This paper is author's private version of the paper published as follows:
Proc. 1st Int'l Symp. on Computing and Networking (CANDAR'13), pp.134-140

Hardware-Supported Pointer Detection for common Garbage Collections

Kei IDEUE*, Yuki SATOMI*, Tomoaki TSUMURA* and Hiroshi MATSUO*
*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 the wide range of platforms now can be affected by the effectiveness of Garbage Collection (GC). 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 objects on the call stack should be traced in many GC algorithms, and propose a hardware support technique for speed up of this trace. To trace objects, it is needed to find pointers on the call stack. Hence, we install tables for managing all pointers on the call stack. By referring these tables, the GC routine can detect pointers immediately. The result of the simulation experiment shows the proposed method leads to low GC latency.**

***Index Terms*—garbage collection, hardware support, pointer detection**

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). On the other hand, it has been well known that GC has a big effect on the performance of the systems such as server-side Java Runtime Environment. In particular, GC can deteriorate the system response time because GC has to halt all other processes for running. To solve this problem, Concurrent GC[1] has been proposed. Concurrent GC can run in parallel with application, and reduces the maximum suspension time of the system while GC is running. However, Concurrent GC is not suitable for the system which is required a high throughput, because it has a problem of decreasing the throughput by parallelizing GC and application processes. Therefore, it has been needed to tune the algorithm or parameters with considering the purpose of use and features of each system, and this has been a burden to users. In this paper, we focus on the point that many GC algorithms commonly trace through the working set of memory, and propose a hardware support technique for speed-up of the trace. We aim to improve the performance of many GC algorithms with a hardware support for the trace. With the hardware support, the GC performance can be improved by the cooperation between software and hardware without tuning. Therefore, it is also expected that the hardware support technique can take the burden of the tuning from the user.

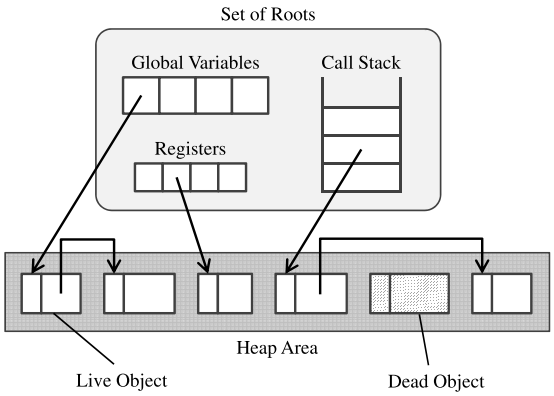

Fig. 1. An example of the heap area and references.

II. GARBAGE COLLECTION

Garbage Collection (GC) is a routine for automatic memory management. It automatically frees the unnecessary part of the heap area where the data has been located dynamically along the execution of a program. Fig. 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 the area which is referable from applications directly. Specifically, such pointers are stored into the global variable area, call stack, or registers. These areas are called the **set of roots**. Among the objects located in the heap area, the objects which are referable from the set of roots, that is, which are referable from applications, are called **Live objects**. On the other hand, the objects which are not referable from the set of roots are called **Dead objects**, and GC frees the unnecessary part of the heap area occupied by these Dead objects.

Mark & Sweep[2] is one of the representative GC algorithms. This algorithm consists of two phases. One is called Mark phase, and the other is called Sweep phase. In Mark phase, objects are traced by following the pointers with the set of roots as the start points, and all Live objects are marked. Then, it goes on to Sweep phase. The whole heap area is scanned, and Dead objects which have not been marked at Mark phase are collected. Mark & Sweep alternates these two phases. Other representative GC algorithms are Copying[3] and Reference Counting[4].

Now, it is known that each of almost all GC algorithms which are being studied now is a combination or an improvement of these three representative algorithms. Especially, Mark & Sweep is widely used in combination with other algorithms because this algorithm is comparatively easy to be implemented. For example, Generational GC[5] is widely used in such as Java Virtual Machines. In Generational GC, the objects are divided into some generations, and the suitable GC algorithm is applied to each generation. In particular, Generational GC widely adopts a GC algorithm based on Mark & Sweep or Copying for each generation. Like this, many GC algorithms are based on Mark & Sweep or Copying. In this paper, therefore, we focus on the point that Mark & Sweep and Copying should trace objects to find Live objects. By reducing the processing time for the trace, we aim to improve the performance of many GC algorithms.

