An Explicit Dynamic Memory Management Scheme in Java Run-Time Environment

SooKang Bae[1], SungYoung Lee[1], Hyon Woo Seung[2], Taewoong Jeon[3]

Dept. Computer Engineering, Kyunghee University[1], {bsk,sylee}@oslab.khu.ac.kr

Dept. Computer Science, Seoul Women's University[2], hwseung@swu.ac.kr

Dept. Computer Science, Korea University[3], jeon@tiger.korea.ac.kr

ABSTRACT

In the Java Programming Language, objects generated by the keyword new are automatically managed by the garbage collector inside the Java Virtual Machine (JVM) unlike using the keywords free or delete in C or C++ programming language. This provides the way of freedom from memory management, which was such burden for the application programmers. The garbage collector, however, inherently has its own run time execution overhead. Thus it causes the performance degradation of JVM significantly. In order to mitigate the execution burden from the garbage collector, we propose a novel way of dynamic memory management scheme in Java environment. In the proposed method, the application programmers can explicitly manage the objects in a simple way, which in consequence the run-time overhead can be reduced while the garbage collector is under processing. In order to accomplish this, Java application should first call the APIs that are implemented by native Java, and then should call the subroutines depending on the JVM, which in turn support to keep the portability characteristic what Java has. In this way, we can not only sustain the stability in execution environments, but also improve performance of garbage collector by simply calling the APIs. Our simulation study shows that the proposed scheme improves the execution time of the garbage collector from 10.07 percent to 52.24 percent working on Mark-and-Sweep algorithm.

1. INTRODUCTION

Memories are called static or dynamic according to whether they are allocated in compile or run time. There are two kinds of memory management. One is the explicit dynamic memory management in which memories are generated and collected explicitly by the application. The other one is the automated dynamic memory management using an automatic memory manager like the garbage collector [1] [2].

The unnecessary complications created by explicit storage allocation are troublesome, because programmer should take whole responsibility for dynamic memory management. Failing to reclaim memory at the proper point may lead to slow memory leaks with unreclaimed memory gradually accumulating until the process terminates. Using garbage collection frees programmers from this burden. However, it is widely believed that garbage collection is quite expensive relative to explicit dynamic memory management. Even if a lot of work has been done in order to enhance the performance of garbage collection, the overhead caused by the automation of memory management still exists. This kind of fact is due to the garbage collector should determine whether an object is live or dead in run time. Consequently, such overhead may be cumbersome under the circumstances of the interaction between users and the system is quite frequent.

We propose a novel way of dynamic memory management in which programmers take the partial responsibility of collecting garbage objects without depending entirely upon the garbage collector. In that way, the runtime overhead of the garbage collector can be greatly reduced at the cost of programmers' burden. In the proposed method, a Java application invokes the API functions written in the pure Java, which, in turn, invoke the subroutines in the JVM(Java Virtual Machine), without losing the portability characteristic of Java. In other words, by using the proposed method, it is not only possible to maintain the stability of the application performance, but also to enhance the performance of the JVM by the programmer's simple calls to the API. The programmer's inconvenience is minimized by making the API as simple as in the explicit dynamic memory management. There are two ways to support the proposed method in the real-time system. One is the automated dynamic memory management scheme based on the explicit dynamic memory management. The other is the explicit dynamic memory management scheme based on the automated dynamic memory management. The latter scheme was adopted because it should be guaranteed that dynamically allocated memories will be collected by the garbage collector at last, even if they are not collected explicitly by the application.

In order to measure the performance of the proposed scheme, a Java run-time system was selected[3][4], since, in the JVM specification, it is stipulated that the method to collect objects explicitly created by programmers should be automated[4], and such a method is usually implemented by a garbage collector. The implementation was performed on the Kaffe JVM[5] whose source code is open to the public. After we measured the average execution time of the test applications, the result showed about 10% to 52% time reduction.

The proposed scheme is expected to be utilized in many application domains, ranging from embedded systems to enterprise computing environments. For the embedded systems in which memory resource is not abundant, the efficiency of the memory usage can be enhanced by the explicit collection of the dynamic memory. Also, the period of the garbage collection can be lengthened and, consequently, the run-time overhead of the garbage collector can be greatly reduced by using the scheme.

The remainder of this paper is organized as follows. Chapter 2 introduces previous works related to the explicit memory reclamation in the automated dynamic memory management system. Chapter 3 presents the proposed scheme in which programmers can perform explicit object reclamation while the garbage collector is under processing in the Java run-time environment. Chapter 4 describes the performance evaluation when the Java run-time system shares the burden of memory management with programmers. Chapter 5 discusses considerations for Java programmers or JVM developers using the proposed scheme. Chapter 6 concludes this paper and discusses possible future research directions.

2. RELATED WORK

Much work has been done in the area of the dynamic memory management for decades. It can mainly be divided into three management schemes: explicit memory allocation/deallocation scheme, region-based allocation/deallocation scheme and automated dynamic memory management scheme[6].

The first scheme is to explicitly manage dynamic memory using library functions like *malloc()* or *free()* in C. Even if this method can maximize the efficiency of the memory usage by providing more precise memory management, programmers are entirely responsible for the allocation and deallocation of the dynamic memory. Failing to reclaim memory at the proper point may lead to slow memory leaks with unreclaimed memory gradually accumulating until the process terminates. In the scheme proposed in this paper, however, the reclamation of the memory unreclaimed by the programmers is guaranteed by the garbage collector.

The second method, the region-based allocation/deallocation scheme, was initiated by Ross[7], in which objects are allocated in a region called "zone". This idea was further developed in many other researches where the region was called by different terms such as group[8] or arena[9]. In many cases, this method shows better performance than the first. However, it still carries the same problem as the first scheme since the allocation/deallocation of the dynamic memory must be done explicitly by the programmers.

