# A High Performance Garbage Collector for Java[1]

Chia-Tien Dan Lo, Witawas Srisa-an[2], and J. Morris Chang[3]

Department of Computer Science
University of Texas at San Antonio
San Antonio, TX, 78249, USA
danlo@ieee.org

*Parallel, multithreaded Java applications such as web servers, database servers, and scientific applications are becoming increasingly prevalent. For these applications, the dynamic memory manager (i.e., the garbage collector) is often a bottleneck that limits program performance and processor utilization on multiprocessor systems. Traditional garbage collectors suffer from long garbage collection pauses (stop-the-world mark-sweep algorithm) or inability of collecting cyclic garbage (reference counting approach). Generational garbage collection, however, is based only on the weak generational hypothesis that most objects die young. In this paper, a new multithreaded concurrent generational garbage collector (MCGC) based on mark-sweep with the assistance of reference counting is proposed. The MCGC can take advantage of multiple CPUs in an SMP system and the merits of light weight processes. Furthermore, the long garbage collection pause can be reduced and the garbage collection efficiency can be enhanced. To simulate the real-world workload (i.e., the SPECjvm98 benchmark), we have implemented the proposed scheme into the Kaffe JVM version 1.0.6 running on Linux RedHat 6.2. Measurement results indicate that the MCGC improves the garbage collection pause time up to 96.75% over the traditional stop-the-world mark-sweep garbage collector. Moreover, the MCGC receives minimal time and space penalties as shown in the report of the total execution time, the memory footprint and the sticky reference count rate.*

**Index Terms—** dynamic memory management, object-oriented programming, multithreaded programming, Java Virtual Machine, concurrent garbage collection system, modified buddy system, parallel garbage collector

## 1. Introduction

While a battery of concurrent garbage collectors have been proposed in the literature [BR 01, DeT 90, DKP 00, DL 93, DLM 76, Dom 00, HuW 98, JoL 98, LP 01, Rov 85], very few of them have been implemented and especially been incorporated into Java virtual machines (JVMs) [BR 01, DKP 00, Dom 00, LP 01].

The first concurrent garbage collector using the tricolor abstraction (white, grey, black) is pro-posed by Dijkstra, et al. [DLM 76]. In their design, the mutator and the garbage collector can run con-currently. However, in the original tricolor reasoning, the marker must be strictly followed by the sweeper because the white object could be marked black if the marking phase is not done yet. This restriction may potentially result in a larger memory footprint because the identified garbage objects are not swept timely.

Recently, the development of concurrent garbage collection for Java is based on either reference counting or mark-sweep. The work based on reference counting is the Recycler proposed by Bacon and Rajan

1.Partial support for this work is provided by the National Science Foundation through grants CCR-0098235 and CCR-0113335.
2.Witawas Srisa-an is in Department of Computer Science and Engineering, University of Nebraska at Lincoln. His email is witty@cse.unl.edu.
3.J. Morris Chang is in Department of Electrical and Computer Engineering, Iowa State University. His email is morris@iastate.edu

[BR 01] and the on-the-fly reference counting garbage collector proposed by Levanoni and Petrank [LP 01]. On the other hand, the generational on-the-fly garbage collector proposed by Domani et al. [Dom 00, DKP 00] is based on mark-sweep.

Although both the Recycler [BR 01] and the on-the-fly reference counting (RC) garbage collector [LP 01] are based on reference counting, there are differences between them. Firstly, the Recycler uses a mutator buffer to solve the synchronization issue in updating the reference counter whereas the on-the-fly RC introduces the sliding view idea. Secondly, the Recycler is implemented into Jalapeno JVM whereas the on-the-fly RC is implemented into Sun JVM. Thirdly, the Recycler adopts a novel on-the-fly cycle detector whereas the on-the-fly RC collects cyclic garbage by seldom running a marker. Fourthly, the Recycler handles the sticky reference count by using a hash table to keep the real reference count whereas the on-the-fly RC uses the marker to restore it.

Among the recent development of concurrent garbage collectors, the generational on-the-fly garbage collector [Dom 00, DKP 00] is a generational garbage collector based on the work of Doligez and Leroy [DL 93]. Although there are young and old generations, the on-the-fly collector does not move objects. Nevertheless, this approach only performs well in some cases but not always. Moreover, some quantitative performance analyses such as maximal garbage pause, etc., have not been reported.

Other work such as the multithreaded generational garbage collector for ML is proposed by Doligez and Leroy [DL 93]. Their garbage collector, called "quasi real-time" collector, combines two generations in which a copying collector works for the young generation and a marking collector collects on the old generation. All threads' stacks and minor heaps belong to the young generation whereas the major heap shared among all threads is considered as the old generation. The surviving objects from the copying collector in the young generation will be copied into the major heap which is collected by a mark-and-sweep garbage collector.

