Modern Considerations of Garbage Collection in the Java HotSpot Virtual Machine

Jeffrey D. Lindblom
lindb310@morris.umn.edu

ABSTRACT

This paper describes garbage collection (GC) from the ground up, with a detailed comparison of various GC techniques and algorithms to date. In this effort we will explore how GC has evolved to what we see in the Java HotSpot™ Virtual Machine today, and discuss the modern performance challenges it faces now and in the future.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection), Optimization, Run-time environments*; D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages, Java*

General Terms

Algorithms, Performance, Design, Reliability, Languages

Keywords

Garbage Collection, GC, Java Virtual Machine, Java, HotSpot

1. INTRODUCTION

Within the field of computer science, *programming* is the process by which computer software, or programs, are engineered. This process is undertaken via a *programming language*, a syntactic set of definitions and rules that allows programmers to symbolically express software functionality. Just as there are many different languages for communication, there are many different programming languages for programming. The programming language we will be focusing on within this paper is Java, which is known to be an *object-oriented* programming language [17].

Object-oriented programming languages are considered as such because of how they represent data. Within an object-oriented programming language, data is represented and related through *objects*. E.g. if we were to program the game of solitaire, we might want to create a card object to hold each card's suit and number, and a deck object to hold all of the card objects. Programming in this fashion can help to preserve data relationships and ideas in an explicit manner.

While objects can help model data in useful ways, they do incur a cost on the computer system. This is realized in the form of a space usage known as *memory allocation*. For each object defined within Java, computer memory has to be allocated to it for it to exist. While in some programming languages we have to allocate this memory manually, such as in C, others, like Java, allocate it automatically [17]. Memory is not infinite however, and non-trivial software may instantiate substantial quantities of objects. This makes it important to maximize the amount of memory we have available, not only to maintain free space, but also to lessen software impact on a computer system.

One way of achieving memory maximization is by ensuring that objects that are active within a program take preference over ones that are not. These inactive objects are described as *dead objects*, and are characterized by being unreachable within the program. An object is considered reachable when it has the possibility of being used by the program presently or in the future [12].

Taking care to identify and eliminate dead objects is an ever-present problem in object-oriented programming. For memory-managed languages like Java, we rely on a system of interpretive, algorithmic strategies known as *garbage collection* (GC) to mind the process for us. This is achieved through what we call a *garbage collection cycle*, an iterative loop of identification and freeing of dead objects within an actively running program. The freeing of these dead objects is referred to as *collection*, the reclamation of memory [14].

2. FUNDAMENTALS OF GC

Garbage collection (GC) originated from two very distinct algorithms, *reference counting* (RC) and *Mark-and-Sweep*, which were both fully developed by 1960. Since then, RC and Mark-and-Sweep have dominated the field of GC. Various implementations of each have made their mark in many different memory managed languages over the years, and they presently continue to be optimized for increasingly complex programs and environments [3] [10].

2.1 Reference Counting vs Mark-and-Sweep

Fundamentally, a RC garbage collector is just as it sounds. It tracks a given object's reachability by counting the number of *references* to it from other objects. An object reference is an object's active association with another object, e.g. a dealer object dealing a card object. When a given ob-

This work is licensed under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/us/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.
UMM CSci Senior Seminar Conference Morris, MN.

ject's reference count drops to zero, the object is deemed unreachable and is subsequently freed. This simple behaviour gives RC the advantage of clarity and understanding. It is easy to follow its motions and predict how it will act. Unfortunately though, such a basic approach allows for *self-referencing objects* to slip through.

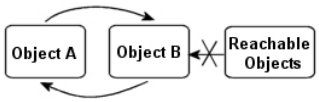

Figure 1: Unreachable Self-referencing Object

Self-referencing objects are objects that refer to themselves either by a direct or indirect means. An example would be a *doubly-linked list* with two objects. A doubly-linked list is a list of objects that are linked together in a bi-directional fashion, i.e. object A points to object B and object B points to object A. In this sense object B refers to A and object A refers to B. If the doubly-linked list were ever to become unreachable, the references between object A and object B would still exist, and as such their reference count would not be zero. This illuminates the reference count loophole that self-referencing objects fall through. Implementations of RC exist to compensate for this, but they incur costs of memory overhead and an increase in algorithmic complexity [10].