III. Pointer Detection in the Call Stack

In this section, we will explain the common routine in Mark & Sweep and Copying. These algorithms should find pointers in the set of roots to trace objects. Then, we will also explain an existing method for finding the pointers.

A. Tracing Pointers from the Set of Roots

Mark & Sweep and Copying which form the basis for many GC algorithms have common routine for finding a pointer in the set of roots and tracing the objects. The call stack is one of the roots. Each stack frame on the call stack contains space for the associated function. It includes the local variables, the return address, the arguments, and so on. Each local variable is either of reference type or primitive type, but the type is generally not distinguishable on the call stack. Therefore, it is needed to detect the pointers within each stack frame for finding a source of trace. In the next section, we will explain the technique for detecting the pointer adopted by Hotspot-VM[6] which is one of the Java Virtual Machines.

B. Stackmap

Stackmap, which is adopted by HotspotVM, is an existing technique for pointer detection. Each stack frame on the Java call stack contains an array of local variables and an operand stack. The operand stack is used for executing instructions by HotspotVM, because it is a stack machine. Stackmap is a bit string for indicating the locations of reference type variables in local variables or the operand stack. A Java source program is compiled into a bytecode. Then, Stackmap is generated by keeping track of each type of values in local variables or on the operand stack along the execution of the bytecode. Java bytecode has an unique store instruction for each type of value. Therefore, the type of a value in the local variable or on the operand stack can be identified based on which store instruction is used for the value.

For example, suppose that the program shown in Fig. 2 is executed. In this program, values are stored into two local variables: *val* and *ref*. This program is compiled into the bytecode shown in Fig. 3. In this bytecode, both 'istore_1' at the line 2 and 'astore_2' at the line 6 are the instructions for storing values into local variables. However, they take different types of values as their argument. Specifically, 'istore_1' is for storing an integer value and 'astore_2' is for a reference. These two instructions in Fig. 3 correspond to the assignments to *val* and *ref* in Fig. 2 respectively. Hence, it is found that *val* has an integer value and *ref* has a reference.

```
1 class Sampleprogram{
2   public static
3     void main(String args[ ]){
4       int val = 1;
5       Object ref = new Object();
6     }
7 }
```

Fig. 2. Sample Program

```
1 iconst_1
2 istore_1
3 new
4 dup
5 invokespecial
6 astore_2
7 return
```

Fig. 3. Bytecode

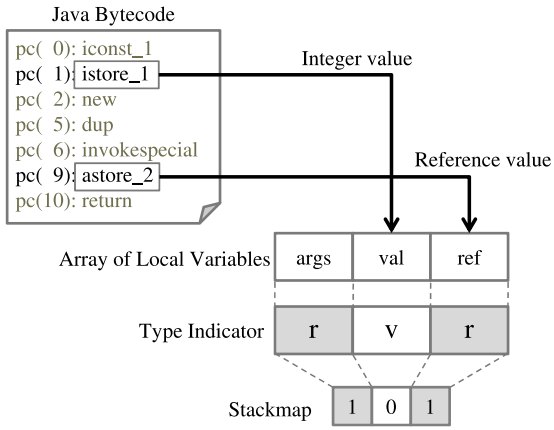

Fig. 4. Example of generating Stackmap.

Fig. 4 shows how Stackmap is generated along the execution of the bytecode shown in Fig. 3. In this figure, 'type indicator' means the type of each value. Specifically, *v* means the primitive type and *r* means the reference type. In this program, *args*, *val* and *ref* are local variables. As mentioned above, it is found that *val* has an integer value and *ref* has a reference. Similarly, it can be also determined that *args* has a reference based on which store instruction is used for the value. Then, each type of the value in the local variables is recorded, and the bit string '101', which indicates the locations of references within the stack frame, is generated. By referring this Stackmap, the GC routine can detect pointers in the call stack.

Incidentally, an execution flow path may vary with a conditional branch. Therefore, Stackmap has to be generated along each execution flow path. However, if many Stackmaps are generated and retained for handling all execution flow paths, they might deteriorate the memory usage efficiency. Thus, Stackmap needs to be generated every time GC runs, and this will be a large overhead of GC.