In 1998, a very concurrent mark-sweep garbage collector (VCGC) for SML/NJ, proposed by Huelsbergen and Winterbottom, can be used to run a marker, a mutator and a sweeper concurrently [HuW 98]. In this design, an epoch is used to interpret colors abstracted in tricolor reasoning. By separating three epochs, one for the mutator, one for the marker and the other for the sweeper, the mutator thread and GC threads can work together. However, for example, the marking effort can be further reduced drastically if the floating garbage is collected in a timely manner.

In this paper, a multithreaded concurrent generational garbage collection (MCGC) algorithm is proposed and its performance is measured. Our goal is to keep the mutators running with minimal interruption from the garbage collection threads such as the marker and the sweeper. Meanwhile several mutators can call the new instruction in parallel. The design of the MCGC algorithm is based on the Dijkstra's tricolor abstract, the VCGC algorithm and the mark-sweep algorithm with the assistance of reference counting. The MCGC algorithm can collect floating garbage on the fly and outperforms the VCGC algorithm and the pure stop-the-world mark-sweep garbage collection. The MCGC algorithm has been implemented into the Kaffe JVM version 1.0.6. Measurement results obtained by running the SPECjvm98 benchmark on the Kaffe JVM with/without the MCGC algorithm in Linux RedHat v6.2 are presented. The results show that the maximal garbage collection pause for the MCGC is less than 8.8 milliseconds (javac) and its average garbage collection pause is less than 1.8 milliseconds (javac and mtrt). The speedup of the GC pause over the stop-the-world mark-sweep GC can be up to 182 and 177 for maximal GC pause and the average GC pause, respectively.

## 2. Previous Work: Very Concurrent Garbage Collection (VCGC)

The very concurrent garbage collection was proposed by Huelsbergen and Winterbottom in 1998 [HuW 98]. The original design aimed to run a mutator thread, a marker thread and a sweeper thread concurrently without explicit fine-grain synchronization. To achieve their goal, an innovative coloring scheme is devised. Memory objects are distinguished by their colors. Colors are interpreted as a function (*COLOR*) of *epoch*, i.e., *epoch* modulo 3. The mutator color, the marker color and the sweeper color are defined as *COLOR(epoch), COLOR(epoch - 1)* and *COLOR(epoch - 2)*, respectively. The mutator thread associates each newly allocated object with the mutator color. The marker thread traverses the object reference graph and brings those reachable objects to have the mutator color. It is worth noting that the marker thread is the only thread that can change objects' colors. The sweeper thread, on the other hand, reclaims garbage objects that have the sweeper color. Once an object becomes garbage, only the sweeper can access it. Thus, there is no need for synchronization between the marker and the sweeper or the mutator and the sweeper. However, the marker and the mutator still require some synchronization which is resolved by applying a store set.

In the VCGC algorithm, a store set is used to solve the mutator and the marker handoff. It records the current content of references prior to the mutator updates them. In [WJN 95], this is called a snapshot-at-the-beginning algorithm since it retains the data reachable from the roots at the beginning of an epoch into the next epoch. By the time the marker and the sweeper finish, the store set is examined until all the objects are traversed. The mutator is then suspended and this concludes an epoch. The root set is obtained from the mutator at the end of an epoch and used by the marker for the next epoch. All the threads are deleted and the next epoch starts.

### 2.1 Problems with the very concurrent garbage collection (VCGC)

The very concurrent garbage collection algorithm has been proposed and proven to outperform the traditional generational garbage collection in terms of the garbage collection pause time. The major advantage is that it allows the mutator, the marker and the sweeper to run concurrently. This is a great improvement over Dijkstra, et al.'s algorithm [DLM 76] in that the sweeper must be executed strictly following the marker. However, we have found some of its drawbacks which are summarized as follows:

- This algorithm is limited to one mutator thread and one marker thread. When introducing multiple mutators and markers, the synchronization between mutators and markers, among mutators, or among markers becomes very complex.
- There are two scenarios which prolong the marking time. Firstly, the mutator allocated a lot of objects remaining alive in the previous epoch. Secondly, most of the memory objects in the entire heap are long lived. One of the reasons for this long marking time is that the marker does not have any information about an object's age. Thus, tenure objects are repeatedly marked. As a result, the memory footprint may be larger because the garbage may not be collected in a timely manner.
- The memory footprint may be dominated by the sweeper if there is a large amount of garbage in some epoch. More memory from the operating system may be requested by the mutator because of the longer sweeping time.
- The sweeping time may degrade the VCGC efficiency because the sweeper needs to go over the whole heap even though there is no garbage.
- The algorithm adopts the snapshot-in-the-beginning approach which determines liveliness at the beginning of an epoch. Therefore, any garbage objects that are created in that epoch may not be detected. However, they will be collected in the next epoch. Consequently, the memory footprint can be increased.
- The threads are created at the beginning of an epoch and destroyed at the end of it. The thread creation and destruction costs may be intolerable in some real-time systems.