A Mark-and-Sweep garbage collector works much the opposite to RC. It instead identifies which objects are reachable first, and then discards whatever objects remain. This operation is performed in two steps, the *mark phase* and the *sweep phase*. The mark phase works to identify reachable objects through their relations to other objects. It does this by assuming reachability through a transitive means, i.e. any object that is referenced from a reachable object is itself reachable. A set of top-level objects are chosen as *roots* in this process, and these roots provide the algorithm a starting point from which to trickle down through references and mark which objects are reachable. Once all reachable object references have been exhausted, the sweep phase is initiated. In this phase all objects within memory are queried, and any unmarked objects discovered are collected [18] [16].

Algorithm 2.1: MARK-AND-SWEEP()

```
procedure markAndSweep(objects)
 roots ⊆ objects
 for each root ∈ roots
   do mark(root)
 sweep(objects)
procedure mark(object)
 if !object.marked?
   object.marked? ← true
 for each referencedObject ∈ object.references
   do mark(referencedObject)
procedure sweep(objects)
 for each object ∈ objects
   if object.marked?
     object.marked? ← false
   else
     objects.release(object)
```

Unlike RC, Mark-and-Sweep is also capable of collecting self-referencing objects. This is due to its emphasis on transitive reachability, rather than reference counts outright. A draw-back of this behaviour though, is that exhausting references to identify what is reachable can take a relatively long time to complete. This type of behaviour is space inefficient and can generate long, distributed interrupts within a program, especially if there is a lot of object relations to traverse. Dead objects also have to wait for the sweep phase to complete before they can be released, which results in longer delays between objects becoming unreachable and actually being collected.

In contrast, RC is an incremental form of GC. It has a very quick dead object collection rate. This is because objects that fit the collection criteria are eliminated immediately after reference counts are updated, and reference counts are updated accordingly with each program computation. In other words, objects are reclaimed as soon as they are determined unreachable, providing for short, incremental GC interrupts. For Java programs that lose effectiveness from long interruptions, a garbage collector like RC is ideal. On the down-side though, continuous updating of reference counts can also generate inefficiency. With RC cycles occurring so frequently, many RC cycles could terminate without any dead objects being freed [3].

2.2 Stop-and-Copy Collection

Stop-and-Copy collection is a more memory intensive variation of the *Mark-and-Sweep* algorithm. It similarly defines a set of roots, of which it identifies reachable objects through transitive association. The difference is that Stop-and-Copy does not perform a sweep to collect dead objects. Instead it relies on a copy then release strategy that utilizes predefined spaces within memory [1].

In its most basic form, two spaces are defined for the Stop-and-Copy algorithm; one labelled the *to space*, and the other labelled the *from space*. When a program is initialized, objects are allocated into the 'to space' until it fills up. Once it reaches capacity, a collection cycle is triggered and the program's execution is halted. At the start of each Stop-and-Copy collection cycle, the 'to space' and the 'from space' switch roles. The space that reaches capacity becomes the new 'from space', while the remaining unoccupied space becomes the 'to space' [18].

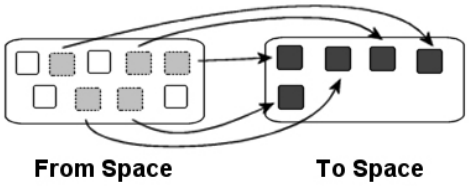

Figure 2: Stop-and-Copy Collection

Once a collection starts, objects reachable from the root set are traversed through their references through a recursive copy operation. The copy operation begins by confirming that the object has not already been copied to the 'to space', as overlaps can often occur. It then copies the object to the 'to space', leaving a forwarding reference to the copied object within the 'from space' as to indicate it has been copied. Last, it calls the copy operation on all objects referenced by the recently copied object [8]. The pseudocode for this

procedure is defined below.

