IPSJ Transactions on Programming Vol.5 No.3 29–39 (Aug. 2012)

Regular Paper

Proposal of GC Time Reduction Algorithm for Large Java Object Cache

Yasushi Miyata[1,a)] Motoki Obata[1] Tomoya Ohta[1] Hiroyasu Nishiyama[1]

Received: December 16, 2011, Accepted: April 6, 2012

Abstract: Server memory size is continuously growing. To accelerate business data processing of Java-based enterprise systems, the use of large memory is required. One example of such use is the object cache for accelerating read access of a large volume of data. However, Java incorporates a garbage collection (GC) mechanism for reclaiming unused objects. A typical GC algorithm requires finding references from old objects to young objects for identifying unused objects. This means that enlarging the heap memory increases the time for finding references. We propose a GC time reduction algorithm for large object cache systems, which eliminates the need for finding the references from a specific object cache region. This algorithm premises to treat cached objects as immutable objects that only allow READ and REMOVE operations. It also divides the object cache in two regions. The first is a closed region, which contains only immutable objects. The other is an unclosed region, which contains mutable objects that have survived GC. Filling an unclosed region changes the region to closed. When modifying an immutable object, the object is copied to the unclosed region and the modification is applied to the copied object. This restricts references from the object cache region to the region for young objects and excludes changes to the objects in the closed region. Experimental evaluation showed that the proposed algorithm can reduce GC time by 1/4 and improve throughput by 40% compared to traditional generational GC algorithms.

Keywords: Java, garbage collection, object cache, memory management, copy on write

1. Introduction

Recently, the amount of memory on enterprise server systems has been continuously increasing. Acceleration of business data processing by using a large amount of memory is required. Because memory access speed is dramatically higher compared with a disk drive, business processing can be accelerated using memory as a cache for frequently referenced data. Java [1] [*1] is commonly used for developing enterprise systems because of its productivities in rich libraries and securities of bytecode interpreters to execute Java programs. Therefore, the use of a large memory as a cache is effective for accelerating business data processing. There are two kinds of memory caches. One is an object cache [2], which is allocated as a part of the heap memory for Java applications. The other is a server style memory cache, which is managed in different processes from Java applications. If a Java application uses a server style memory cache. The business data processing flow is as follows [3]:

(1) The Java application sends a business data key to the memory cache.
(2) The memory cache sends the data corresponding to the key to the Java application.
(3) The Java application deserializes the received data to the internal object format.
(4) The Java application executes business data processing with objects received in step (3).

This business data processing flow sends/receives/converts the data between a Java application and a memory cache. These overheads result in performance degradation. In contrast, an object cache holds data as an object that can be referenced directly from a Java application. The Java application can invoke the access interface directly for operating data. This reduces the overhead of sending/receiving/converting the data to/from/into internal object formats. In common implementation of object caches for a language like Java, data are generally maintained on heap memory. In addition, object-oriented language usually adopts garbage collection (GC) as a means for an automatic memory management. Many GC implementations pause business data processing before collecting garbage objects. If it spends more time on GC processing, the performance of business data processing deteriorates. Generational GC [4], which is usually adopted as a GC algorithm, manages a Java heap memory in two regions. One is a new region for maintaining newly generated data. The other is an old region for maintaining the data that are live during a specified period. A garbage collector pauses a Java program and collects dead objects. A generational garbage collector effectively collects garbage by frequently collecting dead objects in the new region because a new region is expected to contain a relatively large portion of garbage objects. However, finding references from all data on the heap memory to the new region is necessary to find garbage data. If the heap memory contains many objects, the GC time increases by increasing the time for finding references.

[1] Yokohama Research Laboratory, Hitachi, Ltd., Yokohama, Kanagawa 244–0817, Japan
[a)] yasushi.miyata.bz@hitachi.com
[*1] Java is a trademark or registered trademark of Oracle, Inc. in the US and other countries.

Therefore, to accelerate a Java program that uses a large memory, reduction in the GC time for the heap memory for a large object cache is needed. We reveal the reason for increased GC time and propose an algorithm for reducing the time for generational GC with a large object cache. The rest of this paper is organized as follows. Section 2 presents acceleration methods for data processing with an object cache and reveals the reason of increasing GC time depending on the enlargement of the heap memory for an object cache. Section 3 presents the GC time reduction algorithm that incorporates novel data management method for cached data. Section 4 shows the evaluation results regarding the GC time of a Java virtual machine (VM) with our proposed GC algorithm. Section 5 describes related works, and Section 6 concludes the paper.

2. GC Time for Object Cache

In this section, we present accelerating algorithms for data processing with a large Java object cache and describe the reason of increasing GC time with the large object cache.

2.1 Accessing Data in Object Cache

An object cache enables cached data to be maintained as an object format. A Key-Value Store (KVS)-like interface is commonly used for accessing data. A KVS interface consists of basic data access methods; "PUT" for storing, "GET" for drawing, and "REMOVE" for deleting objects. An application applies operations on data obtained by the KVS interface: READ for reading or WRITE for updating its contents.

A Java application can execute READ/WRITE operations directly on data with no copying or moving of data because the data are maintained in a Java heap memory. Consistency of the data must be maintained for WRITE operations when the same data is accessed by multiple activities as in multi-threaded environments. One method for achieving this is Java threads obtaining a lock for accessing the data and preventing other Java threads from executing the WRITE operation. By using simple locking operations, other Java threads cannot execute the WRITE operation to the locked data and that performs READ operations on the object must wait for the release of the lock. This degrades performance of the data processing speed. On the other hand, the Copy on Read and Committed (CORC) method [5] does not lock targets but copies data to the local area for each Java thread and performs READ/WRITE operations without waiting for the lock to be released.

When a Java thread obtains data with the CORC method, it copies them to the local memory area. Then, the Java thread executes the WRITE operation on the replicated data. When a Java thread puts the replicated data back into the cache, it copies the data and replaces them with those cached on an object cache. A Java thread can ensure consistency with other threads because each Java thread writes the data in their local memory area. However, the CORC method must copy data even where the Java thread executes only the READ operation on data. The CORC method cannot take advantage of executing READ operations directly on data in an object cache. In addition, the replicated data fills the Java heap memory and triggers GC because the replicated data usually become unnecessary after READ operation. There are more READ operations than WRITE operations in common business data processing. Therefore, acceleration of data processing to execute direct READ operations on an object cache is more important than that of WRITE operations. In other words, executing READ operations directly and WRITE operations on replicated data is appropriate for data consistency. The Copy on Write (COW) method [5] fills these requirements.

