TECHNOLOGY
WHITE PAPER

Understanding Java Garbage Collection

And What You Can Do About It

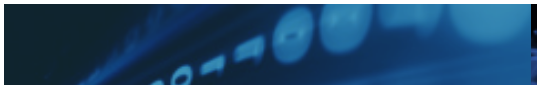


Table of Contents

Executive Summary

Garbage Collection (GC) is an integral part of application behavior on Java platforms, yet it is often misunderstood. Java developers need to understand how GC works and how the actions they can take in selecting and tuning collector mechanisms, as well as in application architecture choices, can affect runtime performance, scalability and reliability.

This white paper reviews and classifies the various garbage collectors and collection techniques available in JVMs today. This paper provides an overview of common garbage collection techniques, algorithms and defines terms and metrics common to all collectors including:

. Generational
. Parallel
. Stop-the-world
. Incremental
. Concurrent
. Mostly-concurrent

The paper classifies each major JVM collector's mechanisms and characteristics and discusses the trade-offs involved in balancing requirements for responsiveness, throughput, space, and available memory across varying scale levels. The paper concludes with some pitfalls, common misconceptions, and "myths" around garbage collection behavior, as well as examples of how certain choices can result in impressive application behavior.

Introduction

The Java programming language utilizes a managed runtime (the Java Virtual Machine, or JVM) to improve developer productivity and provide cross-platform portability. Because different operating systems and hardware platforms vary in the ways that they manage memory, the JVM performs this function for the developer, allocating memory as objects are created and freeing it when they are no longer used. This process of freeing unused memory is called 'garbage collection' (GC), and is performed by the JVM on the memory heap during application execution.

Java garbage collection can have a big impact on application performance and throughput. As the JVM heap size grows, so does the amount of time that an application must pause to allow the JVM to perform GC. The result can be long, unexpected pauses that can delay transactions, deteriorate application throughput, cause user-session time-outs, force nodes to fall out of clusters, or result in even more severe business related losses (e.g. drop in revenue or damage to reputation).

This paper explains in more detail how garbage collection works, the different algorithm types employed by commercially available JVMs, and how developers and architects can make better informed decisions on which garbage collector to use and how to maximize application performance.

Why Care About the Java Garbage Collector?

Overall garbage collection is much better and more efficient than you might think. It's much faster than malloc() at allocating memory and dead objects cost nothing to collect (really!). GC will find all the dead objects, even in cyclic graphs, without any assistance from the developer. But in many ways garbage collection is much more insidious than many developers and architects realize.

For most collectors GC related pauses are proportional to size of their heaps which is approximately 1 second for each gigabyte of live objects. So, a larger heap (which can be advantageous for most apps) means a longer pause. Worse yet, if you run a 20 minute test and tune until all the pauses go away, the likelihood is that you've simply moved the pause to the 21st minute. So unfortunately, the pause will still happen and your application will suffer. In addition, the presence of garbage collection doesn't eliminate object leaks – the developer still has to find and fix references holding those leaked objects.

The good news is Java does provide some level of GC control. Developers and architects can make decisions that can adjust application performance, due to the behavior of the garbage collector. For example, in C++ it makes sense to null every reference field when it's no longer needed. However, in a Java program, coding in nullifiers everywhere is disastrous and far worse than coding in nothing. If every single class uses a finalizer to null reference fields, the garbage collector will potentially have to perform millions of object finalizations per GC cycle – leading to very long garbage collection pauses.

Trying to solve garbage collection at the application programming layer is dangerous. It takes a lot of practice and understanding to get it right; time that could better spent building value-added features. And, even if you make all the right decisions, it is likely that other code your application leverages will not be optimized or the application workload changes over time, and your application will still have GC related performance issues.

Also, depending on the characteristics of your application, choosing the wrong garbage collector type or using the wrong settings can greatly increase pause times or even cause out-of-memory crashes. With a proper understand of garbage collection and what your available options are, you can make better informed decisions and product choices that can improve the performance and reliability of your application at runtime.

Classifying the Collector

Garbage collectors are divided into several types. For each type some collectors are categorized as 'mostly', as in 'mostly concurrent'. This means that sometimes it doesn't operate according to that classification and has a fallback mechanism for when that occurs. So, a 'mostly concurrent' collector may operate concurrently with application execution and only occasionally stop-the-world if needed.

Concurrent collector – performs garbage collection concurrently while application execution continues.

Parallel collector – uses multiple threads. A collector can be concurrent but not parallel, and it can be concurrent AND parallel. (Side note – be cautious when researching older literature on garbage collection, since what we used to call parallel is now called concurrent.)