The third method, the automated dynamic memory management scheme, has long been studied in order to release the application programmers' burden for the dynamic memory management and to enhance the software productivity. Especially, a lot of research has been done in the area of garbage collector. As stated above, the garbage collector however, inherently has its own run time execution overhead. Many algorithms to reduce the response time of the application have been proposed, such as the incremental garbage collector, the real-time garbage collector or the concurrent garbage collector[10]. But, they are not popular because their operations are too complicated to completely implement. Their major concern is not to reduce the run time overhead of the garbage collector, but to improve the application's response time or to predict activation time of the collector. In order to reduce the run time overhead of the collector itself and to overcome the memory fragmentation phenomenon, various garbage collecting algorithms have been published. Typical examples include reference counting, mark-and-sweep, copying and generational garbage collector. All of them have their own merits and demerits. Considering memory efficiency, mark-and-sweep garbage collection is much better than copying garbage collection. However, it has the problem with memory fragmentation. Although the mark-compact garbage collecting algorithm was proposed to remedy the memory fragmentation problem for the mark-and-sweep collection, it has additional run-time overhead due to the compact stage after mark and sweep stages. Consequently, response times of the applications using the mark-compact algorithm are slower. The idea behind this paper is not to develop a new garbage collecting algorithm. Its aim is to provide the systems using garbage collectors with an additional way to explicitly manage dynamic memory, and to take advantages from both ways.

Hans Boehm developed a garbage collection library to automate dynamic memory management for C/C++ which has the same old explicit dynamic memory management scheme without garbage collectors[11][12]. Dynamic memories created by calling *GC_malloc()* or *GC_malloc_atomic()* functions in the library can be reclaimed not only by the mark-and-sweep garbage collector, but also explicitly by the *GC_free()* function[13]. It is similar to the idea proposed in this paper, in that applications may explicitly reclaim memories allocated by automated dynamic memory managers. However, dynamic memories allocated by the *malloc()* function are not under the control of garbage collectors. Garbage collectors only manage objects created by their own library functions. Furthermore, it does not provide a way to reclaim a garbage object, together with all the objects that can be referenced from that object. Applications may replace *malloc()* with Boehm's *GC_malloc()* in order to utilize automatic garbage collection. However, it might cause a problem to use Boehm's library functions together with existing dynamic memory management functions. Suppose there exists an object *A* in the dynamic memory area under the control of Boehm's library, and another object *B* in the dynamic memory allocated by the *malloc()*. If *B* is the only object that has a reference to *A*, *A* could be regarded as a garbage and collected because Boehm's library does not know *B* has a reference to *A*.

In addition, a cache-conscious copying collection was proposed as a means to reduce run-time garbage collection overhead[14], and a hardware-assisted real-time garbage collection system was also introduced in order to overcome the problem with uncertain delay time of real-time garbage collectors[15]. None of the schemes, however, provide a method similar to our scheme in which Java programmers can explicitly collect objects, while the garbage collector is under processing.

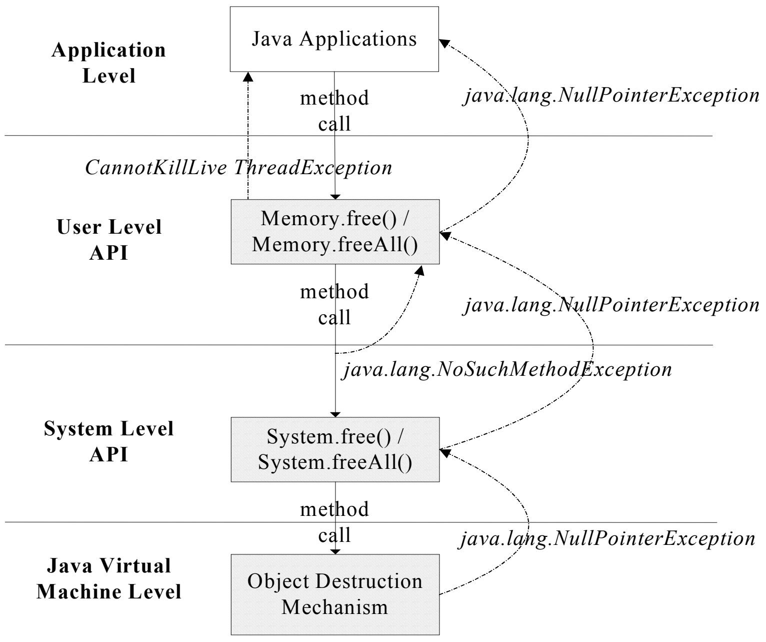
3. EXPLICIT OBJECT COLLECTION ALGORITHM

This Chapter describes the proposed scheme in which an object in the Java application level can be explicitly reclaimed by calling API methods. The algorithm was implemented using the Kaffe Java Virtual Machine[5].

3.1 Explicit Object Collection Model

The model proposed to support explicit object collection consists of four layers(Figure 1.) At the topmost layer, there is a Java Application Level which explicitly calls for object collection. Below the Application Level is a User Level API layer which calls the appropriate function if the JVM supports explicit object collection, or does nothing, otherwise. At the third layer, there is a System Level API which consists of native methods. A Java Virtual Machine Level is at the bottom, which performs actual object collection.

As shown in Figure 1, there are two methods, *Memory.free()* and *Memory.freeAll()* that provide Java applications with the object collection services. The former can be invoked to collect any one object, and the latter frees not only the object given as a parameter, but all the object reachable from the given object. Both methods are implemented in 100% pure Java, and send the application one of the three exception objects according to the situation. A *CannotKillLiveThreadException* object is sent when the application tries to collect a live thread object. A *java.lang.NoSuchMethodException* object is sent when the application tries in a JVM which does not support explicit object collection. The *java.lang.NoSuchMethodException* can be detected at the compile time of the source code, but the exception might occur when a method in a class compiled without errors tries to call the method in a JVM which does not support explicit object collection. The *java.lang.NullPointerException* object is sent when the application tries to collect an object already freed. This exception object is created and sent, not in the above two methods at the User level API, but at the Java Virtual Machine Level, as in Figure 1. In a normal situation, that is, when the JVM has an object collection method, an appropriate native method is called.



Application Level
Java Applications
method call
java.lang.NullPointerException
CannotKillLive ThreadException
User Level API
Memory.free() / Memory.freeAll()
method call
java.lang.NullPointerException
java.lang.NoSuchMethodException
System Level API
System.free() / System.freeAll()
method call
java.lang.NullPointerException
Java Virtual Machine Level
Object Destruction Mechanism

Figure 1: Four Layers of the Object Collection Model

Figure 2 shows the object collection mechanism in Figure 1 implemented in Kaffe. Two methods, *free()* and *freeAll(),* were created in the *System* class, and implemented in native methods. Functions related to the *System* class are implemented in *System.c* among Kaffe source codes, in which we defined the *java_lang_System_free()* function. This function calls *gcManFree()* function in *gc-incremental.c*, where garbage collecting functions of Kaffe are implemented. In Kaffe virtual machine, *void gc_heap_free(void* mem)* function *in gc-mem.c* is in charge of the actual memory-level(not object-level) garbage collection, and takes the address of the memory to be collected as a parameter.