In the COW method, each Java thread obtains data and executes READ operations directly without copying. If a Java thread executes a WRITE operation, the data are copied before the WRITE operation, and the WRITE operation is executed on replicated data. When a Java thread puts the data into an object cache, the data in the object cache are replaced with those cached on the object cache just like with the CORC method. Therefore, the COW method achieves consistency with direct READ operations. Because of this, data processing with the COW method can be accelerated without the overhead of moving or coping data.

2.2 Increase in GC Time According to Heap Memory Enlargement

A generational GC algorithm finds the existence of references from data in an old region to data in collected region. Because this GC algorithm needs to follow references between data, finding references from data in an object cache is also necessary.

Finding references from all data in an old region increases GC time. Card-Marking [6] with the write barrier can reduce regions necessary for finding references. Card-Marking divides a Java heap memory into several fixed-sized regions, as shown in **Fig. 1**. Each region maintains information about references from their regions to the garbage collected region (GC Target). Each region is called a "Card". The Card holds "Card Info," which represents the existence of references from data in the "Card" to the garbage collected region. This Card Info is updated when the write barrier detects the update of a reference in the Card.

The garbage collector finds a Card that has references to the garbage collected region by using the Card Info. Only data in the Card are needed to be checked to find references. This finding method reduces GC time because it must check fewer regions. However, the garbage collector must check all Card Info to detect the existence of references from all data to the garbage collected region. Thus, the time for detecting all Card Info should increase with increased amount of memory. To confirm this increase, we examined the time reduction effect of Card-Marking with a large Java heap memory.

We conducted an experimental study of the generational GC

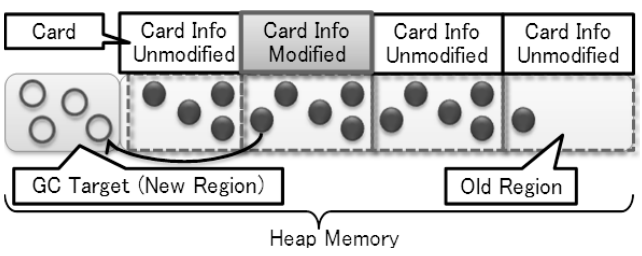

Fig. 1 Partitioning of data region for Generational GC.

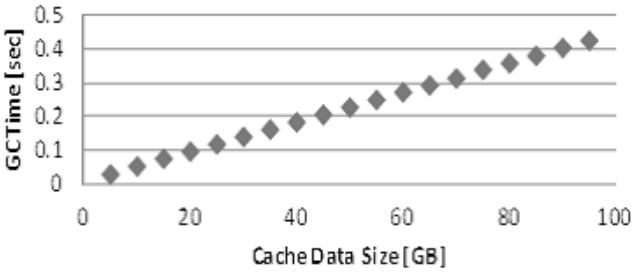

Fig. 2 GC time according to heap memory size.

```
1: class ProxyObject {
2:     Object _data;
3:     Object getObject() { return ( _data.getObject() ); }
4:     Object setObject( Object val ) {
5:         _data = isCopy() ? _data : _data.clone();
6:         _data.setObject( val );
7:     }
8: }
```

Fig. 3 Definition of proxy object.

time for collecting garbage periodically by finding data references. **Figure 2** shows that the GC time for a collected region increases with an increasing amount of data in an old region. This result was obtained as follows. We first filled an old region with dummy data. We then filled a collected region with garbage data, which was not referenced from an old region. Then, the total GC time was obtained for the GC process. There was no update of the data in the old region and no moving of data from the collected region to the old one. The GC time for detecting the Card Info and collecting the garbage in the collected region is shown in Fig. 2.

Figure 2 also shows that GC time increased in almost direct proportion to the amount of data in the old region. As a result, it took about 0.02 seconds in 4 GB region and about 0.4 seconds in the 100 GB region. In addition, the card size was set to 512 bytes in this Java VM (Ver. JDK 1.6). Therefore, the garbage collector had to find 8 million cards for the 4 GB memory region or 200 million cards for the 100 GB memory region. In contrast, the time for collecting garbage data in the collected region remained the same because the memory size of that region was the same. Therefore, it is expected that the increase in the number of cards that must be checked to find references is the central reason for the increase in GC time.

3. Reducing GC Time of Object Cache

In this section, we first discuss the implementation of the COW method described in Section 2. Next, we define the access interface of the object cache. Then, we propose GC time reduction method that contains memory management method for a Java object cache for our GC algorithm and describe the implementation. Finally, we summarize our GC time reduction algorithm with a Java object cache.

3.1 Accelerating Data Processing with Copy on Write

The COW method is effective for consistently accelerating direct access to data in an object cache. It must detect the WRITE operation for the data in an object cache. There are two methods for detecting WRITE operations: proxy and write-protect. The proxy method adds the logic for detecting READ/WRITE operations to the data access process, and the write-protect method protects memory access from the WRITE operations and detects if an exception comes up in the attempted WRITE access.

The proxy method detects the access type of READ/WRITE operations by accessing data through a proxy object. **Figure 3** shows an example of a proxy object. The proxy object has a reference to concrete data and provides the same data access interfaces as the referenced object. This proxy object delegates operations applied to concrete data after accepting an operation to referenced data. In the READ operation, a proxy object simply delegates the READ operation to concrete data. In the WRITE operation, a proxy object copies concrete data to the local area of the thread and delegates the WRITE operation against the replicated data. Since the decision needed before the actual operation is simple, performance degradation is expected to be small because optimizations, such as in-line expansion, are assumed. However, the amount of used memory increases because this proxy method requires proxy objects against concrete data.

The write-protect method sets the memory region for cached data to read only and the exception handler is executed in the WRITE operation. This exception identifies the concrete data for the WRITE operation and copies the concrete data to the destination. After copying, the write-protect method executes a WRITE operation against the replicated data. This method executes the exception handler. Therefore, performance degradation increases compared to the proxy method. In contrast, the amount of memory usage does not increase because there is no need for extra data such as proxy objects.

Reducing the overhead caused by interruptions in processing is difficult. However, the proxy object for the same data structure, the instance generated from the same class, can be reused. Therefore, memory usage should be minimized. As a result, we used a proxy method that can achieve high-speed COW with little increase in memory usage.