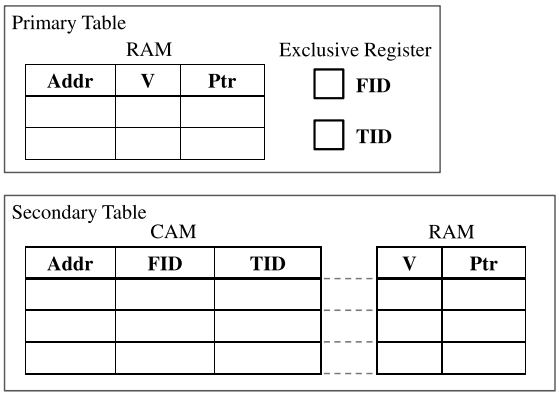

Fig. 5. Structure of Ptr-Table.

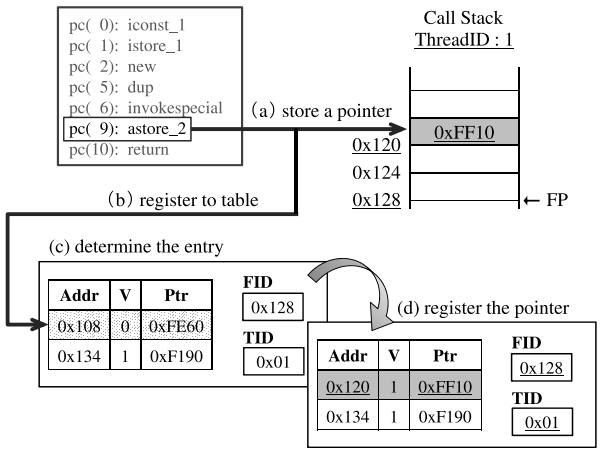

Fig. 6. Execution model of pointer registration.

IV. Hardware Support Technique

In this section, we propose a hardware support technique for reducing the overhead of pointer detection. In the proposed method, we install tables for managing all pointers on the call stack. We will explain the tables and the execution model of the proposed method.

A. Pointer Management with Tables

In the proposed method, tables for managing all pointers on the call stack are installed in the processor. In this paper, we call these tables **Ptr-Table**. Now, pointers on the operand stack will be updated frequently. Such frequent update may increase the overhead caused by the access latency to Ptr-Table. Therefore, for restraining the overhead, Ptr-Table is composed of two tables: **Primary Table** and **Secondary Table**. Primary Table manages only the pointers within the current frame. On the other hand, Secondary Table manages all pointers on the call stack except in the current frame. Since Primary Table manages only the pointers within the current frame, the size of Primary Table can be small and it can be accessed fast.

The structure of these tables is shown in Fig. 5. Each entry of these tables holds a pointer on the call stack and some associated values. Primary Table contains three fields. One is *Address* (Addr) which holds the address of the call stack where the pointer is located. When an entry in Primary Table needs to be updated, the entry can be specified by using this Addr as a key. Another is *Valid* (V) which holds a flag representing validity of the entry. The other is *Pointer* (Ptr) which holds the pointer stored in the call stack. Now, we assume that Primary Table is implemented with a RAM as same as L1 cache. In addition, two exclusive registers are installed which hold the *FrameID* (FID) and *ThreadID* (TID) respectively for identifying the current frame. Here, FID holds the address indicated by the frame pointer of each frame, and TID holds own thread identifier.

Secondary Table is roughly divided into two parts. One is implemented with a ternary CAM (Content Addressable Memory) which is capable of fast associative search. We call this part **CAM-Table**. The other is implemented with a RAM which is capable of sequential access. We call this part **RAM-Table**. These two parts of Secondary Table contain the same number of entries and each entry corresponds to the other table entry which has the same index. The fields *Address* (Addr), *FrameID* (FID), and *ThreadID* (TID) are in CAM-Table. These fields are used for identifying entries when the tables are operated. When a pointer is updated or deleted, the entry which corresponds to the pointer can be determined immediately by associative search. On the other hand, the fields *Valid* (V) and *Pointer* (Ptr) are in RAM-Table. The values in these fields are necessary for the GC routine. Using these values, the GC routine can determine the validity of the pointers registered in the tables and trace only the valid pointers. The GC routine should scan all entries in this table. Therefore, these fields are in RAM-Table so that fast sequential access will be available. When a certain entry needs to be updated, CAM-Table is searched for the entry by using Addr, FID, and TID as the keys. Then, the index of the entry is used for accessing to RAM-table. This enables RAM-Table to be accessed fast because each RAM-Table entry corresponds to the CAM-Table entry which has the same index.