Base on the above analyses, an improved version of the VCGC algorithm is proposed. The new algorithm is explained in detail in the next section.

## 3. The Proposed Multithreaded Concurrent Generational Garbage Collection Algorithm Version 0 (MCGCv0)

In this section, a new multithreaded concurrent generational garbage collection algorithm (MCGC) is introduced. The proposed algorithm overcomes all the problems of the VCGC algorithm mentioned in the previous section. The MCGC is based on the VCGC with the concept of generational garbage collection that takes the advantage of short lived objects. This allows young garbage to be reclaimed in the same epoch.

To clearly demonstrate the mechanism of the proposed algorithm, only one mutator, one marker and one sweeper are assumed in the version 0 (MCGCv0). In doing so, some synchronizations are eliminated but the algorithm can be described effectively. In Section 4. the full version of the MCGC where multiple mutator threads are included is described.

**Figure 1 The Proposed MCGC (Version 0)**

```
(1) int epoch = 2;
(2) root_set_t roots = {}, RS_set = {}, garbage_list = {};
(3) thread_t mutator, marker, sweeper;
(4) thread_create_daemon mutator, marker, sweeper;
(5) loop forever {
(6)    thread_resume(mutator(RS_set, COLOR(epoch)));
(7)    thread_resume(marker(roots, COLOR(epoch)));
(8)    thread_resume(sweeper(COLOR(epoch-2)));
(9)    barrier_sync (marker, sweeper);
(10)   thread_suspend(mutator);
(11)   while (RS_set) {
(12)          thread_resume(marker(RS_set, COLOR(epoch)));
(13)          RS_set = {};
(14)          thread_resume(mutator(RS_set, COLOR(epoch)));
(15)          barrier_sync(marker);
(16)          thread_suspend(mutator);
(17)   }
(18)   send the garbage_list to sweeper;
(19)   garbage_list = {};
(20)   roots = get_roots(mutator);
(21)   epoch++;
(22) }
```

## 3.1 Overview of the MCGCv0

The first version of the proposed multithreaded concurrent generational garbage collection algorithm (MCGCv0) contains one mutator, one marker and one sweeper as shown in Figure 1. By doing so, the synchronizations among mutators, markers or sweepers can be isolated so that we can focus on synchronizations between mutator and marker, marker and sweeper, and sweeper and mutator. To synchronize the mutator and the marker, a rescanned set (*RS_set*), similar to the store set in the VCGC, is associated with the mutator. The *RS_set* is used to memorize all objects that need to be rescanned due to asynchronous operations caused by the mutator and the marker. The marker and the sweeper require no synchronization in this design. However, the mutator and the sweeper do need synchronous operations on manipulating the system free lists. Moreover, to save the cost from thread creation and deletion, all threads are created as daemon threads (Line 3 and 4 in Figure 1). Once these threads are created, they are suspended and wait for a resume signal sent by *thread_resume()* function call (Line 6, 7, 8, 12, 14 and 18 in Figure 1). This threading mechanism can be done simply by applying a mutex and a condition variable.

The MCGCv0 algorithm works similar to the VCGC algorithm. However, there are a spate of differences between the VCGC algorithm and the MCGCv0 algorithm such as the threading mechanism (as mentioned in previous paragraph), the function of the *RS_set* (called store set in VCGC), the mutator, the marker and the sweeper. Due to the snapshot-at-the-beginning property of the VCGC, it may not collect non-reachable objects produced in an epoch. Although these floating garbage objects will be reclaimed in a later epoch, the size of the memory footprint may be raised drastically. However, to collect them in the same epoch, two issues occur: how to identify garbage and how to mark them without duplicating the marking work. Identifying garbage can be done by applying a reference counting mechanism and examining the reference counter. A 2-bit counter is proposed in the design. Note that in practice, the reference counter is incorporated into the color byte without involving more space overhead and unlikely to be saturated. On the other hand, the marker can recursively identify floating garbage in the same epoch. This reduces the marking overhead in the next epoch. Therefore, in the write barrier procedure, the mutator pushes into *RS_set* not only the new object, but also the old object if it becomes garbage after updating their reference counters. If the old object is not garbage, the marker will mark it eventually. Thus, there is no need to put it into the *RS_set*. Finally, the *RS_set* will be rescanned once the markers and the sweepers finish (Line 9 in Figure 1).

Updating reference counter and putting objects into *RS_set* are part of the mutator thread's duty. It it worth noting that only the mutator can modify the counter and there is only one mutator in the MCGCv0 algorithm. Thus, there is no synchronization problem in updating the reference counter here. However, in a multi-threaded environment, server mutators may update a reference counter simultaneously and create a race condition. Special care for this issue will be elaborated in a later section. On the other hand, the synchronization on the *RS_set* for the mutator (producer) and the marker (consumer) may be avoided either by the asynchronous store set approach [HuW 98] or by submitting the *RS_set* to the marker at a certain point (Line 12 and 13 in Figure 1). Our approach is simple and it does not create time penalty because the mutator has to be suspended in that point in both approaches.