3.2 Data Access Interface for Java Object Cache

We assume that a Java object cache provides three basic interfaces:

- `Put` : registers data to an object cache with a key that identifies the data.
- `Get` : retrieves data that correspond to a specified key from an object cache.
- `Remove` : deletes data that corresponded to a specified key from an object cache.

Each operation is performed on Java objects. Java objects are usually composed of nested references. In this section, we first describe data processing without nested references then describe operations on data with nested references.

(1) Data Processing without Nested References

We assume data in an object cache are only modified by PUT/GET/REMOVE interfaces. In other words, we ensure that once an application puts data into an object cache, those data do

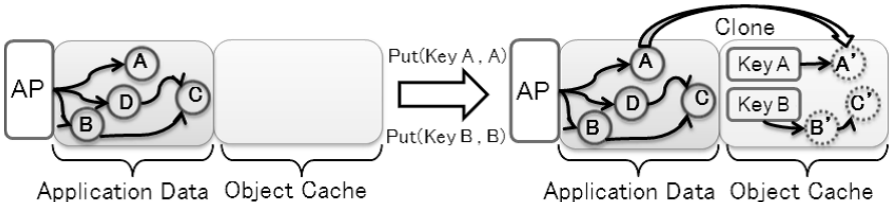

Fig. 4 Put data in the object cache.

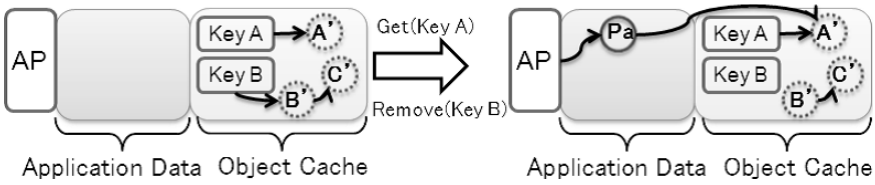

Fig. 5 Get and remove of the data in the object cache.

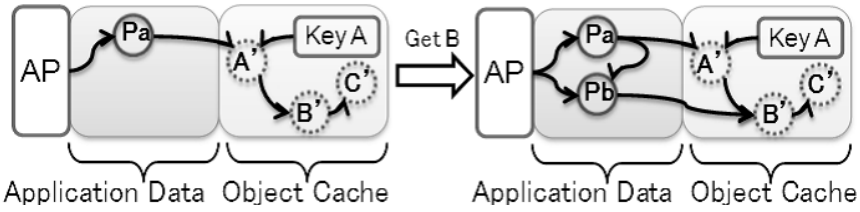

Fig. 6 Get and write of the multistage referenced data in the object cache.

not change if other interfaces have attempted to change them. To achieve this, when an application puts data in an object cache, the object is cloned and the cloned object is put in an object cache. For example, when an application puts data A, shown in **Fig. 4**, an object cache clones data A to A' and puts A' in an object cache with the key information. This ensures that the original data A can be directly accessed and the cloned data A' is not modified. Thus, cached data are not changed by applications.

An object cache has an advantage when there is less overhead of READ operations because the object cache enables applications to perform direct READ operations to data in an object cache. To attain this advantage and confine data processing to PUT/GET/REMOVE, we adopted COW using the proxy method. For example, when an application obtains data A' with the key, as shown in **Fig. 5**, the application obtains the proxy object Pa and accesses data A' through the proxy object Pa. When modifying data A', a WRITE operation to data A' is detected and a clone of data A' is created. When removing data B, a REMOVE operation is executed by specifying the key B and eliminating the reference to data B. These protocols can prevent the cached data from operations except PUT/GET/REMOVE.

(2) Accessing Data with Nested References

Operations on cached data with nested references must be considered. When a PUT operation registers data with nested references, the registered object is treated as a root object. Objects that are reachable from the root object are cloned and registered in the cache. For example, when a PUT operation registers data B shown in Fig. 4, the object cache generates clones of data B and data C referenced by data B. After that, the object cache puts the cloned data to the object cache region with the key information. This operation enables an application to obtain data and refers all data referenced by the source of the cached data.

We accomplished high-performance data access and consistency at the same time by applying COW to data with nested references. To achieve this, a proxy object's operations are defined to obtain the proxy object of referenced data when an application obtains the referenced data from the source data, as shown line 3 of Fig. 3. For example, when an application refers data A' through proxy object Pa, which has nested references, as shown in **Fig. 6**. The application must obtain the reference from data A' to data B' for accessing data B'. When proxy object Pa is accessed to obtain the reference to data B', proxy object Pb is returned to the application. This application accesses data B' through proxy object Pb. When the application executes WRITE operations to data B', modification to object B' is detected via proxy object B' and the clone of data B' (B" is created). These operations enable efficient read access without cloning, even if cached data have nested references.

3.3 Region Management for Java Object Cache

As described in the previous section, data are only added by a PUT operation into an object cache. In Section 2.2, we revealed that GC time increases with an increasing number of the regions to be detected if they are modified. The object cache defined in this paper only allows the PUT operation for modifying or adding data. Therefore, if we can restrict regions of the object cache for which data are put in specific regions, GC can also restrict the regions to find references. As a result, reduction in GC time can be expected by restricting the regions for finding references.

To guarantee these region restrictions, we need to separately manage the regions; one for putting cached data and another for placing other data. However, the restriction for available resources prohibits putting infinite data into an object cache. When a certain amount of data is added to a specific cache region, the re-

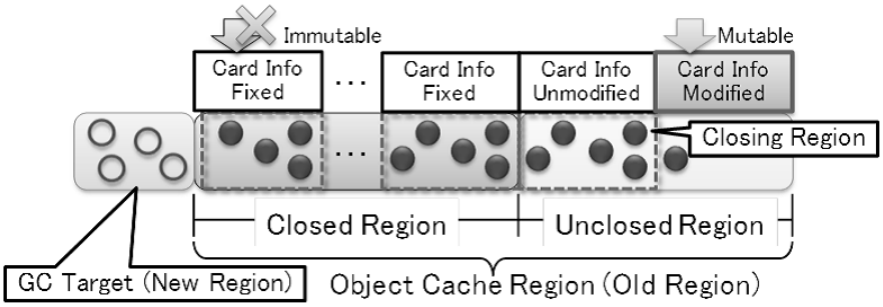

Fig. 7 Closed region and unclosed region.