Algorithm 2.2: STOP-AND-COPY()

```
fromSpace ≡ Objects
toSpace ≡ ∅

procedure STOPANDCOPY()
 roots ⊆ fromSpace
 for each root ∈ roots
   do COPY(root)
 RELEASE(objects ∈ fromSpace)
 SWAP(fromSpace, toSpace)
procedure COPY(object)
 if !object.forwarded?
   toSpace.add(object)
   object.forwarded? ← true
   for each referencedObject ∈ object.references
     do COPY(referencedObject)
```

As can be seen, the Stop-and-Copy collector relies on the copy operation to perform the marking of reachable objects. Once this operation finishes, all reachable objects can now be assumed to exist within the 'to space'. Any remaining objects in the 'from space' are then assumed dead and subsequently collected, ending the cycle. Program execution now resumes, and new objects are allocated to the 'to space' until capacity is reached again.

One of the more important observations of the Stop-and-Copy algorithm is that the work it performs is proportional to the quantity of reachable objects within a program. In other words, this algorithm will perform significantly faster on programs with smaller quantities of reachable objects. A disadvantage though, is that the algorithm necessitates a large quantity of available memory in order to perform its copying procedure. This is the major trade-off it has versus marking, as defined in the Mark-and-Sweep algorithm [1] [18].

2.3 Performance and Optimization

The key to GC optimization, and consequently program efficiency, is balance. This ties into a GC concept known as *throughput*, the ratio of program run-time spent working as opposed to garbage collecting. Mark-and-Sweep, for example, has a higher throughput than reference counting because its collection cycle occurs less frequently [14].

Different programs have different dynamics, and as such require different configurations of GC. The objective here is not to maximize throughput of a running program, but to locate an optimum between collecting too much and collecting too little. GC that collects frequently, while effective at freeing dead objects quickly, has the disadvantage of incurring unnecessary impacts on performance when there are few to none dead objects to collect. GC that collects infrequently however also runs the risk of developing a significant memory overhead from dead objects left uncollected for longer periods of time. This can cause a running program to leave a bigger footprint on the computer system, negatively impacting the program's performance [4].

GC today employs a multitude of strategies to overcome these performance challenges. We will look at approaches used within the *Java HotSpot Virtual Machine* (HotSpot JVM), a software process that interprets and executes Java programs on a computer system [5].

3. GENERATIONAL GC

When considering the life-span of objects in memory, three possible scenarios can be assumed:

(1) The object becomes dead shortly after allocation.

(2) The object becomes dead long after allocation.

(3) The object never becomes dead after allocation.

It is these three scenarios that shape how *generational garbage collection* operates, named as such due to its emphasis on *generations* of objects.

Generations, as the name implies, are sets of objects with similar life-spans. The generational garbage collector works by dividing objects up into several of these sets, the most common configuration involving just two sets, young and old. The distinction between the two are important, as their differences lead into what is known as the *weak generational hypothesis*.

This hypothesis is described by two basic observations: (i) most allocated objects become dead shortly after allocation, (ii) there exist few references from older objects to younger objects. This means that younger generations of objects benefit from GC algorithms that have frequent collection cycles, as the frequency of dead objects being available to collect is much higher. Accordingly, younger generations typically contain a small number of active objects, and as such do not require GC algorithms that are space efficient [12].

Older generations are built up of objects that have been *tenured*, meaning they have survived enough younger generation collections to be promoted. These generations usually contain a large amount of active objects, and are slow to grow due to the large number of short-lived objects in younger generations. Given these attributes, collection cycles on old generations take a relatively long time to complete [13].