The *RS_set* has a close relation to the young generation in the traditional generational garbage collector. The success of the generational garbage collection is highly based on the program behavior that young objects die young. Thus, we try to collect the young garbage at the same epoch in which they become garbage. However, in the VCGC algorithm, the newly created objects have the mutator color which are regarded as live for at least 2 epochs. Unfortunately, this causes unwanted scanning in a later epoch. To tackle this problem, in our approach, young objects are put into the *RS_set* if they become garbage. By scanning the *RS_set* that contains mostly young objects, the young garbage objects can be detected without scanning the whole heap. Therefore, the heap has been divided into an old generation (objects reachable from the root set) and a young generation (garbage found in the *RS_set*) conceptually. Although a typical generational garbage collection algorithm involves memory space separation into generations and copying, the MCGCv0 algorithm utilizes its advantages in the design.

An epoch is not advanced without the marker completely scanning the *RS_set*. While scanning the *RS_set*, the mutator is still active until the marker catches up with it, i.e., the *RS_set* is empty (Line 11 in Figure 1). Keeping the mutator running can prevent long pause if it updates a lot of objects in an epoch that needs to be rescanned. After the marker finishes marking the *RS_set*, a *garbage_list* is used to keep the garbage. The *garbage_list* is given to the sweeper during the short pause (Line 18 and 19 in Figure 1). Therefore, the sweeper can return the garbage objects in the *garbage_list* to the system right away (Line 8 in Figure 1). Note that there are two types of garbage: the garbage detected through the *RS_set* and the garbage identified by the regular tricolor mechanism. Moreover, the epoch advances after the root set of the mutator has been copied (Line 20 in Figure 1).

## 4. The Proposed Multithreaded Concurrent Generational Garbage Collection Algorithm (MCGC): a Full Version

The proposed multithreaded concurrent generational garbage collection algorithm (MCGC) is a multiple mutator version of the MCGCv0 algorithm. By allowing multiple mutators, the mutator/mutator synchronization issue such as concurrent updating reference count and manipulating the *RS_set* must be resolved. The MCGC algorithm is detailed in Figure 2 where each mutator is associated with an *RS_set* (Line 2 in Figure 2). The *RS_set* is used to keep the log for updating the reference count and the objects whose pointers are modified due to the asynchronous operation of the marker and the mutator. By applying an *RS_set* to a mutator, the mutator/mutator synchronization mentioned above can be eliminated.

Since there are multiple mutators, the store set approach proposed in [HuW 98] may not be applied well because it can not handle concurrent appending of elements. Our approach does not have this problem. However, whether it is better to pass these *RS_set's* to the marker altogether or one by one remains to be studied. The later approach is adopted in the MCGC algorithm (Line 10 - 18 in Figure 2). Its advantage is the mutators need not to be suspended at the same time. Therefore, the marker can examine the *RS_set* one by one (Line 10 in Figure 2). In fact, the examining sequence can be arbitrary. To shorten the mutator pause here, an *RS_set* with larger size has higher priority. Note that the mutator remains suspended after the marker has caught up with it (Line 17 in Figure 2).

**Figure 2 The Proposed Multithreaded Concurrent Generational Garbage Collection Algorithm (MCGC)**

```
(1) int epoch = 2;
(2) root_set_t roots = {}, RS_set[n] = {}, garbage_list = {};
(3) thread_t mutator[n], marker, sweeper;
(4) thread_create_daemon mutator[n], marker, sweeper;
(5) loop forever {
(6)    thread_resume(mutator[n](RS_set[n], COLOR(epoch)));
(7)    thread_resume(marker(roots, COLOR(epoch)));
(8)    thread_resume(sweeper(COLOR(epoch-2)));
(9)    barrier_sync (marker, sweeper);
(10)   for i = 1 to n do
(11)     thread_suspend(mutator[i]);
(12)     while (RS_set[i]) {
(13)       thread_resume(marker(RS_set[i], COLOR(epoch)));
(14)       RS_set[i] = {};
(15)       thread_resume(mutator[i](RS_set[i], COLOR(epoch)));
(16)       barrier_sync(marker);
(17)       thread_suspend(mutator[i]);
(18)     }
(19)   send the garbage_list to sweeper;
(20)   garbage_list = {};
(21)   roots = get_roots(mutator[n]);
(22)   epoch++;
(23) }
```

The marker in the MCGC algorithm plays an important role because it does three things: mark live objects, mark garbage objects and update the reference count. It brings the live object to have mutator color, identifies garbage objects with the assistance of reference count and makes the reference count up to date. There are advantages to updating the reference count information accumulated in the *RS_set*. Firstly, the mutator/mutator synchronization on updating the reference count can be removed nicely. Secondly, the reference count would not be sticky soon because any local references to an object will vanish without overflowing it. However, the marker may not be able to identify a zero-reference-count object as garbage without updating reference counts in all the *RS_set's*. This is true because some mutator may hold a reference to it. Therefore, some objects in the *garbage_list* may be resurrected later. By the time all the *RS_set's* being verified, the *garbage_list* contains true garbage. It is interesting but not a problem in the MCGC algorithm.