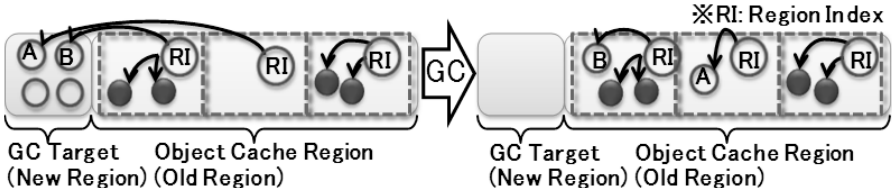

Fig. 8 Closed region and unclosed region.

gion management system restricts the region for adding data then adds a region for adding data. In other words, the object cache expands regions in accordance with the increase in the amount of data. Parts of the regions become puttable and others become unputtable. The data management method for immutable regions of our algorithm enables the expansion of cached regions and the management of added regions.

This data management method for immutable regions divides the object cache into sub-regions, as shown by the dotted line in **Fig. 7**. A sub-region is classified into the following two types; "closed regions," which do not allow putting nor updating data, and "unclosed regions," which allow adding and updating data. Because no data adding or updating of data on a closed region is allowed, we can simply determine that the Card of all closed regions is unchanged without finding data references in each Card. This means that we can mostly eliminate the need for detecting Card information. In addition, we can concentrate updated Cards in unclosed regions because no update is performed in closed regions.

To allow any kind of data to be cached, we must consider varying data size. If we use a fixed sub-region size, some large data will not fit in the sub-region. To prevent such mismatching, we made the cache region variable and limited the maximum size of the region by limiting the maximum number of objects that can be cached. When a sub-region exceeds the limit of registered data size, the state of the sub-region is changed from "unclosed" to "closed." Then, a new sub-region for registering cached data is added.

We made the Java VM to receive information about unclosed or closed regions of the object cache. With this information, the Java VM can determine closed regions as unchanged regions without any checks to find references.

3.4 Application to Generational GC

In this section, we describe the implementation of the data management method for immutable region of our generational GC algorithm. This management method divides the object cache into closed or unclosed regions. To apply this method, we must solve two problems. The first one is the putting of data. In generational GC, data are first placed in a new region as a young object. When the data are put in an object cache, the cloned data are not guaranteed to be placed in cache regions. The second one is the reclaiming of unused cache region. This problem is not specific to generational GC. However, because we cannot hold an infinite amount of data in an object cache, unused regions must be reclaimed. In the rest of this section, we describe the solutions for these implementation problems.

(1) Putting Data into Cache Region

The GC target region in Fig. 7 is the region in which an application generates temporary data for data processing. This region is the same as the new region of generational GC. All data, including cached data, are first placed in this GC target region. After an application puts the data in an object cache, clones of the data are registered. To guarantee the region in which the cloned object is placed as an unclosed region, we need to find all data by traversing references and move the found data that do not reside in the cache region to the unclosed region. This process sometimes leads to a large overhead in PUT operations due to reference traversal and copying. To hide this overhead, we overlay this copying process on GC that executes similar reference traversal and data movement.

When the garbage collector finds references, it begins traversal from a set of nodes. These nodes are called the "ROOT" set. The ROOT set is the data directly accessible from an application. If data cannot be traversed from the ROOT set, they can be treated as garbage. This object cache separately places the ROOT set on each sub-region of the object cache. The garbage collector moves the data traversed from the ROOT set to the sub-region to which the ROOT set is associated. We call the ROOT set associated with each sub-region as the Region Index (RI). The RI holds keys for managing data and the references to the data.

In **Fig. 8**, for example, the garbage collector moves data A and B to the corresponding region in which the RI is associated. If data have nested references, the garbage collector finds all data

reachable from the RI by recursive traversal and moves the found data to the region in which the RI is associated. If RIs only hold information about mapping of keys and data, an application must search multiple RIs when obtaining data. This results in performance loss. To solve this problem, we provide a Master Index (MI), which provides a KVS interface to manage mapping. When putting data, an application puts the mapping in both the RI and MI. When obtaining data, an application obtains data with the reference information of the MI.

The garbage collector moves data to unclosed regions. Therefore, the reference from the unclosed regions to a new region remains, even if the RI of an unclosed region has enough references to data for the unclosed regions. This prevents the unclosed region from changing to a closed region, which requires no reference detection to a new region. The garbage collector must guarantee no existence of references from the unclosed region to a new region. In other words, we need to introduce a new intermediate state that represents the transient state from that of an inhibiting PUT operation for filled unclosed regions that exceed the data registration limit to the state of a closed region. We define the intermediate state as a "closing region." If the garbage collector moves all data referenced from closing regions to an object cache, it changes the state to that of closed regions.

Many generational GC implementations move data that are not recognized as dead several times from a new region to an old one. These implementations do not guarantee that all data referenced from closing regions are moved in one GC process. Therefore, we make the garbage collector monitor the condition of data movement and to keep the state of closing regions if all the data referenced from the closing regions are not moved to the object cache region. In addition, a threshold for the number of times to find data movement is periodical. If the number of GC operations on a closing region exceeds the threshold, all the data referenced from the closing region are guaranteed to be moved to the object cache region. We can change the state of the region from closing to closed after executing GC operation as more times over than the threshold.

(2) Deleting Unnecessary Cache Regions

The object cache for the proposed GC algorithm must change the MI and RIs when deleting or updating data. When deleting data, the object cache deletes the reference from the closed region to the data. When updating data, the object cache deletes the reference from the closed region to the data and adds the reference to updated data in the RI of the unclosed region. In this manner, the data management on closed regions is limited only to removal of references and does not create a new reference to the GC target region. However, by deleting or updating, unused data are generated that are not reachable from the ROOT set. Therefore, after updating these processes, the garbage in closed regions increases and the efficiency of memory usage decreases. To solve this problem, the garbage collector reclaims the closed region. The region manager finds the references from the RI of each region and obtains the ratio of surviving data, which represents the ratio of the number of unmodified data over the maximum number of data in a region. The region manager determines a threshold of the survivor ratio. If the number of surviving data is under the threshold, the region manager adds a new unclosed region and relocates the surviving data to the new region. After the relocation of the surviving data, the region manager frees the closed region.

