

# A Multithreaded Concurrent Garbage Collector Parallelizing the New<sup>1</sup> Instruction in Java<sup>2</sup>

Chia-Tien Dan Lo, Witawas Srisa-an and J. Morris Chang

Department of Computer Science  
Illinois Institute of Technology  
Chicago, IL, 60616-3793, USA  
{danlo, witty, chang}@charlie.iit.edu

## Abstract

*Parallel, multithreaded Java applications such as web servers, database servers, and scientific applications are becoming increasingly prevalent. Most of them have high object instantiation rates through the new bytecode that is implemented in a garbage collection subsystem typically. For aforementioned applications, traditional garbage collectors are often the bottleneck that limits program performance and processor utilization on multiprocessor systems. They 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.*

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

## 1. Introduction

While a battery of concurrent garbage collectors have been proposed in the literature [1, 4, 5, 6, 7, 8, 9, 10, 11, 17], very few of them have been implemented and especially been incorporated into Java virtual machines (JVMs) [1, 5, 8, 11]. The first concurrent garbage collector using the tricolor abstraction (white, grey, black) is proposed by Dijkstra, et al. [7]. 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.

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 [9]. 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.

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] and the on-the-fly reference counting garbage collector proposed by Levanoni and Petrank [11]. On the other hand, the generational on-the-fly garbage collector proposed by Domani et al. [8, 5] is based on mark-sweep. Although both the Recycler [1] and the on-the-fly reference counting (RC) garbage collector [11] 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

---

1. The new instruction is a Java bytecode instruction which allocates dynamic memory for objects.

2. Partial support for this work is provided by the National Science Foundation through grants CCR-0098235 and CCR-0113335.

collectors, the generational on-the-fly garbage collector [8, 5] is a generational garbage collector based on the work of Doligez and Leroy [6]. 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. On the other hand, Lo et al. reports performance analyses on generalized buddy systems [12] and page replacement performance on garbage collection heap [13].

In this paper, a multithreaded concurrent generational garbage collection (MCGC) algorithm is proposed [14]. 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. A performance study of the MCGC is published in [15] due to space limit.

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

The very concurrent garbage collection was proposed by Huelsbergen and Winterbottom in 1998 [9]. 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 con-

tent of references prior to the mutator updates them. In [18], 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 [7] 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 because the store set is designed for one mutator and one marker situation.
- 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 liveness 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,

**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) }
```

the memory footprint can be increased.

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.

#### 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 iso-

lated 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 refer-

ence 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 is 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 multithreaded 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 [9] 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 [9] 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).

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

**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) }

```

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.

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.

## 5. 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 [2, 3] and other garbage collectors that require identifying live objects such as the hardware-assisted real-time garbage collector [16].

The contributions of this work are threefold. First, the proposed MCGC algorithm enhances the merits of the mark-sweep algorithm, the reference counting approach and the generational collection. Second, it requires no explicit synchronization between the mutators and the marker, between the mutators and the sweeper or between the marker and the sweeper. Third, the new instruction can be called by several mutators concurrently.

## 6. References

- [1] 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.
- [2] 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.
- [3] 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.
- [4] John DeTreville, "Experience with concurrent garbage collectors for Modula-2+", Technical Report 64, DEC Systems Research Center, Palo Alto, CA, August 1990.
- [5] 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
- [6] 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.
- [7] Edsger 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.
- [8] 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
- [9] 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.
- [10] R. Jones, R. Lins, *Garbage Collection: Algorithms for automatic Dynamic Memory Management*, John Wiley and Sons, 1998
- [11] 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
- [12] C. D. Lo, W. Srisa-an and J. M. Chang, "Performance Analysis on the Generalized Buddy System," In *IEE Proceedings, Computers and Digital Techniques*, Vol. 148, No. 4/5, July/September 2001, pp. 167-175.
- [13] C. D. Lo, W. Srisa-an and J. M. Chang, "A Study of Page Replacement Performance in Garbage Collection Heap," *The Journal of Systems and Software*, vol. 58, 2001, pp. 235-245.
- [14] C. D. Lo and J. M. Chang, "A Multithreaded Concurrent Generational Garbage Collector for Java," ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2001 (Doctoral Symposium), Tampa Bay, Florida, USA, Oct. 14-18, 2001, pp. 7-9.
- [15] C. D. Lo, W. Srisa-an and J. M. Chang, "A Performance Comparison between Stop-the-World and Multithreaded Concurrent Garbage Collection for Java," In 21st IEEE International Performance, Computing, And Communications Conference (IPCCC 2002), Phoenix, Arizona, April 3-5, 2002.
- [16] 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.
- [17] 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.
- [18] 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.