Stop-the-world (STW) – is the opposite of concurrent. It performs garbage collection while the application is completely stopped.

Incremental – performs garbage collection as a series of smaller increments with potentially long gaps in between. The application is stopped during garbage collection but runs in between increments.

Moving – the collector moves objects during garbage collection and has to update references to those live objects.

Conservative – most non-managed runtimes are conservative. In this model, the collector is unsure of whether a field is a reference or not, so it assumes that it is. This is in contrast to a Precise Collector.

Precise – a precise collector knows exactly where every possible object reference is. A collector cannot be a moving collector without also being precise, because you have to know which references to fix when you move the live objects. Precise collectors identify the live objects in the memory heap, reclaim resources held by dead objects and periodically relocate live objects.

Most of the work the virtual machine does to be precise, is actually in the compiler, not the collector itself. All commercial JVMs today are moving and precise.

Steps in Garbage Collection

Before the garbage collector can reclaim memory, it must ensure the application is at a 'GC safepoint'. A GC safepoint is a point or range in a thread's execution where the collector can identify all the references in the thread's execution stack. The terms 'safepoint' and 'GC safepoint' are often used interchangeably, however many types of safepoints exist, some of which require more information than a GC safepoint. A 'Global Safepoint' is when all application threads are at a safepoint.

Safepoint opportunities in your code should be frequent. If the garbage collector has to wait for a safepoint that is minutes (or longer) away, your application could run out of memory and crash before garbage can be collected. Once the GC safepoint is reached, garbage collection can begin.

Mark

This phase, also known as 'trace', finds all the live objects in the heap. The process starts from the 'roots', which includes thread stacks, static variables, special references from JNI code and other areas where live objects are likely to be found. A reference to an object can only prevent the object from being garbage collected, if the reference chains from a GC root.

The garbage collector 'paints' any objects it can reach as 'live'. Any objects left at the end of this step are 'dead'. If any objects are still reachable that the developer thought were dead, it's an object leak, a form of memory leak.

The work of a marker is linear to the amount of live objects and references, regardless of the size of the objects. In other words it takes the marker the same amount of time to paint 1,000 10KB objects as 1,000 1MB objects.

In concurrent marking all reachable objects are being marked as live, but the object graph is mutating (i.e. changing) as the marker works. This can lead to a classic concurrent marking race. The application can move a reference that has not yet been seen by the marker into an object that has already been visited. If this change is not intercepted or prevented in some way, it can corrupt the heap. The object would be collected, even though a reference to it still exists.

Usually a 'write barrier' is used to prevent this condition. The write barrier captures changes to object references (e.g. in a card table) that might otherwise be missed by the marker. With this information, the marker can revisit all mutated references and track new mutations. When the set is small enough, a stop-the-world pause can be used to catch up, making the collector 'mostly' concurrent. But it is important to note that the collector is sensitive to the mutation rate and the amount of work done grows with the mutation rate and may fail to finish.

Sweep

In this phase the garbage collector scans through the heap to identify the locations of 'dead' objects and tracks their location, usually in some sort of 'free list'. Unlike the Mark phase, the work done by the Sweep phase is linear to the size of the heap, not the size of the live set. If your application is using a very large heap with very little left alive, Sweep still has to scan on the entire heap.

Compact

Over time, the Java memory heap gets 'fragmented', where the dead space between objects is no longer large enough to hold new objects, making new object allocation slower. If your application creates objects of variable sizes, fragmentation will happen more quickly. XML is a great example of this. The format is defined but the size of the information in the object is not controlled, often leading to objects with great variations in sizes and a fragmented heap.

In the Compact phase the garbage collector relocates live objects to free up contiguous empty space. As these objects are moved, the collector must fix all references in the threads to these live objects, called 'remapping'. Remap has to cover all references that could point to an object, so it usually scans everything. The amount of work done in this phase is generally linear to the size of the live set.

Incremental compaction is used in a couple of commercial collectors (Oracle G1 and the Balanced Collector from IBM). This technique assumes that some regions of memory are more popular than others, although this is not always the case depending on the application. The GC algorithm tracks cross-region remembered sets (i.e. which region points to which). This allows the collector to compact a single region at a time and only scan regions pointing into it when remapping all potential references. The collector identifies region sets that fit into limited pause times, allowing the maximum time for application interruption to be controlled. Large heaps have fewer non-popular regions; the number of regions pointing into a single region tends to be linear to the size of the heap. Because of this, the work for this type of compaction can grow with the square of the heap size.

Types of Collectors

Mark/Sweep/Compact Collector – performs the three phases as three separate steps.