B. Execution Model with Ptr-Table

To use Ptr-Table as mentioned before in GC, the tables should be operated along the execution of a program and should always hold all necessary pointers for tracing objects. In the following, we will explain the operation for Ptr-Table and the execution model of GC with the tables.

1) Operation for Ptr-Table: Ptr-Table should hold all pointers which are stored into local variables or the operand stack on the call stack. Therefore, when an instruction which stores a pointer into a local variable or the top of the operand stack is executed, the pointer should be registered into Primary Table.

Fig. 6 shows the execution model of pointer registration. In this example, (a) 'astore' instruction stores the pointer '0xFF10' into a local variable which is located at the address '0x120' on the call stack. In the proposed method, when the instruction for storing a pointer into the call stack, such as

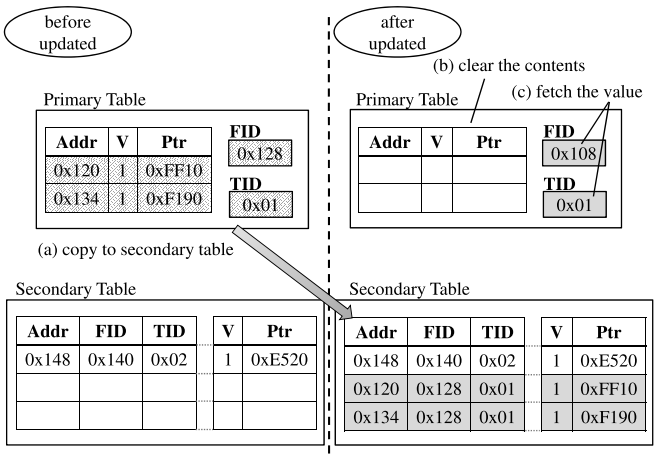

Fig. 7. Processing flow when invoking a method.

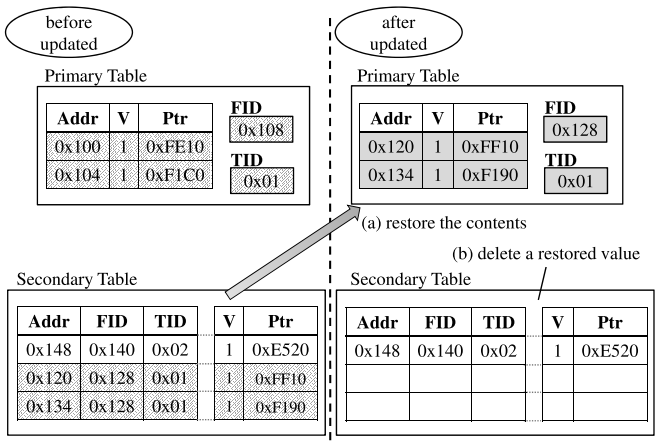

Fig. 8. Processing flow when returning from a method.

'astore', is executed, (b) the pointer is also registered into Primary Table. Before a pointer is registered, Primary Table should be searched by using the address where the pointer is located as a key to confirm whether the entry corresponding to the address is already registered or not. This is for preventing a situation where a pointer is managed by multiple entries. If the content of the table is as shown in Fig. 6 (c), the entry which corresponds to the local variable located at '0x120' has not registered yet. Then, Primary Table is searched for an entry which does not hold a valid pointer by scanning Valid field, and the entry is used for the pointer. Next, (d) the address of the local variable and the pointer are registered into the fields of the entry. In the example shown in Fig. 6, the address '0x120' which indicates the location of the pointer '0xFF10' on the call stack is registered into Addr field. Then, the pointer '0xFF10' is registered into Ptr field, and Valid is set. Meanwhile, although this example shows only how to register a pointer into Primary Table, some pointers in Primary Table should be updated or deleted in some cases. When a new pointer is stored into a local variable where another pointer has been already stored, the associated entry should be updated. In this case, Primary Table is searched for the entry by using Addr as a key. Then, its Ptr field is updated. On the other hand, when a NULL pointer is stored into an entry, the object which is referred by the old Ptr in the entry is considered to be deleted, and its Valid is unset.