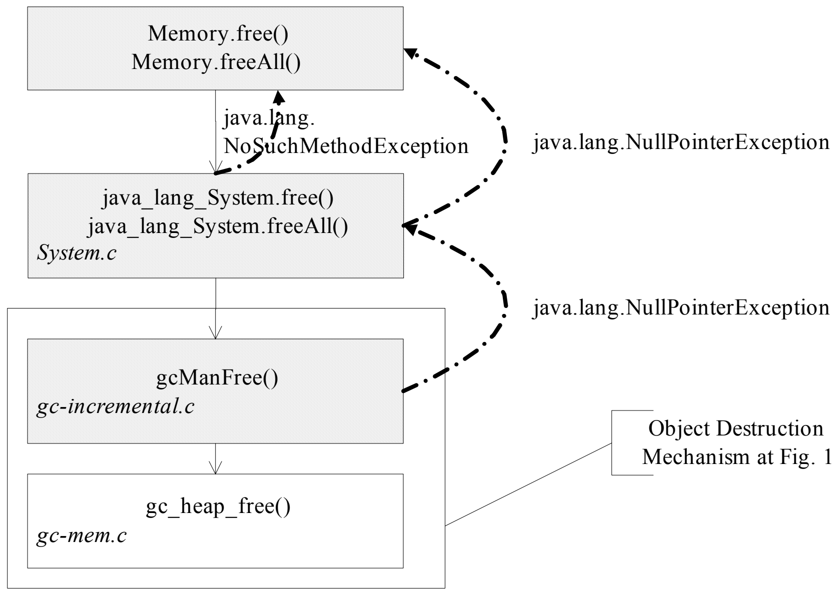


Figure 2: Implementation of Object Collection Mechanism in Figure 1

3.2 Algorithm 1: Explicit Object Collection Algorithm

Figure 3 shows the proposed explicit dynamic memory management algorithm, which is the implementation of *gcManFree()*. If the object to be collected is a live thread object, then a *CannotKillLiveThreadException* object is created and sent to the caller(lines 1, 2). If it is already freed, then a *NullPointerException* object is created and sent to the caller(lines 3, 4). If the live object has a *finalize()* method, then it is added to the *finalize* object list(line 6), all the application threads are stopped(line 7), and the *finalizer* thread, in charge of the *finalize* operation, is started(line 8). After the *finalize* operation is finished, all the application threads are resumed(line 9). With a lock for the Java heap acquired(line 10), the actual collection is performed by invoking the proper *free()* function already implemented in the JVM(line 11). Finally, the lock for the Java heap is released(line 12).

```
1: if the object is a live thread then
2:     create a KillThreadException object and Throw it
3: If the object is already freed then
4:     create a NullPointerException object and Throw it
5: If the object has a finalize() method then
6:     add it to the finalize object list
7:     stop all the application threads
8:     start the finalizer thread
9:     resume all the application threads
10: acquire a lock for the Java heap
11: call the proper free function that is already implemented in your JVM
12: release the lock for the Java heap
```

Figure 3: Algorithm 1: Explicit Object Collection Algorithm

The garbage collector starts its operation, at some point when no more object creation requests can be allowed due to the lack of available memory. If dead objects are freed explicitly by the proposed scheme, we have two gains. One is storage gain. If there was 1MB heap memory, for instance, and objects amounting to 200KB memory were explicitly freed, it could be said that there was 1MB+200KB available memory. The other is time gain. The run-time overhead for garbage collecting could be reduced, since the start time of the garbage collector could be postponed due to the increase of memory availability.

3.3 Algorithm 2: Algorithm supporting Object Reference Consistency

JVM specifications do not have any comment on the definition of object representation in the Java heap. Generally, there are two ways to access to an object in a JVM. One is indirect access through the handle as in PersonalJava of Sun MicroSystems, and the other is direct access as in Kaffe.

In the indirect access method, an object reference in a Java program is not pointing to an actual object, but pointing to its handle in the handle pool. In order to access an instance data, the object pool the handle is pointing must be referenced. Figure 4 shows the indirect object access. In this way, even if the position of an instance data is moved by copying garbage collector, there is no need to search for all the object reference variables in the Java program since all that need is change the pointer to the handle pool. However, the object access speed is relatively slow.

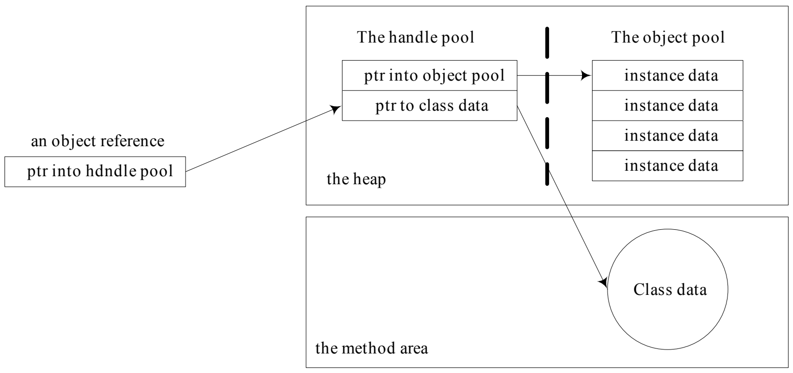


Figure 4: Indirect Object Reference using Handle

In the direct access method, an object reference variable is pointing directly to the instance data, as shown in Figure 5. While its object access speed is relatively fast, pointers of all the variables referencing to the object in the Java program must be changed, if the position of an instance data in the heap is moved by the garbage collector.