Mark/Compact Collector – skips the sweep and moves live objects to a contiguous area of the heap.

Copying Collector – performs all three phases in one pass. A copying collector is pretty aggressive. It uses a 'from' and 'to' space and moves all the live objects then fixes the references all in one pass. When the 'from' space is empty the collection is complete. Work done in a copying collector is linear to the size of the live set.

Fig. 1: Comparing collector types

	Mark/Sweep/Compact	**Mark/Compact**	**Copying**
Amount of memory needed to perform collection	1x the size of the live set plus a little more	2x the size of the live set	2x the size of the live set
Monolithic – the whole heap must be garbage collected at once	No	No	Typically yes
Amount of work linear to	Size of heap (in sweep)	Size of live set	Size of live set
Requires large amount of 'free' memory	No	Yes	Yes
Fastest for	'Full' heaps with little free memory; large heaps	Heaps that become fragmented in M/S/C	Low live object counts
Fragments the heap	Yes	Yes	No

Generational Collectors

A generational collector is based on the hypothesis that most objects die young. The application creates them, but quickly doesn't need them anymore. Often a method creates many objects but never stores them in a field. When the method exits those objects are ready for garbage collection. The developer can set a 'generational filter' that reduces the rate of allocation into the old generation. This filter is the only way to keep up with today's CPU throughput – otherwise applications can create objects much faster than garbage collection can clean them up.

For applications where this hypothesis holds true, it makes sense to focus garbage collection efforts on the 'young generation', and promote objects that live long enough to an 'old generation' which can be garbage collected much less frequently as it fills up.

Because these young generation objects die quickly, the live set in the young generation takes up a small percentage of the available space. Thus a moving collector makes sense, since we have space in which to place live objects and the work done is linear to the size of the live set which is small. A generational collector doesn't need memory equal to 2x the live set, since objects can spillover into the old generation space. This compensates for the main downside of a copying collector. You can allow the young generation to get completely full before collecting it.

Deciding when to promote objects can dramatically improve efficiency. Keeping objects in the young generation a little longer may allow many of them to die and save collection time. If you keep them too long the young generation can run out of space or ruin the generational assumption altogether. Waiting too long to promote can also dramatically increase the work needed to copy the live objects and therefore the time it takes to do GC.

Remembered Set

Generational collectors use a 'remembered set' to track all references into the young generation from the outside, so the collector doesn't have to scan for them. This set is also used as part of the 'roots' for the garbage collector. A common technique is 'card marking', which uses a bit (or byte) indicating that a word or region in the old generation is suspect. These 'marks' can be precise or imprecise, meaning it may record the exact location or just a region in memory. Write barriers are used to track references from the young generation into the old generation and keep the remembered set up-to-date.

Oracle's HotSpot uses what's called a 'blind store'. Every time you store a reference it marks a card. This works well, because checking the reference takes more CPU time, so the system saves time by just marking the card.

Commercial Implementations

Commercial server-side JVMs typically use a copying collector for the young generation that employs a monolithic, stop-the-world collection. In other words the collector stops application processing and copies the entire live set into a new section of the heap. The old generation usually uses a Mark/Sweep/Compact collector which may be stop-the-world, concurrent, mostly concurrent, incremental stop-the-world or mostly incremental stop-the-world.

[1] Normally, the system wants to be able to get to the large live set in the old generation without having to stop at some increment.

1 Translation, in order: it may stop the application entirely to perform the collection, perform collection concurrent with application processing, collect concurrently with the application up to a point when it (for example) gets behind, and has to stop the application to catch up, stop the application processing for shorter periods to do part of the garbage collection at a time, or it may do these incremental collections for as long as possible, before it has to stop the application to complete GC.

What Developers and Architects Can Do

First, understand the characteristics of your application and the basics of how garbage collection works.

Garbage Collection Metrics

Many characteristics of your application will affect garbage collection and performance at runtime. First is how fast your application is allocating objects in memory, or the allocation rate. Next is how long those objects live. Do you have a fairly typical application where objects die young, or do you have many objects that are needed for a long time? Your program may also be updating references in memory, called the mutation rate. The mutation rate is generally linear to the amount of work the application is doing. And finally, as objects are created and are dying, another set of metrics to track is the live set (also called the 'heap population') and the heap shape, which is the shape of the live object graph.

The mark time and compaction time are the most important metrics to track for overall garbage collection cycle time. Mark time is how long it takes for the collector to find all live objects on the heap. Compaction time is how long it takes to free up memory by relocating objects, and is only relevant for a Mark/Compact collector. For Mark/Sweep/Compact, sweep time is also important, which indicates how long it takes the collector to locate all the dead objects. Cycle time for the collector is the total time from the start of garbage collection, until memory is freed and available for use by the application.