What makes these insights particularly useful, is the fact that generational GC allows for varied types of collection algorithms to be utilized upon different generations of objects. The HotSpot JVM takes full advantage of this fact to prioritize collection algorithms toward generations of objects matching their best-case scenarios [12].

3.1 Generational GC in the HotSpot JVM

In the HotSpot JVM, object memory allocation occurs in a space construct known as the *heap*. This is where generational GC in the JVM operates, dividing up the heap into two sets: the *young generation* and the *old generation*. The vast majority of Java objects are initially allocated to the young generation [12].

The young generation consists of three spaces, *Eden* and two equally sized *survivor spaces*. Eden is where objects are initially allocated. Once Eden reaches capacity, a collection cycle know as *minor garbage collection* is triggered, which collects dead objects within the young generation and allows for surviving objects to be tenured to the old generation. Before and after minor garbage collection, at least one survivor space is always left unoccupied. This survivor space is commonly referred to as the *to* survivor space, while the other one is known to be called the *from* survivor space [14].

When a minor garbage collection occurs, the Stop-and-Copy algorithm is performed. The reasoning for this ties

back into the weak generational hypothesis we defined above. If observation (i) holds, most objects allocated to the young generation will die quickly. This means the collector will typically encounter high volumes of dead objects, and low volumes of reachable objects. As was mentioned earlier, Stop-and-Copy collection work is proportional to the quantity of reachable objects. This gives us a performance advantage over other collection algorithms, as Stop-and-Copy performance is negligibly effected by the quantity of dead objects to collect. This contrasts with algorithms like Mark-and-Sweep, which has to traverse all objects within memory in both its mark and sweep phases.

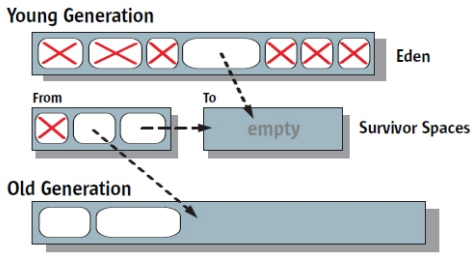

Figure 3: Minor Garbage Collection [12]

During the Stop-and-Copy collection, the reachable objects within Eden are moved to the 'to' survivor space, while objects that are too big to fit within the 'to' survivor space are tenured to the old generation. Next, reachable objects within the 'from' survivor space are moved to the 'to' survivor space, similarly tenuring objects that are too large. By this point, the Eden space is empty, and the survivor spaces will have swapped roles. The now occupied survivor space becomes the new 'from' survivor space, switching roles with its counterpart. Objects moved from the 'from' survivor space to the 'to' survivor space during a minor garbage collection each receive a count. This count serves to enumerate the number of times an object has survived a minor garbage collection. Once an object's count reaches a predefined threshold, it is tenured to the old generation.

The old generation, consisting of long-lived objects, aggregates slowly over time into a sizeable chunk of the heap. Once it fills up, a collection cycle known as *major garbage collection* is triggered. During this cycle, the *Mark-Sweep-Compact* collection algorithm is performed on all objects within the old generation.

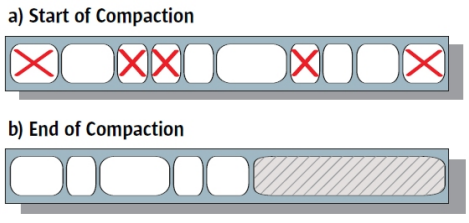

Figure 4: Sliding Compaction [12]

Mark-Sweep-Compact collection is very similar to Mark-and-Sweep collection. The initial behaviour is the same, objects reachable from the roots are traversed and marked, while unmarked objects are collected during the sweep phase. The only distinction is what occurs after the sweep phase has completed. A process known as *sliding compaction* is initiated, which simply slides the surviving objects within the old generation to the beginning of their respective generations. This allows for fast allocation of objects to the old generation, because instead of having to keep track of multiple distributed empty-spaces within the generation space we can just append the object to the end of the last allocated object within a generation.