For allowing Primary Table to manage only the pointers within the current frame, some operations are required when another frame becomes new current frame, for example, when a new stack frame is built. In the proposed method, when invoking a method or returning from a method, Primary Table is managed for adapting to frame changes.

Fig. 7 shows how Ptr-Table is operated. When a method is invoked, the contents of Primary Table should be saved and correspond to the new current frame. Therefore, when a method is invoked, (a) valid entries in Primary Table are copied to Secondary Table. Then, (b) after the contents of Primary Table is cleared, (c) new FID and TID are registered so that Primary Table corresponds to the new current frame.

Next, Fig. 8 shows how Primary Table is operated when returning from a method. When returning from a method, the contents of Primary Table should be cleared and restored from Secondary Table. Therefore, after all the contents of Primary Table are cleared, Secondary table is searched for entries corresponding to the new current frame by using FID and TID as the keys. Then, (a) the entries found in Secondary Table are restored to Primary Table. Meanwhile, (b) the entries which are restored to Primary Table are deleted from Secondary Table. If such entries remain in Secondary Table, they can cause inconsistency between Primary Table and Secondary Table, because updating or deleting a pointer in the current frame affects only to Primary Table. Therefore, the pointers in the current frame should be always managed only in Primary Table.

When a thread exits and the call stack is destructed, all pointers in the call stack should be deleted from Ptr-Table. In such cases, both Primary Table and Secondary Table are searched for the entries associated with the thread by using TID corresponding to the thread as a key, and the Valid fields of these entries are cleared. In this way, even an entry in Secondary Table may become invalid, although only valid entries are copied to Secondary Table when a method is invoked. Therefore, Secondary Table also needs the Valid field.

Meanwhile, the number of entries in Ptr-Table is finite. Therefore, if many pointers have to be stored, an entry overflow may occur in Ptr-Table. In such cases, we assume that VM turns off the hardware pointer detection and uses Stackmap again. If VM starts to use Stackmap again, however, Ptr-Table may lose some necessary pointers. Therefore, VM cannot adopt the proposed method again. Hence, it is important to determine the adequate size of Ptr-Table for preventing the frequent entry overflow. In addition, we will also research a more efficient technique for handling the entry overflow in the future.

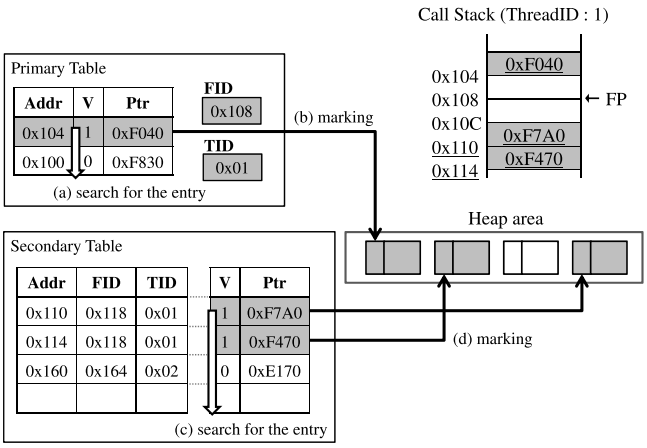

Fig. 9. Execution model of GC.

C. Execution Model of GC

The GC routine uses the pointers which have been registered in Ptr-Table as the root for tracing objects. This means that Ptr-Table is added to the set of roots. Then, when the GC routine traces objects, it refers Ptr-Table as one of the source for tracing pointer instead of the call stack. All pointers on the call stack are stored in Primary Table or Secondary Table. Therefore, it is possible to trace all the Live objects by referring to these tables. By referring these tables, the overhead for pointer detection in the call stack can be reduced.

Fig. 9 shows an execution model of GC with Ptr-Table. This example shows a situation where three pointers are stored in the call stack, and one of them is in the current frame. In addition, a thread whose TID is '0x02' is assumed to have exited and Secondary Table has some invalid entries. First, (a) Primary Table is searched for the entries whose Valid fields are set. In the case of the example shown in Fig. 9, only the entry which has the pointer located in '0x104' is found. Therefore, (b) the object which is referred by the pointer in the entry is marked. Similarly, (c) Secondary Table is also searched for the entries which hold valid pointers. Then, (d) the object which is referred by the pointer in each of the entries is marked. In this way, pointers in each frame on the call stack can be detected only by accessing to Ptr-Table. This can reduce the overhead for pointer detection and improve the performance of GC.