3.5 Object Cache Management System

In this section, we first describe the object cache with COW-type data access and immutable data region management for reducing GC time.

We call this object cache Explicit Object Cache (EOCache). **Figure 9** shows the organization of the EOCache. The Java object cache management system works in cooperation with object cache managers and the Java VM. When the user application written as a Java program sends PUT/GET/REMOVE, the object cache manager executes the following processes:

- `Put` : The index selector obtains the unclosed region that is the current target of the PUT operation and adds the reference to the PUT data. After that, the mapping information is registered in the MI.
- `Get` : The object cache manager obtains the data requested by the GET operation. The proxy selector obtains the proxy object for accessing data with the COW method. After that, the object cache manager sends the reference information of the proxy object to the user application. In this manner, the object cache manager permits the user application to access the data only through the proxy object. This access method can prevent the data from changing due to other activities. To reduce the overhead for generating the proxy object, the proxy selector generates the proxy object in the temporal data area before proxy object creation is requested, and reference information is only changed for the retrieved

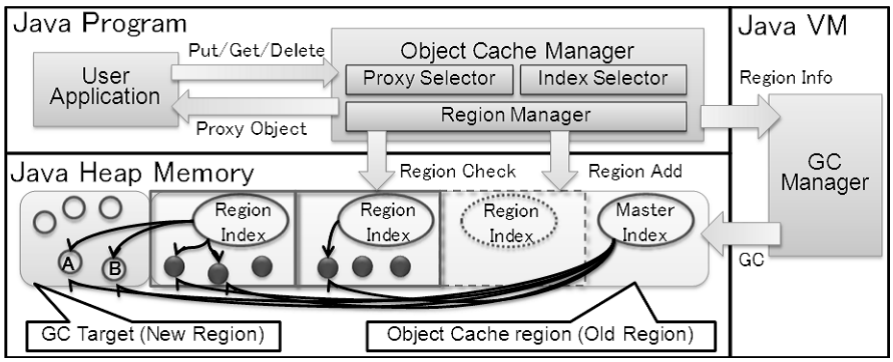

Fig. 9 Java object cache management system.

data on demand.

- `Remove` : The index selector obtains the RI, which has the reference to remove targets, by finding the reference for the RI from the MI. After that, it removes the references in the MI and RI.

The object cache manager manages the object cache region divided into unclosed and closed regions by the region manager. It sends the region information to the GC manager in the Java VM. The GC manager executes GC using received region information. First, the garbage collector finds the references for the Java heap memory except the object cache region and moves the data if necessary. Next, the garbage collector finds the references from unclosed regions and moves the data to the unclosed region if references to the data exist. If all the data referenced from a closing region are moved to the unclosed region, the GC manager takes the closing region away from a group of unclosed regions and changes it to a closed region. These processes enable the GC process to skip finding the references from closed regions.

4. Evaluation

The data management method for immutable regions of our GC algorithm is aimed at reducing GC time of a large Java object cache. This method reduces GC time that increases in accordance with the size of an object cache. In this section, we show the results of evaluating its effectiveness.

4.1 GC Time

Table 1 lists the evaluation environment conditions. We used generational GC as the GC algorithm and divided the Java heap memory in a new region and an old region. Their ratio of memory size was 1 : 2. GC is performed on the new region that contains 1.3 GB of data when obtaining GC time resulted in data in a new region becoming garbage. The total time spent for finding references and freeing a new region was obtained by this measurement. **Figure 10** shows GC time of previously proposed and our methods under these settings. The previously proposed method, labeled as "JDK1.6 HashMap," manages data using JDK 1.6 HashMap that uses the same data structure with that of our data management method. The method, labeled as "EOCache," manages data using the object cache for the proposed GC algorithm shown in Fig. 7. The vertical axis represents the amount of cached data and the horizontal axis represents GC time.

The results of our data management method shows that the increase in GC time of a new region is kept small compared to the increase in the size of the object cache. For example, when 96 GB of data is cached, JDK1.6 HashMap spent 0.43 seconds for GC, while our method reduced the time to 0.092 seconds. However, we also see that the GC time of our method also increased with the increase in the size of the object cache. The reason is that the MI has the references to all data in the object cache, and maintenance of the references must be executed if an application updates or deletes data. This means that we must find a modified part of the MI in GC. The MI only holds keys and references to data. The data size is small compared to that of cached data. Thus, finding a modified region in GC requires finding only a narrow region. In this evaluation, all data in the GC target were unused data not referenced from others. In real-world enterprise applications, GC time becomes much longer because data reachable from the ROOT set must be moved.

We also evaluated the GC time when updating data in the object cache. The test application randomly retrieved data from the 100 GB object cache and updated the retrieved data. Then, the application created a large amount of garbage data to invoke GC. **Figure 11** shows the obtained GC time. The horizontal axis represents the amount of updated data in the object cache. The vertical axis represents GC time. Our data management method (EOCache) reduced GC time by 1/4 compared with that of JDK1.6 HashMap. In this evaluation, the garbage collector found all data references in the updated Card in addition to usual reference finding and collection of garbage data. With JDK1.6 HashMap, updated data spreads in the heap memory if data are randomly updated. This increases data that require references finding and results in increased GC time. In contrast, our method gathers all updated data to unclosed regions. Thus, updates are directed to a small number of Cards, and the amount of data that requires references found is also small.

4.2 Performance of Data Processing

We evaluated data processing performance with the data management method for the proposed GC algorithm from the following points of view:

(1) Read/Write Performance of Cached Data
This evaluates the overhead of our data management method for reading and writing data.

(2) Number of Transactions Executed between GCs
This measures the number of transactions executed before a

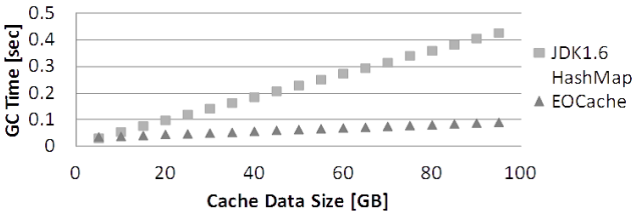

Fig. 10 GC time according to object cache usage.

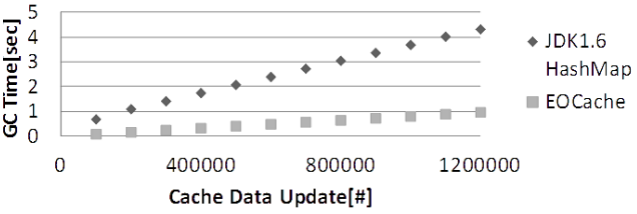

Fig. 11 GC time when updating 100 GB object cache.