As might be guessed, major garbage collection occurs relatively infrequently. The reasoning for this set-up can be found in the observations of the weak garbage collection hypothesis outlined above. If we are willing to assume that the vast majority of objects die young, it is reasonable to focus JVM resources into collecting objects within the young generation [14].

4. PARALLEL GC IN THE HOTSPOT JVM

The most basic application of generational GC within the HotSpot JVM is the *serial collector*. The serial collector performs GC in a *stop-the-world* fashion, meaning that Java programs are halted when collection is taking place. This halt can last for a relatively long period of time because the serial collector is developed to work on only one *central processing unit* (CPU), which is where Java program computations are scheduled for processing [12]. This type of processing is quickly becoming obsolete as we begin to rely more and more on *parallel systems* for running programs.

A parallel system is one that embodies multiple central processing units for which to run and schedule computer processes. As hardware becomes cheaper, parallel system architectures become increasingly wide-spread and complex. The advantages of such systems are the maximization of multi-tasking behaviour, which presents some increasingly interesting and complex scheduling challenges in terms of program execution and process management. One of these challenges is how to best utilize parallel environments in the optimization of GC.

4.1 The Throughput Collector

The most basic parallel processing collector in the HotSpot JVM is the *throughput collector*, also known as the *parallel collector*. Its functionality is similar in concept to the serial collector, but ran in a parallel fashion. The young generation set-up is the same, we have an Eden space along with two survivor spaces. The difference lies in how minor garbage collection is performed. We now use a parallel version of the Stop-and-Copy collector to collect on the young generation [5] [12].

Implementing Stop-and-Copy to work in parallel is actually not too difficult. We rely on an artifact known as *threads* to take care of the parallel behaviour for us. Threads are spawned processes that work separate to the process that spawned it. They are scheduled independent of their parent process, which allows them to be processed on separate CPUs. This approach ties into the the goal of parallel processing GC, which is to utilize as many scheduling constructs as possible.

Parallel Stop-and-Copy starts the same way as normal Stop-and-Copy, objects are initialized into the 'to-space' until it becomes full. Once a collection cycle is initiated, the 'to space' and 'from space' roles are switched and the algorithm begins to copy reachable objects from the newly defined 'from space' to the newly defined 'to space'. This is

where parallel processing kicks in, every copy operation performed on objects within the 'from space' is spawned into its own thread. This allows for objects to be traversed and copied to the 'to space' in parallel, greatly reducing the time taken to copy all reachable objects.

One observation that is important to note here though, is that there now runs the possibility of one object having a copy operation performed on it by two or more threads at the same time. This can lead to erroneous results, so the parallel Stop-and-Copy algorithm must take care to ensure that an object can only be processed by one copy thread at a time. The process by which this is done is called *thread synchronization*, and is one of the additional overheads parallel GC can incur [11].

What distinguishes the throughput collector from other parallel collectors within the HotSpot JVM, is that it does not use parallel processing to collect the old generation. For most Java programs, this is generally acceptable, because the old generation usually has very few dead objects to collect. Parallel processing makes use of multiple CPUs, so it tends to increase the footprint of Java program processing on a computer system. This makes it important to utilize parallel processing only when the potential benefits outweigh the costs. Some examples of situations where this might be true are; Java programs that require shorter pauses from GC, Java program execution on computer systems that would be negligibly effected by the parallel processing, or expectations of large amounts of dead objects to collect. For situations that may benefit from parallel old generation collection, the HotSpot JVM currently provides three options as of Java SE 7: the *parallel compacting* collector, the *concurrent mark sweep* (CMS) collector, and the *garbage first* collector [5] [12].

4.2 CMS versus Parallel Compaction

The CMS collector and the parallel compacting collector are both generational and collect the young generation using the same process as the throughput collector. The main difference between the two, is that the CMS collector operates in a *concurrent* fashion, while the PC collector does not [5] [12]. What this means is that the CMS collector performs most of its collection cycle in parallel to Java program computation. This allows for the Java program to have higher throughput than the parallel compaction collector, but instead suffers higher latency.