V. Performance Evaluation

We have evaluated the performance of the proposed method with a simulator. In this section, we discuss the effectiveness of the proposed method.

A. Simulation Environment

In this evaluation, we used gem5 simulator system[7]. The evaluation environment is shown in TABLE I. As the target architecture, we selected ARM which is widely used for embedded systems. ARMv7 is a 32-bit RISC microprocessor, and ARM-RealView PBX is a baseboard which is a highly integrated software and hardware development system based on ARMv7 architecture. In this environment, We evaluated the performance of HotspotVM 1.6.0. In addition, we prepared benchmark programs which tax the heap memory by invoking a method repeatedly for generating objects.

TABLE I
EVALUATION ENVIRONMENT

Machine	ARM-RealView PBX
Processor	ARMv7
Frequency	2.0 GHz
Memory	128MB
OS	Linux 2.6.38.8-gem5

Meanwhile, we assume that the ISA is extended and HotspotVM can update and read Ptr-Table with some dedicated instructions. We will implement the GC library for using the extended ISA easily in the future.

B. Evaluation Results

The proposed method uses Ptr-Table instead of Stackmap. Therefore, the overhead for generating Stackmap can be reduced. In HotspotVM, Stackmap is generated in the function compute_map(). Hence, to confirm the effectiveness of the proposed method, we evaluated the execution cycles of whole GC routine and compute_map() while the benchmark program is running.

Now, as mentioned in section III-B, Stackmap needs to be generated every time GC runs because the execution flow paths may vary. To reduce this overhead, HotspotVM caches some generated Stackmaps and reuses them. Therefore, it is expected that the effectiveness of the proposed method varies depending on the reusability of the Stackmap cache. To examine the effect caused by the cache reusability, we prepare four benchmark programs and compare the execution cycles of each program with other programs. Currently, the proposed model cannot execute complex programs such as real applications. Therefore, each benchmark program consists of a simple recursive function func() and some conditional branches. If the depth of the recursion and the number of conditional branches increase, more Stackmaps should be generated and the reusability of Stackmap cache will be deteriorated. The program (a) has no conditional branch. On the other hand, the program (b), (c), and (d) has some conditional branches. Each program invokes func() 10 times in main() function. Being invoked by main(), func() calls itself recursively. Program (a) and (b) repeat the recursive call 10 times. Now, program (b) has the conditional branch in func() so that it has 10 execution flow paths. Furthermore, we also evaluated program (c) and (d) for reference which are expected that the proposed method works more effectively. Program (c) repeats the recursive call 15 times and has 15 execution flow paths. On the other hand, program (d) repeats the recursive call 20 times and has 20 execution flow paths. Incidentally, the effectiveness of the proposed model depends on not only the reusability of the Stackmap cache but also the size of objects which are generated in the benchmark programs. This is because, the size of objects affects how many times GC should run. In this

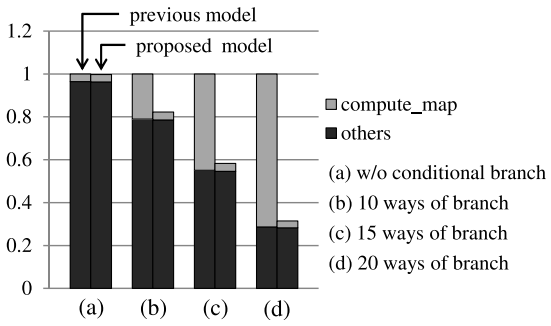

Fig. 10. Ratio of execution cycles.

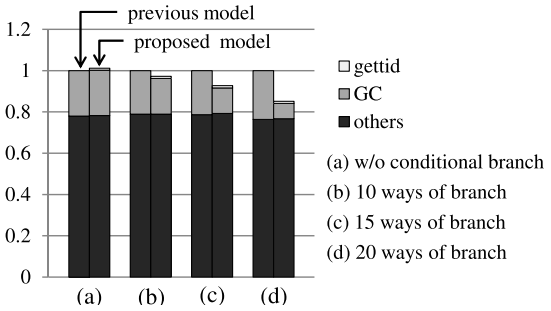

Fig. 11. Total cycles of each program.

evaluation, therefore, we adjusted the ratio of GC in the total execution cycles to about 20% in each program by adjusting the size of objects.