The Need for Empty Memory for GC

Garbage collectors need at least some amount of empty memory in order to work. More empty memory makes it easier (and faster) for the garbage collector. Doubling empty memory halves the work done by the collector and halves the CPU consumption needed to run. This is often the best tool for efficiency.

To illustrate, here are a couple of intuitive limits. If we have infinite empty memory we would never have to collect and GC would never use any CPU time. If we have exactly 1 byte of empty memory at all times, the collector would have to work very hard and GC would use up 100% of CPU time. Overall, garbage collection CPU time follows an approximate 1/x curve between these two limits, with effort dropping as empty memory increases.

Mark/Compact and Copying collector work is linear to the size of the live set. How often each needs to run is determined by the amount of empty memory. Since collection is a fixed amount of work each time, doing this work less often is more efficient. In these two types of collectors the amount of empty memory available doesn't control the length of the garbage collection pause, only the frequency. On the other hand Mark/Sweep/Compact work grows as the heap grows. More empty memory for a collector that pauses for sweeping, means less frequent but longer pauses.

Now that we understand what the characteristics are for our application, we can make changes that will improve performance, scalability and reliability.

GC Strategy: Delaying the Inevitable

Although compaction is unavoidable in practice, many GC tuning techniques focus on delaying a full GC as long as possible and freeing the 'easy' empty space as quickly as possible.

Generational garbage collection can be partially effective at delaying the inevitable. Young generation objects are collected frequently and this doesn't take much time. But eventually, space in the old generation must be reclaimed using a monolithic, stop-the-world collection. Another delay strategy is to perform concurrent marking and sweeping, but skip compaction. Freed memory can be tracked in lists and reused without moving live objects. But over time this will leads to fragmentation, forcing a compaction.

Finally, some collectors rely on the idea that much of the heap is not popular. A non-popular region will only be pointed to from a small portion of the overall heap. Compaction can be done on non-popular regions incrementally with short stop-the-world pauses to free up space. However, at some point the popular regions will need to be compacted, causing an application pause.

The bottom line is that despite numerous techniques and creative ways to tune away garbage collection pauses, competition is inevitable with most commercial collectors. Developers and architects need to make good decisions about which collector to use to maximize application performance.

Choosing a Garbage Collector

The following table summarizes the types of popular commercially available garbage collectors.

	Collector Name	Young Generation	Old Generation
Oracle's HotSpot	ParallelGC	Monolithic, stop-the-world, copying	Monolithic, stop-the-world, Mark/Sweep/Compact
Oracle's HotSpot	CMS (Concurrent Mark/Sweep)	Monolithic, stop-the-world, copying	Mostly concurrent, non-compacting
Oracle's HotSpot	G1 (Garbage First)	Monolithic, stop-the-world, copying	Mostly concurrent marker, mostly incremental compaction, fall back to monolithic stop the world
Oracle's JRockit*	Dynamic Garbage Collector	Monolithic, stop-the-world, copying	Mark/Sweep - can choose mostly concurrent or parallel, incremental compaction, fall back to monolithic stop-the-world
IBM J9*	Balanced	Monolithic, stop-the-world, copying	Mostly concurrent marker, mostly incremental compaction, fall back to monolithic stop-the-world
IBM J9*	Opt throughput	Monolithic, stop-the-world, copying	Parallel Mark/Sweep, stop-the-world compaction
Zing	C4 (Continuously Concurrent Compacting Collector)	Concurrent, compacting	Concurrent, compacting

*Can choose a single or 2-generation collector

Oracle's HotSpot ParallelGC

This is the default collector for HotSpot. It uses a monolithic, stop-the-world copying collector for the young generation and a monolithic, stop-the-world Mark/Sweep/Compact algorithm for the old generation.

Oracle's HotSpot CMS

The Concurrent Mark/Sweep collector (CMS) is an option in HotSpot. It attempts to reduce the old generation pauses as much as possible by concurrently marking and sweeping the old generation without compacting. Once the old generation becomes too fragmented, it falls back to a monolithic, stop-the-world compaction.

CMS performs mostly concurrent marking. It marks concurrently while the application is running and tracks any changes (mutations) in card marks which are revisited as needed. CMS also does concurrent sweeping, which frees up space occupied by dead objects and creates a free list for allocation of new objects. When objects can no longer be promoted to the old generation or concurrent marking fails, the system will throw a 'promotion failure' message and/or pause for a second or more while it performs compaction.