Latency in GC is the measurement by which a garbage collector negatively affects program processing performance. The CMS collector has relatively high latency because it performs most of its collection cycle in parallel to the Java program's computation. This implies that CPUs used for processing the Java program will also be shared with the garbage collector, taking away the potential performance advantages those extra scheduling cycles might yield. If higher throughput is a reasonable trade-off for this constraint, then CMS is your garbage collector of choice.

The CMS collector itself uses a parallel implementation of Mark-and-Sweep, while the parallel compacting collector uses the *Mark-Summary-Compact* algorithm. The Mark-Summary-Compact algorithm is similar to the Mark-Sweep-Compact algorithm, except that it focuses on regions rather than objects. The data for the regions that live objects are located in is stored during the mark phase, and dead objects are overwritten with live objects pushed to the left during the summary and compact phase. This leaves the old generation with one chunk of live data on the left and the remaining unallocated space on the right [15].

5. REAL-TIME GC

In object-oriented languages like Java and C#, *real-time applications* are becoming an increasingly tangible performance challenge in the field of computer science. A real-time application is a program that incorporates tasks with strict computational deadlines to meet. Task failure to meet those deadlines can result in erroneous behaviour by the program, and irrevocably skewed results. As such, it is important for *real-time garbage collection* (RTGC) to work incrementally, interleaving itself within the expected computations of the real-time applications [9]. Some examples of fields that require a real-time emphasis are military command-and-control operations, financial trading systems such as the stock market, and on-the-fly audio processing [7].

A *real-time garbage collector* attempts to operate in a fashion that effectively bounds its space and time overheads. Ideally, if RTGC can maintain predictability and incremental behaviour, real-time Java applications can more safely run within those defined constraints [2] [9]. Currently, there are three defined criteria for effective real-time computing: *hard*, *firm*, and *soft*. Applications that fit under the hard real-time bound are very fragile. Missing just one task deadline means total application failure. Obviously, this is the most difficult bound to meet with GC. Firm real-time applications can tolerate infrequent deadline misses, but any such miss nullifies the results associated with the task. Soft real-time applications are the low-end criteria. Deadline misses do not necessarily nullify task results, but the usefulness of said results are seriously degraded proportional to how much the deadline was missed by.

5.1 The Garbage First Collector

As of Java SE 7, the HotSpot JVM has only one garbage collector that satisfies the soft real-time criteria. It is called the garbage first collector (G1), and is touted as the functional replacement for the CMS collector. The HotSpot JVM's G1 collector takes a wholly different approach then generational collection. It divides the heap up into a set of equally sized regions. One of these regions is chosen for the initial allocation, and new regions are chosen for allocation each time the current one is filled. Each individual region is associated with a *remembered set*, which indicates all locations that might contain references to live objects within its region.

Somewhat similar to the CMS collector, the G1 collector utilizes a concurrent Mark-Sweep algorithm based around object location data. Regions that are calculated to have the highest yields of dead objects, paired with the lowest time costs, are collected first, forming what is called the *collection set*. The collection set is then collected through an *evacuation pause*, a process that 'evacuates' the collection set regions by copying their live objects to other regions throughout the heap, leaving the remaining dead objects to be freed.

The nature of this process helps to maintain compaction within the heap. Typically, when dead objects are freed, it can leave heavily distributed pockets of space throughout the heap. Copying the live objects to new regions throughout the heap in an incremental fashion allows for the distributed

spaces to become contiguous. Contiguous spaces are useful because they allow for objects to be cleanly copied or allocated to regions. Instead of spending time searching for a pocket of space that fits, objects can simply be placed on the end of the last allocated object within a given region [6].