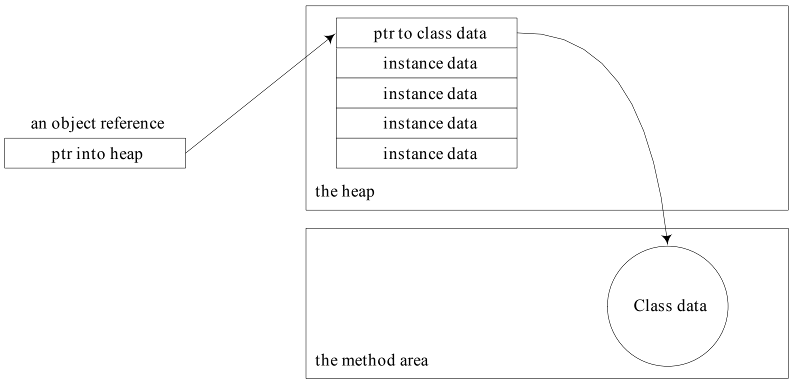


Figure 5: Direct Object Reference

Since the proposed scheme is implemented using Kaffe JVM, where the direct object referencing is used, consistency of the object reference must be guaranteed. An object, explicitly freed by a Java program, could be accessed later owing to the programmer's mistake. Therefore, the explicit object collection algorithm must be modified so that programmers could be aware that an access to the already freed object occurred.

```
 1: public class SampleClass {
 2:     public int member;
 3:     public SampleClass(int i) {
 4:         member = i;
 5:     }
 6:     public static void main(String[] args) {
 7:         SampleClass a = new SampleClass(1); // (i)
 8:         SampleClass b = a;                  // (ii)
 9:         System.out.println("a = " + a);
10:         System.out.println("b = " + b);
11:         System.free(a);                     // (iii)
12:         SampleClass c = new SampleClass(2); // (iv)
13:         System.out.println("c = " + c);
14:         System.out.println("b = " + b);
15:     }
16: }
```

Figure 6: An Example Code where Consistency of Object Reference is Damaged

When the reference to an object to be collected is shared by another variable, the consistency of the object reference could be damaged if another object of the same type is created at the same memory position. Figure 6 shows an example code for such situation. The changes of reference variables and objects in the heap at the positions (i) through (iv) in Figure 6 are shown in Figure 7.

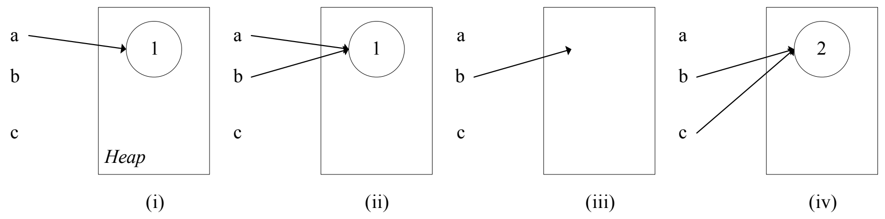


Figure 7: Illustration of the Object Reference Inconsistency in Figure 6

After execution of the Java code in Figure 6, reference variables 'a' and 'b' become both value 1 by lines 9 and 10. After execution of the lines 13 and 14, variables 'c' and 'b' become both value 2. Variable 'b' is a reference to the object to which variable 'a' is pointing. Even if the object that is referenced by 'a' is collected, 'b' still has the address of the old object((iii) of Figure 7). Since objects of the same size are stored in the same block in Kaffe, the object created in line 12 of Figure 6 is placed at the same position as that of the object created in line 7 and freed in line 11. Therefore, a NullPointerException occurs if an access using 'b' is attempted before line 12((iii) of Figure 7), but 'b' points to the object that 'c' is pointing after line 12((iv) of Figure 7). In order to solve this problem, all the variables having the reference to the object to be collected should be nullified. It is, however, so costly since the entire heap should be searched. Boehm's garbage collection library also has the similar object reference inconsistency problem.

To prevent the object inconsistency in the explicit object collection scheme using Algorithm 1 in Section 3.2, we propose Algorithm 2 which supports the object reference consistency using, what we call, "Position Version Technique". Most of JVM including Kaffe utilizes 32-bit pointers to access objects in the Java heap. Since positions of the objects to be created are aligned in blocks of 8-byte, in the actual implementation of the virtual machine, the lower 3 bits have always zero value. In this paper, these 3-bit area is defined as Position Version(or Address Version), in which position versions of objects are stored.

Figure 8 shows the contents of a 32-bit pointer with lower 3-bit Position Version. Again, a position version of the object that is pointed by the 29-bit address is stored in the Position Version. For instance, if an object is created at a certain address according to the allocation policy of the virtual machine, the Position Version value in the object header information stored at the position is incremented by 1. Then, the increased new value is stored in the lower 3 bits of the pointer. Accordingly, there can be 8 Position Versions, from 0 to 7. Even if an object is collected, the version information of the object is left at the position. When a new object is created there later, that information is referenced and incremented by 1.

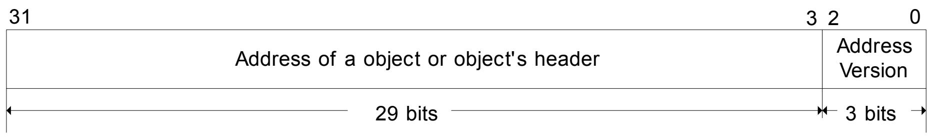


Figure 8: 32-bit Pointer with Lower 3-bit Address Version

In Figure 7(iv), both 'b' and 'c' are pointing the same object. However, the Position Version stored in 'b' is different from that of the object, while the Position Version stored in 'c' is the same. Therefore, the virtual machine can detect the difference and restrict any operation or access by the reference 'b'. In such cases, our Algorithm 2 creates a *java.lang.NullPointerException* object, so that java application programmers can implement their own code using "try-catch" clause, in case an access to an already freed object occurs. Algorithm 2 is the explicit object collection algorithm(Algorithm 1) with the "Position Version Technique", and is shown in Figure 9, where the appended parts are underlined.

```
1: if the object is a live thread then
2:       create a ThreadKillException object and Throw it
3: If the object is already freed then
4:       create a NullPointerException object and Throw it
5: If the position versioin of the object reference is different from it of the object itself then
6:       create a NullPointerException object and Throw it
7: If the object has a finalize() method then
8:       add it to the finalize object list
9:       stop all the application threads
10:      start the finalizer thread
11:      resume all the application threads
12: acquire a lock for the Java heap
13: call proper free() function that is already implemented in your JVM
14: release the lock for the Java heap
```

Figure 9: Algorithm 2-Explicit Object Collection Algorithm with Position Version Technique

4. PERMORMANCE EVALUATION

In this Chapter, the proposed scheme is applied to the two typical garbage collection algorithms, mark-and-sweep and copying, for the performance evaluation. First, we test the mark-and-sweep algorithm with and without our algorithm, and compared two cases. Next, copying garbage collection algorithm is tested.