## 5. Implementation, Measurements and Results

The proposed multithreaded concurrent generational garbage collection algorithm (MCGC) has been implemented into the Kaffe JVM version 1.0.6 [Kaf 99]. Results are obtained by running the SPECjvm98 benchmark [SPEC 98] on an Intel Pentium-III 650 MHz with Redhat Linux v6.2. All the benchmark programs are measured in their maximal size of 100. The results show that the MCGC achieves up to 1052 speedup over the traditional stop-the-world garbage collection and yields a garbage collection pause of less than 8.8 milliseconds. Moreover, in an epoch, the MCGC algorithm can detect up to 2806 garbage objects (javac) which are considered as floating garbage in the very concurrent garbage collection (VCGC) algorithm.

### 5.1 Benchmarks

Seven Java programs from the SPECjvm98 benchmark suite [SPEC 98] are used to study the performance of the MCGC algorithm. These benchmark programs are designed to measure the performance of Java Virtual Machine implementations. Several criteria such as high byte-code content, flat execution profile (large loops), repeatability, heap usage and allocation rate, and I-cache or D-cache misses on the reference platform are used to test JVMs. Most of the benchmarks contain integer and floating point computations, library calls, or I/O operations; however, SPECjvm98 does not cover *AWT*, networking, or graphics applications. Moreover, only *mtrt* is multi-threaded in the suite.

A thorough study of SPECjvm98 was performed by Dieckmann and Hölzle [DH 99]. In their paper, metrics such as object age, size distribution, type distribution, and object alignment overhead were reported. Furthermore, the system-level performance investigation has been studied and reported in [LSC 00, LSC 02]. The descriptions of benchmark programs from SPECjvm98 are summarized in Table 1.

**Table 1 Descriptions of the SPECjvm98 Benchmarks**

| Program | Description |
|---|---|
| Check | Test various features of the JVM to ensure a suitable environment for Java programs |
| Compress | Compress/decompress program based on modified Lempel-Ziv method. |
| jess | A Java expert shell system based on NASAs CLIPS expert shell system |
| db | Performs multiple database functions on memory resident database |
| javac | The JDK 1.0.2 Java compiler compiling 225,000 lines of code |
| Mpegaudio | An ISO MPEG Layer-3 audio decoder |
| mtrt | A dual-threaded raytracer that works on a scene depicting a dinosaur |

## 5.2 Implementations

Since the implementation of the MCGC is based on the Kaffe GC, its design principle is summarized as follows. The GC system in the Kaffe JVM 1.0.6 is implemented as stop-the-world mark-sweep garbage collection with a segregated list allocation paradigm [WJN 95]. Its memory objects with size larger than a page are called large objects whereas objects with size less than a page are called small objects. A small object request is rounded up to the nearest object size predefined during the virtual machine start-up. The Kaffe Garbage Collection (GC) system keeps a free list for every GC block which contains some particular objects of the same size. Every GC block is aligned to a page. Once the GC block containing the requested object size is found, the free object can be returned immediately by using special indexing techniques. At the same time, the Kaffe GC needs to maintain GC object status by applying the tricolor concept (even though the Kaffe uses more than three colors, we adopt tricolor conceptually). However, the Kaffe GC is a version of the stop-the-world mark-sweep garbage collection. For a large object request, i.e., object size larger than one page, the Kaffe GC will round it up to page boundary and allocate memory size of multiple pages for it, i.e., large objects are handled separately.

The data structures used in the MCGC implementation are defined hereinafter. Firstly, every object has one of the six colors: inuse, free, fixed, sweeper, marker and mutator. All the colors are self-explained except fixed color which is used for a fixed object that does not need to be collected. Secondly, every object is associated with a 2-bit reference count. Note that the colors and the reference counts are encoded into a single byte. Thirdly, there are five lists: *finalise_list*, *sweeper_list*, *marker_list*, *mutator_list*, *makrer_live_list* and *garbage_list*. Each of them is a doubly circular linked list and used to keep objects in different garbage collection phases. The *finalise_list* keeps objects that need to be finalized where the *garbage_list* is used by the marker to collect garbage in an epoch. Furthermore, the *mutator_list*, *marker_list* and *sweeper_list* contain objects whose color is the same as its name, respectively. Fourthly, every page contains a header which is used to store all the information for the block such as the object size, the number of the free objects, the address for the first free object, colors, reference counters, etc. It is worth noting that every page is divided into equal-size objects, i.e., a segregated list mechanism is adopted.