6. CONCLUSION

Having examined the many variations of GC that exist within the Java HotSpot Virtual Machine, it is clear that there does not exist any one modern GC strategy that vastly exceeds the performance capabilities of another. This is because GC selection is an adaptive process. Not only do the dynamics of the Java program play a part, but also do the underlying computer systems that process it.

When we consider the increased memory capacities and processing abilities of modern-day computer architecture, there are three criteria that stand out:

(1) Incurring low stop-the-world pause-times.

(2) Achieving high throughput.

(3) Achieving low latency.

The HotSpot JVM garbage collectors described within this paper address these criteria in many different and interesting ways. For example, the CMS collector well addresses both criteria (1) and (2) because of its concurrent behavior, but also neglects criteria (3) for the same reason. Emphasis on any one trait, such as high throughput or low latency, can and will lead to deficiencies in another. The advantage of garbage collecting on rapidly evolving computing architectures is that we are given more and more leeway to delegate these deficiencies on to the underlying computer system. Bigger memory capacities, for example, allow us to select memory intensive collection algorithms with less of a risk to memory overhead.

So to conclude, the potential to benefit from GC lies not only in your Java program itself, but also within your program's expected audience and computing environment(s). Understanding this is crucial to effectively preparing for the performance challenges that now exist, and may yet exist in the future.

7. REFERENCES

[1] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 11–20, New York, NY, USA, 1988. ACM.

[2] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 285–298, New York, NY, USA, 2003. ACM.

[3] D. F. Bacon, P. Cheng, and V. T. Rajan. A unified theory of garbage collection. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, pages 50–68, New York, NY, USA, 2004. ACM.

[4] T. Brecht, E. Arjomandi, C. Li, and H. Pham. Controlling garbage collection and heap growth to reduce the execution time of java applications. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 353–366, New York, NY, USA, 2001. ACM.

[5] O. Corporation. Java se 6 hotspot^{TM}virtual machine garbage collection tuning, 2011. [Online; accessed 3-November-2011].

[6] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM.

[7] D. Frampton, D. F. Bacon, P. Cheng, and D. Grove. Generational real-time garbage collection. In *ECOOP*, pages 101–125, 2007.

[8] R. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996. With a chapter on Distributed Garbage Collection by Rafael Lins. Reprinted 1997 (twice), 1999, 2000.

[9] T. Kalibera, F. Pizlo, A. L. Hosking, and J. Vitek. Scheduling real-time garbage collection on uniprocessors. *ACM Trans. Comput. Syst.*, 29:8:1–8:29, August 2011.

[10] Y. Levanoni and E. Petrank. An on-the-fly reference-counting garbage collector for java. *ACM Trans. Program. Lang. Syst.*, 28:1–69, January 2006.

[11] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th international symposium on Memory management*, ISMM '08, pages 11–20, New York, NY, USA, 2008. ACM.

[12] S. Microsystems. Memory management in the java hotspot^{TM}virtual machine, 2006. [Online; accessed 20-October-2011].

[13] T. Printezis and D. Detlefs. A generational mostly-concurrent garbage collector. In *Proceedings of the 2nd international symposium on Memory management*, ISMM '00, pages 143–154, New York, NY, USA, 2000. ACM.

[14] D. Vengerov. Modeling, analysis and throughput optimization of a generational garbage collector. In *Proceedings of the 2009 international symposium on Memory management*, ISMM '09, pages 1–9, New York, NY, USA, 2009. ACM.

[15] M. Wegiel and C. Krintz. The mapping collector: virtual memory support for generational, parallel, and concurrent compaction. *SIGPLAN Not.*, 43:91–102, March 2008.

[16] Wikipedia. Garbage collection (computer science) — wikipedia, the free encyclopedia, 2011. [Online; accessed 4-October-2011].

[17] Wikipedia. Java (programming language) — wikipedia, the free encyclopedia, 2011. [Online; accessed 4-October-2011].

[18] B. Zorn. Comparing mark-and sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 87–98, New York, NY, USA, 1990. ACM.