4.1 Explicit Collection Scheme with Mark-and-Sweep Algorithm

We implemented the algorithm of Figure 9 on a Kaffe JVM, and carried out an experiment on the performance of the algorithm according to the number of objects to be freed and the number of objects referenced by the application program. We assumed the objects to be explicitly freed do not have *finalize()* method. The heap size is fixed to 5, 10 and 20 MB. For each case, we measured the execution times of the application program, increasing the number of objects that have references from the application program(variable *L* in Figure 10) from 0 to 100,000 by 10,000, and the total number of objects to be created from 1 million to 10 millions by a million. We tested on a Pentium-III 600MHz with 256KB cache memory and 128MB RAM, using the Linux *time command*. The JVM was Kaffe 1.0.6 version, and the source code was compiled in the debug mode.

```
public class UsingGC {
    int member;
    public static void main(String[] args) {
        int L = Integer.parseInt(args[0]);
        int G = Integer.parseInt(args[1]);
        UsingGC[] ugc = new UsingGC[L]
        for (int i = 0; i < L; i++)
            ugc[i] = new UsingGC();
        for (int i = 0; i < G; i++)
            new UsingGC();
    }
}
```

Figure 10: Example Codes for Performance Measurement

The left-side code in Figure 10 shows the case garbage objects are collected using the garbage collector only. It was executed on a general Kaffe without the support of the explicit collection. The right-side is the code in which programmers can call *MM.free()* method to explicitly collect garbage objects, and was executed on a Kaffe modified to support the explicit collection. In the left-side code, the references to the objects created by the first *for* loop are stored in *gcf*, an array object of the *GCFree* type. The number of references to be stored is determined by the first command line argument(*arg[0]*). Since references to the objects created in the second *for* loop are not stored in any place, they are garbages. The garbage collector simply performs the garbage collection by the statement *new GCFree();*. In the right-side code, the collection of the garbage objects created by the second for loop is done explicitly by the *MM.free()* method.

The execution times for the above two Java application programs are shown in Figures 11 through 14. In Figures, *L* stands for the number of objects the garbage collector considers live in every garbage collection period, and corresponds to the integer variable *L* in Figure 10. The variable *G* in Figure 10 stores the number of garbage objects without references. *H* stands for the size of the Java heap. *L* number of objects will be scanned by the garbage collector and stay live, while *G* number of garbage objects will be collected in every execution period. The legend "GC" implies the execution on a general JVM depending on the garbage collector only without the explicit collection, and the legend "Explicit" the execution with the proposed explicit collection.

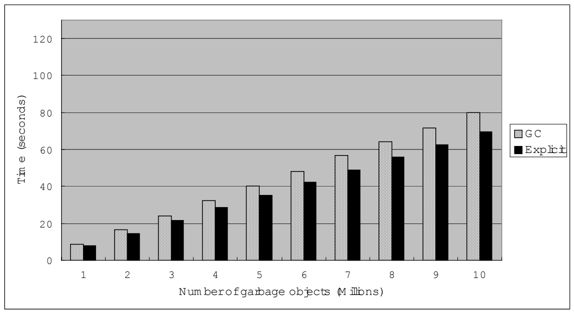


Figure 11: Execution Time with Mark-and-Sweep, where L=0, H=5(MB)

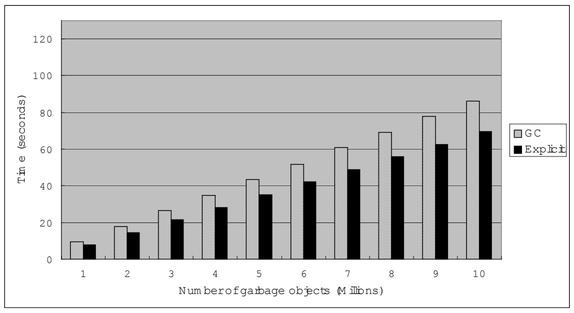


Figure 12: Execution Time with Mark-and-Sweep, where L=30,000, H=5(MB)

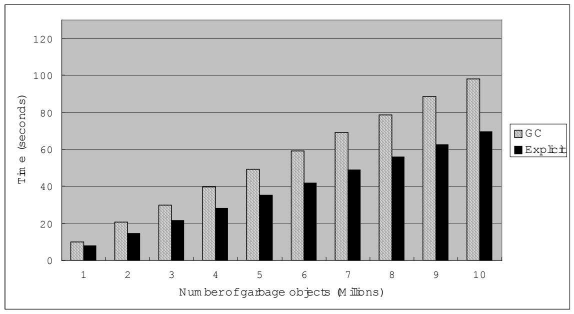


Figure 13: Execution Time with Mark-and-Sweep, where L=60,000, H=5(MB)

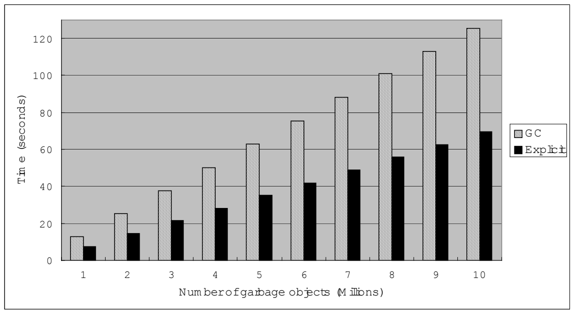


Figure 14: Execution Time with Mark-and-Sweep, where L=90,000, H=5(MB)

Figure 11 where *L* is 0 and *H* is 5 MB shows the result of the execution speed measurements for both cases, "GC" and "Explicit", as the number of garbage objects, *G*, increases from 1 million up to 10 millions. As shown in the figure, our scheme is much better than the case when only garbage collector is used. We can see the similar result in Figure 12 where *L* is 30,000. Compared with Figure 11, the execution speed is slower when only garbage collector is used("GC"), since there are 30,000 objects for the garbage collector to operate on in every collection period.