An object is allocated as mutator color and appended to the *mutator_list* (Line 6 in Figure 2). To avoid concurrent appending of objects to the *mutator_list*, each mutator is associated with a separated *mutator_list*. However, objects in the *mutator_list* may be removed by the marker if they become garbage. The synchronization can be reduced by applying a lock on removing the head object in the list. Removing objects is safe because the mutator always inserts objects right after the head object. On the other hand, the mutator also logs the reference count update information and rescanning objects into its own *RS_set*. These *RS_set*'s are then handled by the marker.

The marker starts marking the root set (Line 7 in Figure 2), removes live objects from the *marker_list* and appends live objects to the *marker_live_list*. The *marker_list* and the *marker_live_list* are operated only by the marker and their operations require no synchronization. At the same time, the sweeper is sweeping objects back to the system lists (Line 8 in Figure 2). The sweeper traverses objects in the *sweeper_list* and polls the locks associated with the bins to cooperate with the mutator. Since the *sweeper_list* is only seen by the sweeper, it involves no synchronization. Moreover, the sweeping time would not be too long because the garbage objects are in the *sweeper_list* that avoids scanning the whole heap.

When the marker and the sweeper finish (Line 9 in Figure 2), the mutators are suspended one by one and their *RS_set*'s are examined that including reference count update and rescanning live objects or garbage. The *RS_set* is passed to a local structure of the marker (Line 13 and 14 in Figure 2) while a mutator is suspended. Before resuming the mutator, the *RS_set* is nullified (Line 14 in Figure 2). The marker then marks objects on the *RS_set* and puts live ones into the mark_live_list and garbage ones into the *garbage_list* where the local *RS_set*, the mark_live_list and the *garbage_list* are used only by the marker and require no synchronization. Eventually, this mutator is suspended until its *RS_set* is empty and the epoch ends when all the *RS_set*'s are empty. It is worth noting that the *RS_set* can be simply implemented as single word structure where the 2 least significant bits are used to store the reference count update information due to the 4-byte memory alignment.

During the examination of the *RS_set*'s (Line 10 - 18 in Figure 2), a garbage object in the *garbage_list* may be resurrected because of the asynchronous updating of the reference count. However, this is not a problem because the marker can remove the garbage from the *garbage_list* and append it back to the *marker_live_list* without any synchronization. Once the whole process finishes, the *garbage_list* is appended to the sweeper and the *garbage_list* is nullified (Line 19 and 20 in Figure 2). Remark that all the mutators are suspended now. Finally, the root sets are copied in preparation for the next epoch.

A write barrier routine is implemented to record the new and old references if the mutator updates the object reference graph. The write barrier is called by the Java bytecode instructions: PUTFIELD, PUTSTATIC and AASTORE. The PUTFIELD instruction is used to set a field in an object. When the field is reference type, the write barrier is called. The case of the PUTSTATIC is similar to the PUTFIELD but a static field in a class is written. The AASTORE instruction stores into a reference array and thus the write barrier is invoked.

## 5.3 Results

The results shown in the section are collected by running the SPECjvm98 benchmark on an Intel Pentium-III 650 MHz with the RedHat Linux 6.2 and the Kaffe JVM with the MCGC or the stop-the-world mark-sweep garbage collector. Table 2 shows the maximal number of floating garbage objects found by examining the *RS_set* among all epochs and sticky reference count statistics. The result shows that up to 7251 floating garbage objects (mtrt) can be detected in an epoch. In other words, these floating garbage objects cannot be collected in a timely manner if the VCGC algorithm is applied. Therefore, the memory footprint may increase. This shows the high efficiency of the MCGC algorithm over the VCGC algorithm. On the other hand, the rate of the sticky reference count is less 16.09% (mtrt); most of the sticky rates are less than 6.63% which show that 2-bit reference count is sufficient.

**Table 2 Maximal Number of Garbage Objects Found and Sticky RC in the MCGC**

| Benchmark | Garbage Objects | Sticky Reference Count | Total Allocated Objects | Sticky Rate (%) |
|---|---|---|---|---|
| check | 112 | 1,017 | 41,952 | 2.42 |
| compress | 113 | 928 | 37,658 | 2.46 |
| jess | 1,498 | 4,358 | 110,595 | 3.94 |
| db | 159 | 11,171 | 168,562 | 6.63 |
| javac | 1,447 | 5,132 | 139,677 | 3.67 |
| mpegaudio | 113 | 968 | 45,674 | 2.12 |
| mtrt | 7,251 | 157,162 | 976,511 | 16.09 |

Due to its usage of reference count mechanism, the MCGC may have a time penalty in the total execution time. To our surprise, the penalty of most of the benchmarks are shorter than 7.78% as shown in Table 3. Moreover, for the mtrt benchmark, the MCGC improves 99.36% in total execution cycles. The outcome confirms the fact that the multithreaded program is better than the non-multithreaded program even in a single processor system.