We have evaluated the previous model and the proposed model. Fig. 10 shows the total execution cycles of GC and the ratio of the compute_map() of each program. Each bar is normalized to the execution cycles of the previous model.

Note that as the number of the recursive call and the execution flow paths increases, the ratio of compute_map() in GC also increases and the total cycles of GC is much reduced. This is because, the reusability of the Stackmap cache is deteriorated along the change of execution flow paths by conditional branches. As a result, while the execution cycles of (a) is reduced by only 0.31%, each execution cycles of (b), (c) and (d) are reduced by 17.8%, 41.7% and 68.5% respectively. The result shows that the previous model which caches Stackmap cannot reduce the execution cycles of compute_map() when the execution flow paths change frequently. On the other hand, the proposed model can reduce the execution cycles of compute_map() regardless of control divergence. This means that the proposed model can have the good effect as well or better than the Stackmap cache for various programs. Therefore, the memory area for holding the Stackmap cache is no longer necessary, and the memory usage efficiency can increase.

Next, we evaluated the total cycles of each program. The result is shown in Fig. 11. Each bar is normalized to the execution cycles of the previous model. Here, *gettid* is a function for retrieving the thread identifier in the proposed model. The result shows that the total cycles of program (a) and (b) have no considerable difference between the previous model and the proposed model. One of the reasons for this is the small ratio of execution cycles of compute_map() to total cycles in these programs. In addition, currently, the proposed model uses *gettid* function as mentioned above and execute some instructions to operate Ptr-Table. Thus, the total execution cycles of the function and the instructions appear to be the overhead of the proposed model.

However, in the case of program (b), the execution cycles of GC is reduced by 17.8%, and this can lead to low maximum suspension time. Moreover, the total execution cycles of program (c) and (d) are reduced by 7.28% and 14.8% respectively. This means that the effectiveness of the proposed model depends on the characteristics of each program. However, we can evaluate only simple programs at present. Therefore, to confirm the performance of the proposed model in the environment which is similar to real applications, we will increase the number of executable programs and evaluate them in the future.

In the proposed model, the access latency for Ptr-Table should be considered while the execution cycles of GC is reduced. In this evaluation, we assume that the RAM used for Ptr-Table is similar to the RAM used for L1 cache. Thus, the access latency for the RAM is assumed to be two cycles which are equal to the L1 cache access latency in the simulator. In addition, the CAM used for Ptr-Table is modeled on DC18288[8] and its frequency is assumed to be 200MHz. The associative search in the CAM can return the result in one cycle. However, the clock frequency of the processor is 2GHz which is 10-times faster than the CAM. Thus, the access latency for the CAM is assumed to be 10 cycles by taking the ratio of these frequencies into account. Therefore, the total latencies for accessing one entry in Secondary Table is assumed to be 12 cycles which are consist of 10 cycles for accessing the CAM to determine the entry, and 2 cycles for accessing the RAM.

We approximate the overhead of the proposed model by multiplying the access count by the access latency. TABLE II shows the reduction rate of proposed model, the ratio of the overhead, and the total reduction rate in each program.

TABLE II
THE CYCLE REDUCTION RATE INCLUDING THE OVERHEAD(%)

	(a)	(b)	(c)	(d)
Proposed model	-1.22	2.75	7.28	14.83
Overhead	0.96	0.97	0.99	0.99
Total	-2.18	1.78	6.29	13.84

TABLE III
BREAKDOWN OF THE OVERHEAD(%)

Overheads	(a)	(b)	(c)	(d)
Registration	20.14	19.97	21.12	22.32
Copy(invoke)	70.76	71.19	70.74	69.95
Copy(return)	6.18	6.22	6.37	6.45
Reference	2.93	2.62	1.78	1.58
Total	100	100	100	100

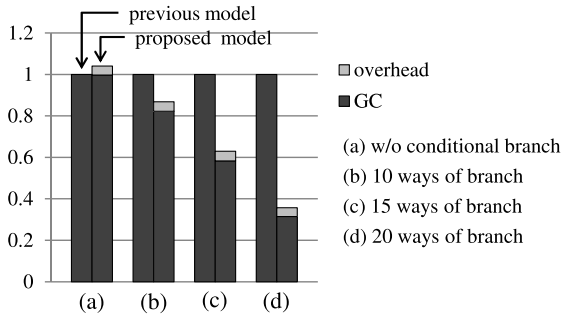

Fig. 12. Ratio of the overhead to GC.