In Figures 13 and 14, *L* is increased to 60,000 and 90,000, respectively. Compared with Figure 11, we can see the execution speed in Figure 14 is remarkably slower when only garbage collector is used("GC"). In the case of "Explicit", however, there is no difference in the execution speed, even if *L* is increased a lot. The reason is that the garbage collector did not run at all, since the memory size of the explicitly freed objects was commensurate with that of the objects the application requested to create. Therefore, the execution times for the "Explicit" shown in the Figures can be said to be the pure cost of the application, excluding the overhead of the garbage collector. If the application explicitly collects garbage objects and reclaims the memory space occupied by them before a new cycle of garbage collection begins, the operation of the garbage collector can be put off, and, consequently, the number of execution times of the garbage collector throughout the entire execution of the application can be reduced. When the number of live objects(*L*) is small as in Figure 11, the scanning time of the garbage collector, if any, could not be long. That is why there is little difference between "GC" and "Explicit" in Figure 11. But, even in such cases, "Explicit" is still better than "GC". The reason is that the time for initialization and postprocessing of the garbage collector is needed in every collection cycle, even if the number of objects to be scanned is very small.

The simulation study shows that the proposed scheme improves the execution time of the garbage collector from 10.07 % to 52.24 % working on mark-and-sweep algorithm. The lowest 10.07 % improvement was achieved when the number of live objects(*L*) was 10,000 and the number of garbage objects(*G*) was 990,000. The highest 52.24 % gain was achieved when *L* was 100,000 and *G* was 8,900,000.

Figures 15 through 18 are results of the simulation performed under the same situation as Figures 11 through 14, except that the size of the Java heap is 10 MB. Compared with the previous four Figures, two facts can be observed. First, the performance gap between the "Explicit" JVM and the "GC " JVM is reduced as the size of the Java heap grows. This is because the execution period of the garbage collector gets longer and, consequently, the number of executions is reduced due to the bigger size of the heap, even if only the garbage collector is utilized without the explicit collection scheme. Second, the size growth of the Java heap does not have a great effect on the JVM using the proposed scheme, since the memory space of explicitly freed objects can be readily reused.

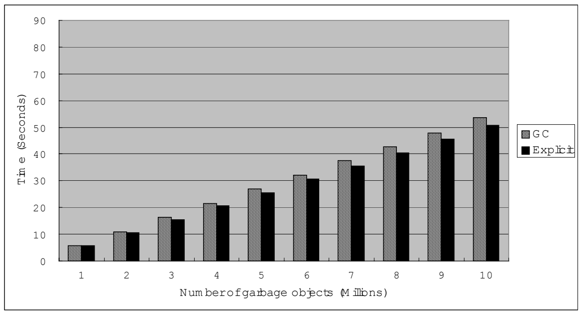


Figure 15: Execution Time with Mark-and-Sweep, where L=0, H=10(MB)

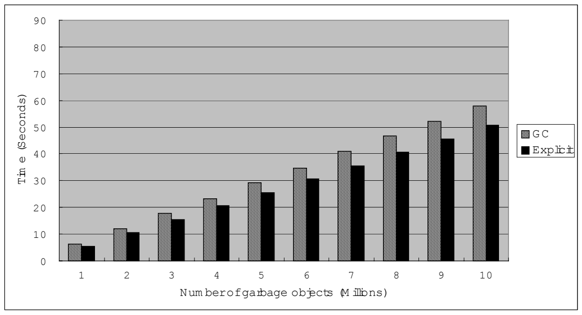


Figure 16: Execution Time with Mark-and-Sweep, where L=30,000, H=10(MB)

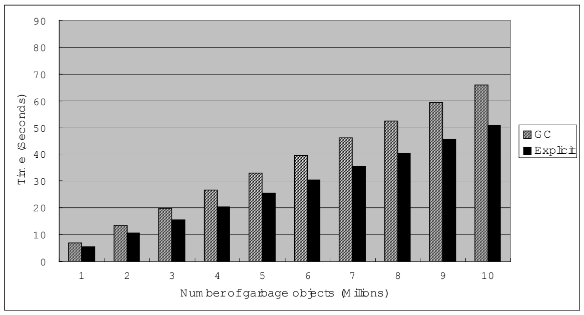


Figure 17: Execution Time with Mark-and-Sweep, where L=60,000, H=10(MB)

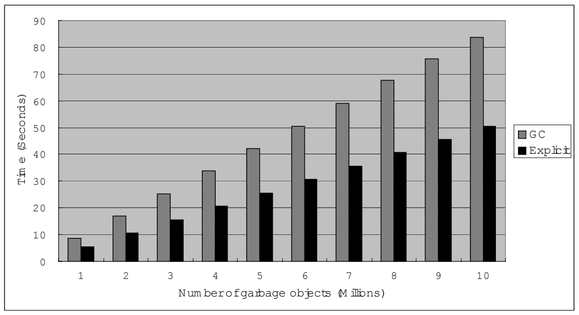


Figure 18: Execution Time with Mark-and-Sweep, where L=90,000, H=10(MB)

4.2 Explicit Collection Scheme with Copying Algorithm

In copying garbage collection algorithm, the space devoted to the Java heap is subdivided into two contiguous semi-spaces. During normal program execution, only one of these semi-spaces is in use[16]. Accordingly, the efficiency of dynamic memory usage could be reduced below 50%. However, copying algorithm is quite faster than mark-and-sweep algorithm in dynamic memory allocation, and does not have the problem with memory fragmentation.

In order to evaluate the performance of the proposed explicit collection scheme in copying algorithm, we replaced Kaffe's mark-and-sweep garbage collector with the copying collector we developed. It is meaningless to compare the two collectors on the machine we developed, since the original garbage collector of Kaffe is optimized, but the copying collector we implemented on Kaffe is not optimized.

The position where object allocation in copying algorithm takes place is the place pointed by the free pointer of the semi-space currently in use. When the running program demands an allocation that will not fit in the unused area of the current semi-space, the program is stopped and copying garbage collector is called to reclaim space. All of the live objects are copied from the current semispace(fromspace) to the other semispace (tospace). This process is often called a flip. Once the copying is completed, the tospace is made the current semi-space, and program execution is resumed. Thus, the roles of the two spaces are reversed each time the garbage collector is invoked. Figure 19(a) shows an initial semi-space and (b) shows the semi-space right before a flip occurs.