**Table 3 Memory Footprint and Total Execution Time**

| Benchmark | Memory Footprint (Bytes) | | | Total Execution Time (Cycles) | | |
|---|---|---|---|---|---|---|
| | MCGC | Stop-the-World GC | % Improvement | MCGC | Stop-the-World GC | % Improvement |
| check | 1,564,680 | 1,642,504 | 4.97 | 455,758,284 | 449,763,031 | -1.33 |
| compress | 9,129,992 | 9,195,528 | 0.72 | 104,049,298,977 | 102,117,023,959 | -1.89 |
| jess | 4,969,384 | 5,043,112 | 1.48 | 2,488,729,893 | 2,309,055,573 | -7.78 |
| db | 10,968,000 | 8,391,616 | -23.49 | 10,604,173,496 | 10,188,447,970 | -4.08 |
| javac | 3,665,928 | 3,678,216 | 0.34 | 2,080,763,023 | 2,054,940,275 | -1.26 |
| mpegaudio | 1,974,280 | 2,060,528 | 4.37 | 93,125,493,303 | 92,337,776,597 | -0.85 |
| mtrt | 21,056,464 | 16,785,360 | -20.28 | 302,244,851 | 47,419,200,739 | 99.36 |

On the other hand, Table 3 also compares the memory footprint. Although memory is presently inexpensive, the MCGC with its ability to collect floating garbage does improve the memory footprint in most of the cases ranging from 0.72% to 4.97%. However, some cases such as db and mtrt may have negative effects. Different execution rates for the mutators and the garbage collector are the major reason. We believe that the memory footprint can be controlled by carefully prioritizing the mutator and the collector threads.

Although the MCGC runs its marker and sweeper threads with mutators concurrently, there are situations that must be synchronized. For example, the mutators are suspended when their contexts are examined at the end of an epoch. These transient garbage collection pauses are recorded and summarized in Table 4. The performance of the MCGC is compared with the stop-the-world mark-sweep GC through the maximal and average garbage collection pause. The results shows that the MCGC performs much better than the stop-the-world mark-sweep GC. The maximal garbage collection pause ranges from 527,611 machine cycles to 43,870,810 machine cycles. In all the benchmarks, the maximal garbage collection pause for the MCGC is improved by 79.45% - 96.75% and its average garbage collection pause is improved by 88.32% - 95.66%.

**Table 4 Garbage Collection Pause Comparison**

| Benchmark | Maximal Garbage Pause Performance (Cycles) | | | Average Garbage Pause Performance (Cycles) | | |
|---|---|---|---|---|---|---|
| | MCGC | Stop-the-World GC | % Improvement | MCGC | Stop-the-World GC | % Improvement |
| check | 527,611 | 3,618,717 | 85.42 | 285,884 | 2,535,443 | 88.72 |
| compress | 611,365 | 3,469,500 | 82.38 | 328,090 | 2,809,855 | 88.32 |
| jess | 674,494 | 20,727,866 | 96.75 | 317,980 | 7,322,160 | 95.66 |
| db | 15,507,740 | 75,473,024 | 79.45 | 3,269,434 | 34,671,346 | 90.57 |
| javac | 1,125,079 | 17,108,962 | 93.42 | 470,283 | 7,739,970 | 93.92 |
| mpegaudio | 619,202 | 5,114,325 | 87.89 | 295,993 | 3,315,065 | 91.07 |
| mtrt | 43,870,810 | 238,256,339 | 81.59 | 8,843,814 | 96,835,940 | 90.87 |

## 6. Conclusions

Currently, much effort has been spent on concurrent garbage collection such as concurrent generational garbage collector, very concurrent mark-sweep garbage collection (VCGC) and hardware assisted garbage collectors. The ultimate goal is to reduce the pause time while the garbage collector is collecting garbage. In this paper, a multithreaded concurrent generational garbage collection (MCGC) algorithm has been proposed. The MCGC algorithm can take advantage of multiple CPUs in a SMP system or the merits of light-weight processes. Moreover, it can be incorporated into hardware garbage collection systems such as the modified buddy system [ChG 96, CSL 99] and other garbage collectors that require identifying live objects such as the hardware-assisted real-time garbage collector [Nis 94].

The MCGC algorithm has been implemented into the Kaffe JVM version 1.0.6. The SPECjvm98 benchmark suite is used to measure the performance of the MCGC algorithm. Performance evaluation is conducted on a Pentium-III 650 MHz with 128 MB memory running RedHat Linux 6.2. Results show that the MCGC outperforms the VCGC where up to 2806 garbage objects (javac) can be detected in an epoch by examining the *RS_set*. This shows the high efficiency of the MCGC over the VCGC. Additionally, the MCGC performs much better than the traditional stop-the-world mark-sweep garbage collector. The results show that the maximal garbage collection pause for the MCGC is improved up to 96.75% and its average garbage collection pause is improved up to 95.66%.