Table 1 Evaluation environment.

CPU	Memory	OS	Java VM	GC Algorithm	Java Heap	Object Cache
2.3 GHz × 16Core	112 GB	Linux 64bit	Hotspot VM [7] JDK1.6 update 13	Generational GC	4 GB	100 GB

fixed amount of memory has been consumed by data processing until invoking GC. This confirms if our method can reduce the amount of memory usage for data processing.

(3) Throughput

This evaluates the throughput of applications for confirming that our method can reduce the GC time contained in all Java processing times and increase the amount of processed data in a fixed time window.

(1) Read/Write Performance for Cached Data

Read performance is the processing time required for obtaining data from a 100 GB object cache and executing the READ operation to the data. Write performance is the processing time required for executing the WRITE operation to data and putting the clone data into the object cache. This evaluation confirmed the read/write performance of our method with three types of data access methods: First the "EOCache + COW" method for the proposed GC algorithm that accesses data with COW, the "JDK HashMap + COW" method that maintains data with HashMap of JDK1.6, and the "JDK1.6 HashMap Direct R/W" method that directly executes READ/WRITE operations on data for comparison without guaranteeing data consistency. **Figure 12** shows READ/WRITE performance with 100 GB data in the EOCache or HashMap. With COW, the READ performance of our method was lower compared with the both previous methods. However, WRITE performance degraded by 8.8%. This overhead might have been caused by the overhead for accessing both the RI and MI on write access. The READ performance of our method degraded by 7.9% and WRITE performance degraded by 33% compared with Direct R/W. The decrease in READ performance may have been caused by the overhead for copying data before the WRITE operation. In common business data processing, READ operations are executed more frequently than WRITE operations. In addition, operations executed in data processing are not restricted only to READ/WRITE operations on cached data. Therefore, the WRITE ratio of all business data processing is small, and this leads to a small decrease in performance of business data processing compared with this evaluation.

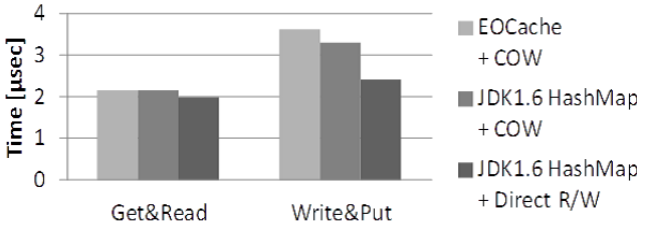

Fig. 12 Get & read performance/Write & put performance.

(2) Number of Transaction between GCs

Transaction is a series of data operations consisting of READ (GET & READ), WRITE (WRITE & PUT), and other operations such as generating a key or data to execute READ/WRITE operations. This transaction generates data and consumes Java heap memory. In generational GC, the transaction uses the memory allocated from a new region and triggers GC if the available memory of the new region is exhausted. We measured the number of transactions executed from the point of not consuming a new region to that of starting the GC process when the new region has been consumed. The experimental environment and evaluation results are discussed below.

We assumed the transaction in this experiment was to handle two kinds of data: cached data for READ/WRITE operations and temporal data necessary for the READ/WRITE operations. Both data sizes were about 220 Bytes. With the COW method, READ operations did not copy cached data and WRITE operation copied cached data. This means that the memory usage changed in accordance with the change in the ratio of READ operations to WRITE operations in the transaction. To investigate this effect, we had the transaction consist of 10 READ/WRITE operations and varied the number of WRITE operations executed in the transactions from 1 to 9. Therefore, the application handled 2.2 KB of cached data and generated 2.2 KB of temporal data. The cached data was set to 100 GB, which is the same as that for the evaluation of READ/WRITE performance, and the Java heap memory size for data processing was set to 4 GB. The GC algorithm used for evaluation was the generational GC. A new region in a size of 1.3 GB was treated as the GC target. **Figure 13** shows the results of the simulation estimating the number of transactions executed until 1.3 GB of memory allocated for a new region was consumed.

The "EOCache + COW" method for the proposed GC algorithm does not require copying of cached data in the READ operation. In contrast, it requires copying of cached data in the WRITE operation. This means that proposed data management method can execute 0.6 million transactions, which does not depend on the WRITE ratio. This number is obtained by dividing the heap size (1.3 GB) by the temporal data size (2.2 KB). Direct R/W can also execute 0.6 million transactions, which does not depend on the WRITE ratio because no extra data are generated in READ/WRITE operations except temporal data. The "JDK1.6

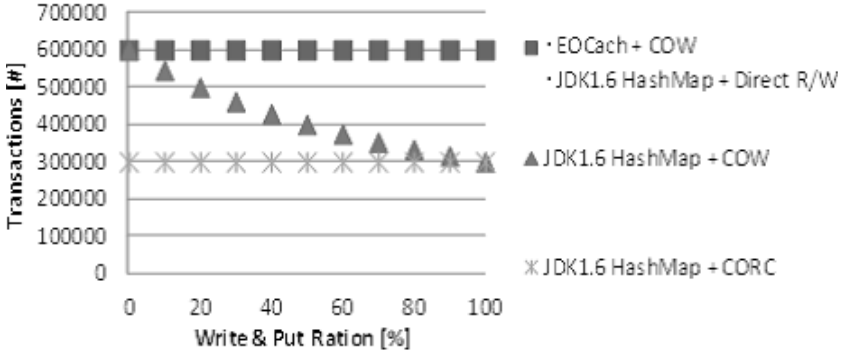

Fig. 13 The number of processed transactions until GC.

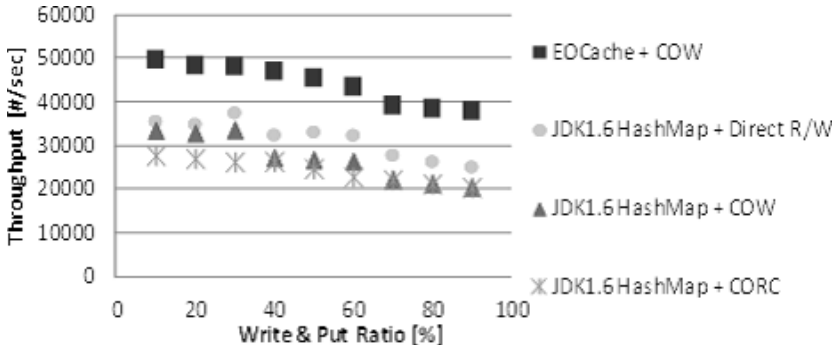

Fig. 14 Transaction throughput according to the ratio: Write & Put.