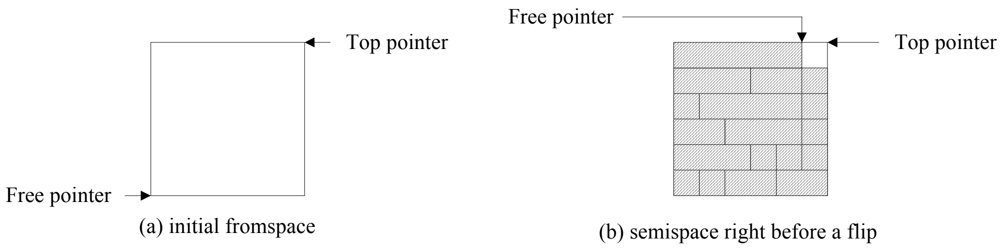


Figure 19: Initial Semispace and Semispace right before a Flip

Free area can be created in the current semi-space if the explicit object collection scheme is applied to the copying collector. However, there is no way to appropriately reuse such area(the shaded area in Figure 19(b)) before a new collection cycle begins, that is, a new flip occurs, since simple increment of the free pointer is the allocation policy of copying collector. Consequently, even if the explicit dynamic memory collection is applied, the delay of the garbage collection can not be expected. It can be proven through experiments.

Figures 20 and 21 where *L* is 0 and 30,000, respectively, when the heap size is 5 MB show the result of the execution speed measurements for both cases, "GC" and "Explicit", as the number of garbage objects, *G*, increases from 1 million up to 10 millions. As shown in the figures, there is no difference between both cases.

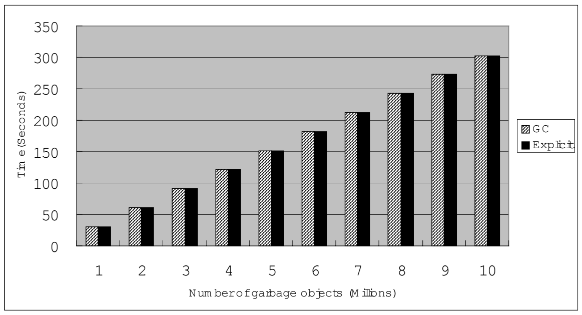


Figure 20: Execution Time with Copying Collector, where L=0, H=5(MB)

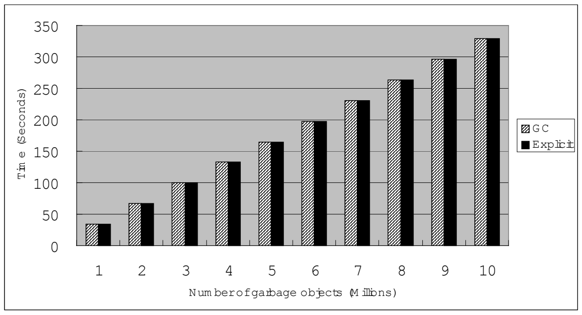


Figure 21: Execution Time with Copying Collector, where L=30,000, H=5(MB)

We tested the execution times with various *L*'s and *H*'s. As shown in Figures 22 through 25, the result was the same, that is, no difference between both cases. Comparing Figure 21 and Figure 22, we just can observe the application's execution time decreased when the heap size increased.

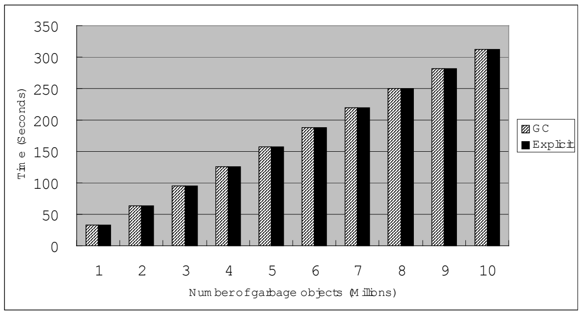


Figure 22: Execution Time with Copying Collector, where L=30,000, H=10(MB)

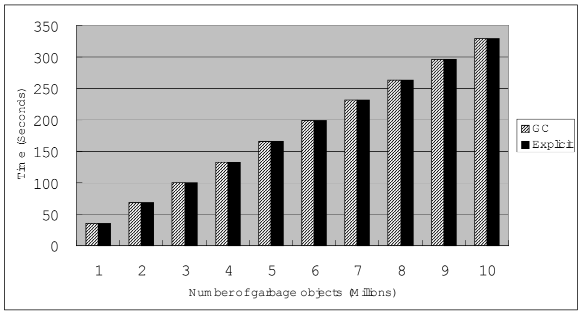


Figure 23: Execution Time with Copying Collector, where L=60,000, H=10(MB)

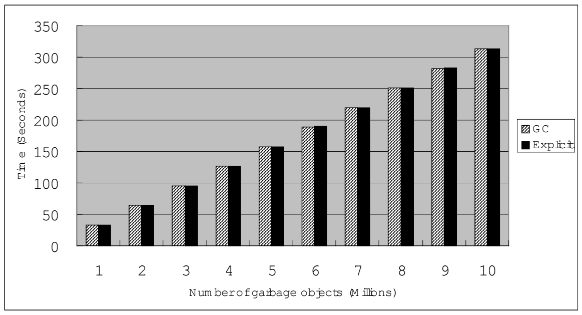


Figure 24: Execution Time with Copying Collector, where L=60,000, H=20(MB)

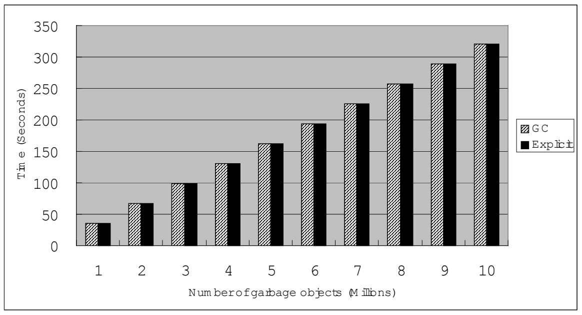


Figure 25: Execution Time with Copying Collector, where L=90,000, H=20(MB)

In order to reuse the memory area of the explicitly collected objects before the copying garbage collector is activated, the memory allocation policy of the collector should be changed. The addresses of explicitly collected memories could be stored in a queue or a stack. When no more objects can be created by incrementing the free pointer, available memories, if any, could be reused after searching the queue/stack. Various searching techniques could be applied for the memory allocation, but it might aggravate the fast memory allocation scheme of copying garbage collector. This means, when copying collector is used in real-time systems, the amount of time needed to process dynamic memory allocation can not be fixed in advance any more, and, consequently, execution times of the tasks utilizing dynamic memories in such systems would not be readily predicted.