The contributions of this work are fivefold. Firstly, the proposed MCGC algorithm enhances the merits of the mark-sweep algorithm, the reference counting approach and the generational collection. Secondly, it requires no explicit synchronization between the mutators and the marker, between the mutators and the sweeper or between the marker and the sweeper. Thirdly, the new instruction can be called by several mutators concurrently. Fourthly, it has been implemented into the Kaffe JVM which improves the maximal garbage collection pause up to 96.75% and up to 95.66% in average. Finally, the MCGC receives minimal time and space penalties in terms of the the total execution time, the memory footprint and the sticky reference count rate.

## 7. References

[BR 01] David F. Bacon and V.T. Rajan, “Concurrent Cycle Collection in Reference Counted Systems”, *Proc. European Conference on Object-Oriented Programming*, LNCS vol. 2072, June 2001.

[ChG 96] J. M. Chang and E. F. Gehringer, “A High-Performance Memory Allocator for Object-Oriented Systems,” *IEEE Transactions on Computers*. March, 1996. pp. 357-366.

[CSL 99] J. M. Chang, W Srisa-an, and C.D. Lo, "Introduction to DMMX (Dynamic Memory Management Extension)", *Proceeding of ICCD Workshop on Hardware Support for Objects and Microarchitectures for Java*, Austin, TX. October 10, 1999.

[DeT 90] John DeTreville, “Experience with concurrent garbage collectors for Modula-2+”, Technical Report 64, DEC Systems Research Center, Palo Alto, CA, August 1990.

[DH 99] S. Dieckmann and U. Hölzle, “A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks”, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'99), Lecture Notes on Computer Science*, Springer Verlag, Lisbon, Portugal, June 1999.

[DKP 00] Domani, T., Kolodner, E. K., and Petrank, E., “A generational on-the-fly garbage collector for Java,” In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (June 2000). *SIGPLAN Notices*, 35, 6, 274-284

[DL 93] D. Doligez and X. Leroy, “A Concurrent, generational garbage collector for a multithreaded implementation of ML”, *Papers of the 20th ACM Symposium on Principles of Programming Languages*, January 11-13, 1993.

[DLM 76] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. “On-the-fly garbage collection: An exercise in cooperation.” In *Lecture Notes in Computer Science*, No. 46. Springer-Verlag, 1976.

[Dom 00] Domani, T., et al., “Implementing an on-the-fly garbage collector for Java,” In *Proceedings for the ACM SIGPLAN International Symposium on Memory Management* (Minneapolis, MN, Oct. 2000). *SIGPLAN Notices*, 36, 1(Jan., 2001), 155-166

[HuW 98] L. Huelsbergen and P. Winterbottom, “Very Concurrent Mark-&-Sweep Garbage Collection without Fine-grain Synchronization,” *Proceedings of the Int. Symposium on Memory Management*, pp. 166-175, October 1998.

[JoL 98] R. Jones, R. Lins, *Garbage Collection: Algorithms for automatic Dynamic Memory Management,* John Wiley and Sons, 1998

[Kaf 99] Kaffe version 1.0.6 released under GPL license by GNU. http://www.kaffe.org

[LP 01] Yossi Levanoni and Erez Petrank, “An on-the-fly Reference Counting Garbage Collector for Java”, ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA, Florida, USA, October 14-18, 2001

[LSC 00] C. D. Lo, W. Srisa-an, J. M. Chang, "Page Replacement Performance in Garbage Collection Systems", in the *Proceedings of 13th International Conference on Parallel and Distributed*

*Computing Systems*, Las Vegas, Nevada, August 8-10, 2000.

[LSC 02] C. D. Lo, W. Srisa-an and J. M. Chang, "A Study of Page Replacement Performance in Garbage Collection Heap", accepted for publication in *The Journal of Systems and Software*, 2002

[NiS 94] K.D. Nilsen and W.J. Schmidt, “A High Performance Hardware-Assisted Real-Time Garbage Collection System,” *Journal of Programming Languages*, 1994. 2(1): p. 1 - 40.

[Rov 85] Paul Rovner, “On adding garbage collection and runtime types to a strongly-typed, statically-checked, concurrent language”, Technical Report CSL-84-7, Xerox PARC, Palo Alto, CA, July 1985.

[SPEC 98] Standard Performacne Evaluation Corporation. SPECjvm98 Documentation, Release 1.0. August 1998. Online version at http://www.spec.org/osg/jvm98/jvm98/doc/index.html.

[WJN 95] Paul Wilson, M. Johnstone, M Neely and D. Boles, “Dynamic Storage Allocation: A Survey and Critical Review”, Proc. 1995 *Int’l workshop on Memory Management*, Scotland, UK, Sept. 27-29, 1995.