HashMap + COW" does not require copying of cached data in the READ operation. However, it requires copying of cached data to a new region in the WRITE operation. Therefore, it can execute 0.3–0.6 million transactions. This result is obtained by dividing the heap size (1.3 GB) by the sum of temporal data size (2.2 KB) and copied data size, which varies from 0 to 2.2 KB depending on the WRITE ratio. For comparison, we simulated the number of transactions with CORC (JDK1.6 HashMap + CORC in Fig. 13), which that also copies cached data in the READ operation. CORC requires copying cached data in both READ and WRITE operations. Therefore, it can execute 0.3 million transactions, which does not depend on the WRITE ratio. This is obtained by dividing the heap size (1.3 GB) by the sum of temporal data size (2.2 KB) and copied data size (2.2 KB).

(3) Throughput

Throughput is the number of transactions executed in a unit time window. **Figure 14** shows the results of throughput evaluation in the same environment as (2). The horizontal axis represents the WRITE operation ratio over all operations. The vertical axis represents the number of transactions per second. This figure also shows four types of throughputs: the COW method (EOCache + COW); the previously proposed method (JDK1.6 HashMap + COW); CORC with copying in both READ and WRITE operations (JDK1.6 HashMap + CORC); and Direct READ/WRITE with direct READ/WRITE operation (JDK1.6 HashMap + Direct R/W). The throughput of our method showed 40% performance increase over the other methods. The main reason for this may have been the reduction in GC time caused by narrowing the garbage collected regions. Regarding narrowing, our method handles 100 GB of cached data in the closed region, and 1 GB of cache region used as the unclosed region to place copied data. In addition to the 1 GB GC target, 4 GB Java heap memory for data processing was also the GC target. Therefore, only 5 GB was the GC target. In contrast, all the other methods (JDK1.6 HashMap + COW, CORC, Direct R/W) require 105 GB regions, which contain additional 100 GB for the object cache region. This indicates that our method can reduce the GC target size by about 1/20. In addition, GC with our method requires 5% of all processing time. All the other methods (JDK1.6 HashMap + COW, CORC, Direct R/W) require 20%. This means that our method can reduce GC time by 1/4. This result is the same with that shown in Section 4.1. The reason for this throughput improvement seems to be that the increased amount of processed data with a reduction in GC time is longer than the decreased amount of processed data by the WRITE operation overhead indicated in (1). All methods increased throughput if the WRITE operation ratio was low. This result seems due to the increased number of transactions along with reduction in the WRITE operation ratio, which is heavier compared to the READ operation. Our method has an overhead for WRITE operation as indicated in (1). This overhead reduces the margins between the other methods with an increase in the WRITE ratio. However, all methods require additional task for generating temporal data in addition to READ/WRITE operations. The effect of the overhead as indicated in (1) and the performance difference between READ and WRITE operations seems to be small.

In JDK1.6 HashMap + COW, the number of processed transactions increased with a decrease in the WRITE operation ratio is larger than that of CORC. This relation of the number of transactions is similar to the relation between JDK1.6 HashMap + COW and CORC methods, as indicated in (2). This throughput performance gap decreased because a portion of the operations that are not READ/WRITE operations exists.

5. Related Works

Besides cache, the in-memory database (DB) technique [8] is another technology that accelerates data processing using large memory. In-memory DB is a kind of DB that holds its data in memory. It can recognize a database language (e.g., SQL) query and return the search results. As with a disk, in-memory DB supports relations and indexes for data management, which lead to the support of complex searches or aggregate calculations. In contrast, many object cache systems provide simple data access interfaces such as KVS. Therefore, an object cache is not ideal for doing complex searches or aggregate calculations. Because an object cache does not require the interpreting of complex queries, such as SQL, simple data access can be performed faster than using in-memory DB. Business data management or analysis usually requires aggregate calculations. In contrast, business data processing requires simple data access. Therefore, by using a large object cache, we can improve the performance of business data processing.

There have been many studies on large-scale caches for managing large data sets. One example is a distributed cache that connects several cache nodes via a network. Another example is a large object cache with large memory similar to the one described

in this paper. These features are summarized by Xiulei [3] et al. They described that a large-scale cache requires performance, scalability, and availability to meet these requirements. There are three cache strategies. The first one is replication, which enhances availability and reliability. The second one is partitioning, which ensures scalability by maintaining data separately placed in distributed nodes. The last one is using a near cache, which accelerates the access speed of data. A near cache is a kind of client cache that is located on the node in which data processing is executed. The object cache described in this paper is a type of near cache. This object cache improves speed for data access and performance for data processing by caching data in the Java heap memory used for data processing.

Studies on cache management algorithms for effective utilizations have also been aggressively conducted. Even if a cache size is large, it is not always possible to maintain all data in the cache. Therefore, maintaining frequently accessed data in a near cache is necessary for accelerating data access. In addition, even if the size of a distributed cache can be used to enlarge memory space, maintaining frequently accessed data in a near cache is necessary for accelerating data access. Several types of selecting methods for frequently accessed data have been proposed [9].

One method for implementing near caches is placing cached data in external heap memory that is outside the heap memory by converting a Java object to a byte array. Hichens [10] et al. showed that the direct buffer provided by Java NIO can accelerate READ/WRITE operations on a byte array in the external heap memory and can improve data processing performance. The direct buffer can also prevent the increase in GC time because the external heap memory is not checked for collecting garbage. However, all data are serialized to the byte array if we use the external heap memory. To reduce this overhead, methods for reducing serialization overhead have been investigated [11], [12]. However, completely eliminating serialization overhead is difficult. Thus, data access performance decreases compared with an object cache using direct data references.

