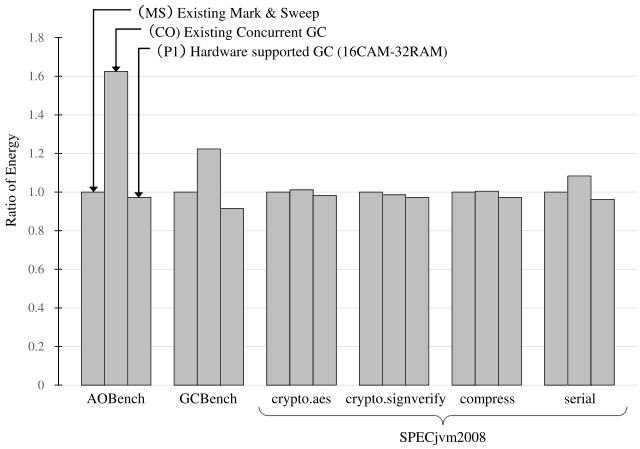


Fig. 9 Energy Consumption.

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 *Sec-Table* requires two bits for the counter and $32 \times 4 = 128$-bits for managing the addresses of the objects. Hence, *Sec-Table* can be configured in a 130-Byte RAM.

Thus, the total hardware cost is only 210 Bytes, and we confirmed that the hardware cost of proposed method is quite small.

6.2.5 Energy Consumption

At last, the evaluation result of energy consumption in consideration of the additional hardware is shown in Fig. 9. Here, we note that the processor model we used in this evaluation includes only minimal units for executing instructions in non-pipelined in-order manner. Therefore, the ratio of the power consumption by additional hardware can be overestimated, because the base architecture of processor model is simpler than general commodity processors.

The result indicates that the proposed GC can reduce the energy consumption with all benchmark programs. Even though the required additional hardware for our proposal raises power consumption, the amount is quite small, and the total energy consumption results in being even smaller than (MS) by execution cycles being largely reduced.

7. Conclusion

In this paper, we proposed a hardware supported technique to improve the object tracing routine that is required in many GC algorithms. In this technique, we installed 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 percentage of redundant marking was reduced by more than 50% compared with previ-

ous Mark & Sweep, and the total GC cycles were reduced by 22.4% at a maximum. 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.

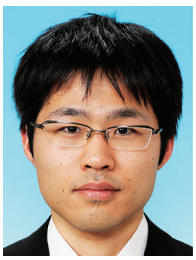
One of our future works is to study a new GC algorithm which is not ridden by conventional algorithms, that is, which can highly cooperate with some assistant hardware. In this paper, we proposed a hardware supported GC, but only object tracing routine of whole GC execution is improved. Therefore, in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

Acknowledgments

This work was partially supported by JSPS KAKENHI Grant Numbers JP25540019, JP17H01711, JP17K19971, and JP17H01764.

References

[1] S. Kawamura and T. Tsumura, "Hardware supported marking for common garbage collections," Proc. 4th Int'l Workshop on Computer Systems and Architecture (CSA'16), pp.381–387, Nov. 2016.
[2] J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," Communications of the ACM, vol.3, no.4, pp.184–195, April 1960.
[3] M. Minsky, "A LISP Garbage Collector Algorithm Using Serial Secondary Storage," tech. rep., Massachusetts Institute of Technology, 1963.
[4] G.E. Collins, "A Method for Overlapping and Erasure of Lists," Communications of the ACM, vol.3, no.12, pp.655–657, Dec. 1960.
[5] S. Nakamura et al., Garbage Collection - Algorithms and Implementations (in Japanese), SHUWA SYSTEM CO., LTD, 2010.
[6] Y. Ossia, O. Ben-Yitzhak, I. Goft, E.K. Kolodner, V. Leikehman, and A. Owshanko, "A Parallel, Incremental and Concurrent GC for Servers," Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'02), pp.129–140, May 2002.
[7] I. Takeuchi et al., "Lisp can be "Hard" Real Time," Proc. Japan Lisp User Group Meeting (JLUGM), no.14, May 2000.
[8] C. Click, G. Tene, and M. Wolf, "The Pauseless GC Algorithm," Proc. 1st ACM/USENIX Int'l Conf. on Virtual Execution Environments (VEE'05), pp.46–56, 2005.
[9] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M.D. Hill, D.A. Wood, B. Beckmann, G. Black, S.K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D.R. Hower, and T. Krishna, "The gem5 Simulator," ACM SIGARCH Computer Architecture News, vol.39, no.2, pp.1–7, May 2011.
[10] S. Fujita, "Ambient occlusion benchmark." http://code.google.com/p/aobench/.
[11] H. Boehm, "An Artificial Garbage Collection Benchmark." http://hboehm.info/gc/gc_bench.html.
[12] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko, "SPECjvm2008 Performance Characterization," Proc. SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking, vol.5419, pp.17–35, 2009.
[13] ARM Ltd., "Arm architecture reference manual armv7-a and armv7-r edition." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406b/index.html, July 2012.
[14] S. Li, J.H. Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," 42nd Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO-42), pp.469–480, Dec. 2009.



Shinji Kawamura received his B.E. degree from Nagoya Institute of Technology in 2015. Currently, he is a 2nd year graduate student in the Department of Scientific and Engineering Simulation, Nagoya Institute of Technology, Japan. His current research interest is computer architecture. He is a student member of IPSJ.

Tomoaki Tsumura received his M.E. and Ph.D. degree from Kyoto University in 1998 and 2004 respectively. After graduating from the Ph.D. candidate course of the Graduate School of Informatics, Kyoto University in 2001, he joined the university as a research associate. He joined Toyohashi University of Technology in 2004, and then joined Nagoya Institute of Technology as an associate professor in 2006. His current research interests are computer architecture, applications of parallel processing, and brain-type information processing. He is a member of ACM, IEEE-CS, IPSJ and IEICE.