5. DISCUSSION

This Chapter discusses some considerations for Java programmers or JVM developers, when they are to make application programs in the systems where the proposed explicit scheme is applied.

5.1 Collection of java.lang.Thread Object

The collection of thread objects using the explicit object collection API should be restricted. If a thread object currently in execution would be collected, the application might be suddenly terminated depending on the implementation of the virtual machine. So, a *CannotKillAliveThreadException* object will be thrown to the caller in our implementation, in case an active thread object is to be collected.

5.2 Collection of Object Having Method Being Executed by Thread

The following situation might occur. While a thread *T1* is in execution, calling a method of an object *O*, another thread *T2* would attempt to explicitly collect *O*. If the object *O* would be collected, *T1* would lose the codes to execute. A possible plan to prevent the situation is, when an object is designated to be collected, to compare it with the object pointed by "*this*" pointer of each stack frame in the Java stack of all the threads. If the same object is found, the collection request for the object could be ignored. Another possible plan is to store all the object collection requests in a queue and process them later. Then, whenever each stack frame of all the threads is popped up, every object in the queue is compared with the object pointed by "*this*" pointer of the popped-up frame. If the same object is found, it could be collected explicitly. Another possible plan is to throw an exception object instead of ignoring the collection request.

5.3 Collection of Object for which a Thread Waits to Get a Lock

If an object *O* is locked by a thread *T1*, there could be other threads blocked to get a lock for *O*. After *T1* finishes executing the synchronized method or block, one of the threads waiting for a lock is selected and gets a lock for *O* according to the implementation policy of the JVM. Before it gets a lock, another thread *T2* could attempt to explicitly collect *O*. If *O* is collected, the threads waiting for the lock would be blocked forever. One possible solution is to ignore the collection request for *O* if there are threads waiting for a lock for it. Another possible solution is to throw an exception object, so that the programmer can handle the situation as an exception.

6. CONCLUSION AND FUTURE WORK

Garbage collectors and explicit dynamic memory management systems have been studied independent of each other. Practically speaking, however, neither the execution time overhead of the garbage collector nor the programmer's overhead for the explicit memory management can be ignored. We proposed a novel way of dynamic memory management in which programmers take the partial responsibility of collecting garbage objects without depending entirely upon the garbage collector. In that way, the runtime overhead of the garbage collector can be greatly reduced at the cost of programmers' burden.

We also introduced problems and solutions about the object reference inconsistency, collection of an object having a method being executed by a thread, and collection of an object for which a thread is waiting to get a lock. The Position Version Technique was proposed for a solution to the first problem.

The proposed scheme was implemented and tested for performance evaluation on a Kaffe JVM with the mark-and-sweep garbage collector. The simulation study showed that the proposed scheme improved the execution time of the garbage collector from 10.07 % to 52.24 % working on mark-and-sweep algorithm. The more the number of live objects, and the smaller the size of the Java heap, the performance improvement was larger.

In order to evaluate the performance of the proposed explicit collection scheme in copying algorithm, we replaced Kaffe's mark-and-sweep garbage collector with the copying collector we developed. No performance difference was found between the case when copying garbage collector worked alone and the case when our explicit scheme worked together. The reason is that, even if the objects in the currently active semi-space are explicitly collected, the execution cycle of the copying garbage collector does not change. In order to reuse the memory area of the explicitly collected objects before the copying garbage collector is activated, the memory allocation policy of the collector should be changed. Part of our future study is to modify the allocation policy of copying garbage collector to make the most of the explicit object collection scheme.

7. REFERENCES

[1] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles, "Dynamic Storage Allocation: A Survey and Critical Review," International Workshop on Memory Management, Lecture Notes in Computer Science, Vol.986, pp.1-116, 1995.

[2] Paul R. Wilson, Uniprocessor Garbage Collection Techniques, In Yves Bekkers and Jaques Cohen, editors, Proceedings of the 1992 International Workshop on Memory Management, pp.1-42, St Malo, France, pp.17-19, 1992.

[3] J. Gosling, B. Joy, and G. Steele, The Java Language Specification, Addison-Wesley, Boston, 1996.

[4] T. Lindholm, and F. Yellin, The Java Virtual Machine Specification, Addison-Wesley, Boston, 1997.

[5] Kaffe.org, Kaffe Java Virtual Machine, http://www.transvirtual.com and http://www.kaffe.org.

[6] David Gay and Alex Aiken, "Memory Management with Explicit Regions," In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, number 33:5 in SIGPLAN Notices, pp.313-323, 1998.

[7] D. T. Ross, "The AED free storage package," Comm. ACM, Vol.10, No.8, pp. 481-492, 1967.

[8] Yuuji Ichisugi and Akinori Yonezawa, "Distributed garbage collection using group reference counting," In OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems, 1990.

[9] David R. Hanson, "Fast allocation and deallocation of memory based on object lifetimes," Software Practice and Experience, Vol.20, No.1, pp.5-12, 1990.

[10] Richard Jones, and Rafael Lins, Garbage Collection: Algorithms for Automatic Dynamic Memory Management, John Wiley & Sons, England, 1996.

[11] Hans-Juergen Boehm, and Mark Weiser, Garbage Collection in an Uncooperative Environment, Software Practice and Experience, Software Practice & Experience, vol.18, No.9, pp.807-820, 1988.

[12] Hans-Juergen Boehm, Space Efficient Conservative Garbage Collection, Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation, Vol. 28(6) of ACM SIGPLAN Notices, pp 197-206, 1993.

[13] Hans-Juergen Boehm, A garbage collector for C and C++, http://www.hpl.hp.com/personal/Hans_Boehm/gc/index.html.

[14] Henry G. Baker, Cache-conscious copying collection, In OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems, 1991.

[15] Kelvin D. Nilsen and William J. Schmidt, "A high-performance hardware-assisted real time garbage collection system," Journal of Programming Languages, Vol.2, No.1, 1994.

[16] Henry Baker, "List processing in real time on a serial computer," CACM, Vol.21, No.4, pp.280-294, 1978.