Studies on GC time reduction that is not solely targeted at an object cache have been conducted. Metronome GC is a GC algorithm that is modified to be applied to real-time systems. Metronome GC [13] divides normal GC operation into pieces. It periodically stops the Java program for a short time to execute a piece of GC operation. However, Metronome GC does not reduce the work required for GC, as stated in this paper. Concurrent Mark-Sweep GC [14] is also an algorithm for reducing GC pause time. This algorithm executes GC operations in parallel with the Java program. However, the concurrent Mark-Sweep GC algorithm requires finding the object changed by Java program execution. The overhead of this finding may significantly reduce throughput.

Our GC time reduction algorithm with the specific data management eliminates a part of the heap memory from finding the references in a GC operation. Obata [15] et al. proposed an explicitly managed heap memory that allows parts of Java heap memory not to be checked to collect garbage data. This explicitly managed heap memory prevents a garbage collector from managing all heap memory and decreases the GC time by explicitly controlling a specific region of memory. However, explicitly managed heap memory must be checked to find the references from the region when the garbage collector executes GC operation to another region from the explicitly managed heap memory. An increase in explicitly managed heap memory size results in enlargement of regions to which references are found, causing an increase in GC time. To solve this problem, the explicitly managed heap memory and our method can be combined to inhibit GC with long pause time and reduce the GC time of a sub-region of an object cache.

6. Conclusion and Future Work

We proposed an algorithm for reducing GC that incorporates novel data management method on a Java object cache. It is aimed at resolving a serious performance problem when using a large server memory as a large Java object cache. A Java object cache enables direct accesses to data and accelerates data access because cached data is placed in the heap memory. By using a data access method with COW, consistent direct data access is possible. Although, direct data access can accelerate data access, the increase in GC time is a problem for a large object cache. This problem is caused by the increase in the number of data to be checked to find references to the GC target. To solve this problem, we use a data management method for immutable regions with our GC algorithm. This method manages an object cache in two divided regions: an unclosed region, which permits an application to add or change the cached data, and a closed region, which does not permit an application to add or change the cached data. This method creates immutable data regions and eliminates necessary to find the references from them. Our experimental evaluation showed that the proposed method with 4 GB data processing, 100 GB object cache, and 1.3 GB collected-garbage regions decreased the GC time from 0.5 to 0.09 seconds. The evaluation also showed that this algorithm can reduce GC time by 1/4 and improve throughput by 40% compared to the previously proposed generational GC algorithm. The proposed GC algorithm can improve throughput and reduce GC time with a large object cache. However, since an object cache cannot hold an infinite amount of data, an algorithm for detecting and deleting unnecessary data should be implemented. We plan to implement several cache maintenance algorithms and evaluate their effectiveness in reducing GC time. This is for extending the proposed GC algorithm.

References

[1] Gosling, J., Joy, B., Steele, G.L. et al.: *The Java Language Specification*, Addison Wesley (1995).

[2] Blackburn, S.M. and Stanton, R.: The Transactional Object Cache: A Foundation for High Performance Persistent System Construction, *Proc. 8th International Workshop on Persistent Object Systems*, pp.37–50, August 30-September 1 (1998).

[3] Xiulei, Q., Wenbo, Z., Wei, W., Jun, W., Hua, Z. and Tao, H.: A Comparative Evaluation of Cache Strategies for Elastic Caching Platforms, *Proc. 2011 QSIC*, pp.166–175 (2011).

[4] Appel, A.W.: Simple Generational Garbage Collection and Fast Allocation, *Software Practice and Experience 19*, pp.171–183 (1989).

[5] Keith, M. et al.: Object-Relational Mapping, *ACM Queue*, Vol.6, Issue 3, pp.38–47 (2008).

[6] Wilson, P.R. and Moher, T.G.: A Card-Marking Scheme for Controlling Intergenerational References in Generation-Based GC on Stock

Hardware, *SIGPLAN Notices*, Vol.24, No.5, pp.87–92 (1989).
[7] Bak, L. et al.: The New Crop of Java Virtual Machines, *Proc. 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp.179–182 (1998).
[8] Silberschatz, A. Korth, H.F. and Sudarshan, S.: *Database System Concepts*, pp.429–474, McGraw-Hill International Edition (2011).
[9] Gu, X. and Ding, C.: On the Theory and Potential of LRU-MRU Collaborative Cache Management, *Proc. 2011 International Symposium on Memory Management*, pp.43–54 (2011).
[10] Hichens, R.: *Java NIO*, O'Reilly Media, Inc., pp.43–50 (2002).
[11] Palmer, N., Kielmann, T. and Bal, H.: Serialization for Ubiquitous System: An Evaluation of High Performance Techniques on Java Micro Edition, *UBICOMM*, pp.356–361, Spain (2008).
[12] Palmer, N., Miron, E., Kemp, R., Kielmann, T. and Bal, H.: Towards Collaborative Editing of Structured Data on Mobile Devices, *Proc. MDM2011*, pp.194–199 (2011).
[13] Auerbach, J., Bacon, D.F., Bomers, F. and Cheng, P.: Real-Time Music Synthesis in Java Using the Metronome Garbage Collector, *Proc. International Computer Music Conference*, pp.100–109, (2007).
[14] Printezis, T. and Detlefs, D.: A Generational Mostly-Concurrent Garbage Collector, *Proc. 2nd International Symposium on Memory Management*, *ISMM 2000*, Vol. 36, No.1, pp.143–154, (2000).
[15] Obata, M., Nishiyama, H., Adachi, M., Okada, K., Nagase, T. and Nakajima, K.: Explicitly Managed Heap Memory for Java, *IPSJ Journal*, Vol.50, No.7, pp.1693–1715 (2009).

Yasushi Miyata received his B.Eng. degree from Kyoto University in 2005, and M.Eng. degree from Kyoto University in 2007. He has been working at Yokohama Research Laboratory, Hitachi, Ltd. since 2007, and is now a Researcher. He has been working on middleware.

Motoki Obata received his Ph.D. degree from Waseda University in 2003. He has been working at Yokohama Research Laboratory, Hitachi, Ltd. and is now a Senior Researcher. He has been working on Java runtime environment.

Tomoya Ohta received his Ph.D. degree from Shizuoka University in 1998. He has been working at Yokohama Research Laboratory, Hitachi, Ltd. and is now a Senior Researcher. He has been working on language processor and Java runtime environment.

Hiroyasu Nishiyama received his Ph.D. degree from University of Tsukuba in 1993. He has been working at Yokohama Research Laboratory, Hitachi, Ltd. and is now a Supervisory Researcher. He has been working on language processor, optimizing compilers, and Java runtime environment.