The overhead is mainly divided into registration overhead, copy overhead, and reference overhead. TABLE III shows the breakdown of each overhead. 'Registration' is the overhead for registering a pointer into Primary Table. 'Copy(invoke)' is the overhead for copying valid entries from Primary Table to Secondary Table in invoking a method. 'Copy(return)' is the overhead for copying valid entries from Secondary Table to Primary Table in returning from a method. 'Reference' is the overhead for referring Ptr-Table in GC. As shown in TABLE II and TABLE III, we confirmed that the overhead is sufficiently small in each program.

In the proposed model, the execution cycles of GC in program (a), (b), (c), and (d) respectively account for 21.6%, 17.8%, 13.4%, and 8.7% of the total execution cycles. Now, the ratio of the overhead to GC in the proposed model is shown in Fig. 12. The ratio of the overhead in each program is respectively about 4.37%, 5.58%, 8.00%, and 13.3%. As a result, it is found that the more execution cycles of GC are reduced, the larger the ratio of the overhead is. However, we confirmed that the overhead is sufficiently small compared with the reduced cycles.

Meanwhile, the current proposed model cannot be applied to some routines, such as Java Classloader which loads Java classes before main() function is called. Therefore, the proposed model still uses Stackmap only for finding pointers which are stored onto call stack in such the routine. Thus, if all pointers on the call stack can be managed properly with Ptr-Table, the execution cycles of compute_map() function will be completely eliminated.

Incidentally, reducing processing time for adjusting a pointer can be also considered as another technique for improving GC. This pointer adjustment is required when an object is copied to other area. HotspotVM adopts Generational GC and uses Mark Compact algorithm which is an improvement of Mark & Sweep. In this algorithm, all Live objects are relocated in the top of the heap area for compaction after Mark phase. We evaluated the execution cycles of each program for adjusting the pointer in this compaction, and it is found that the routine can account for about 20% of GC. Therefore, it is expected that the routine can be processed fast by calculating and managing the adjusted pointer with some dedicated mechanism. We will also consider another technique for improvement of GC, including such fast pointer adjustment, from various aspects in the future.

VI. Conclusion

In this paper, we focused on a common GC routine for tracing pointers from the set of roots, in order to improve the performance of many GC algorithms. Then, we proposed a hardware support technique for speed-up of the trace. The call stack, which is one of the roots, contains not only pointers but also non-pointers such as integer values. Therefore, the GC routine should detect the pointers within the call stack. Hence, we installed tables, which manage all pointers on the call stack, in the processor. By using the tables, the GC routine can detect pointers on the call stack immediately. Finally, we evaluated the execution cycles of the proposed model and the overhead for managing the tables by using a simulator. As a result, it is found that the proposed model brings the low GC latency.

One of our future works is to reduce the hardware implementation cost and the access overhead for the tables. For example, the overhead for referring the tables by the GC routine can be reduced by efficiently partitioning the tables. Meanwhile, the proposed method may not be effective in some situation. For example, if the ratio of GC in the total execution cycles is small, the overhead for storing pointers will become larger than the reduced cycles in GC. Therefore, we should develop a dynamic switching method which can determine whether VM should adopt the proposed method or not. In addition, because some hardware units are installed in the proposed model, the energy consumption will increase. Therefore, after evaluating the energy consumption in detail, we will also examine a technique for suppressing it. Especially in mobile systems, the energy consumption is very important, and this will be a prior issue for our project.

Acknowledgment

This research was partially supported by JSPS KAKENHI Grant Number 25540019 and Inamori Foundation Research Grant.

References

[1] Y. Ossia *et al.*, "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.

[2] 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.

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

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

[5] M. Hirzel, "Data Layouts for Object-Oriented Programs," in *Proc. ACM SIGMETRICS Int'l Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'07)*, 2007, pp. 265–276.

[6] L. Bak, J. Duimovich, J. Fang, S. Meyer, and D. Ungar, "The New Crop of Java Virtual Machines," in *Proc. 13th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, 1998, pp. 179–182.

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

[8] MOSAID Technologies Inc., *Feature Sheet: MOSAID Class-IC DC18288*, 1st ed., Feb. 2003.