The young generation uses a monolithic, stop-the-world copying collector, just like the ParallelGC option.

Oracle's HotSpot G1GC (Garbage First)

G1 is an option in HotSpot. Its goal is to avoid, "as much as possible" a full GC. G1 uses a mostly concurrent marker for the old generation. It marks concurrently as much as possible, then uses a stop-the-world pause to catch up on mutations and reference processing. G1 tracks inter-regional relationships in remembered sets and does not use fine-grained free lists. It uses stop-the-world, mostly incremental compaction and delays compaction of popular objects and regions as much as possible. G1 falls back to monolithic, stop-the-world full compaction of these popular areas when needed.

The young generation uses a monolithic, stop-the-world copying collector, just like the ParallelGC and CMS options.

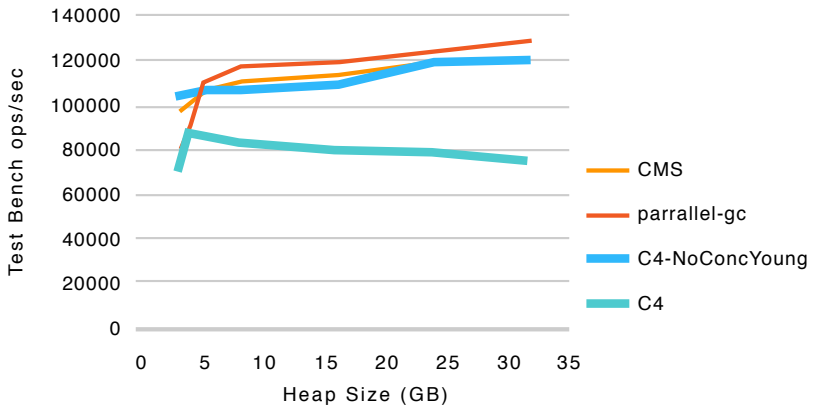
Azul Systems Zing C4

C4 (Continuously Concurrent Compacting Collector) is the default in Azul Systems' Zing. C4 concurrently compacts both the young generation and old generation. Unlike other algorithms, it is not 'mostly' concurrent, but fully concurrent, so it never falls back to a stop-the-world compaction.

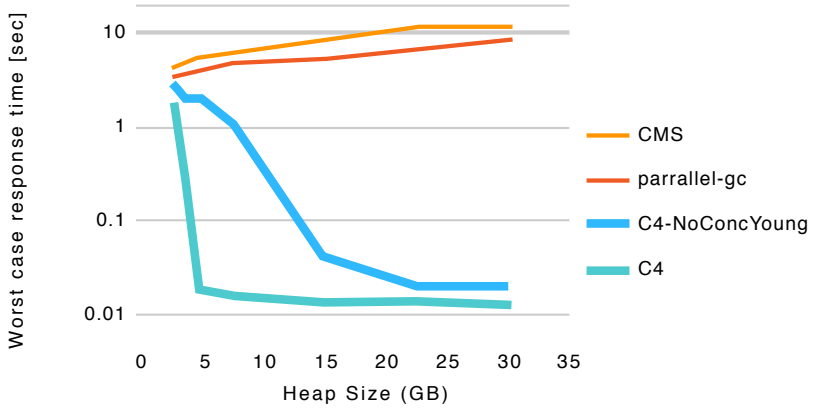
C4 uses a Load Value Barrier (LVB) to verify each heap reference as correct when loaded. Any heap references that point to relocated objects are caught and fixed in this self-healing barrier. C4 has a guaranteed single pass concurrent marker. No matter how fast your application mutates the heap, C4 can keep up. The C4 collector also performs concurrent reference processing (including weak, soft and final references) and relocation and remapping are both concurrent.

C4 also uses 'quick release' to make freed memory available quickly to the application and for the relocation of objects. This enables 'hand over hand' compaction that does not require empty memory to function.

Test Raw Bench Throughput of Collectors



Worst Case Response Times



GC Tuning Observations

Garbage collection tuning for most collectors is hard to get right, even when you understand the characteristics of your application. The figure below shows two sets of tuning parameters for the CMS collector in the HotSpot JVM. While they may use similar parameter, they are very different and in some areas diametrically opposed. Yet the performance of your application could be optimized with either set, depending on its particular characteristics. With most collectors no 'one size fits all' answer exists. Developers and architects have to tune garbage collection carefully and retune every time the application, environment or anticipated load changes. Getting these parameters wrong can cause unexpected, long pauses during peak load times.

However, performance of an application on the Azul Zing C4 collector is insensitive to the 'usual' tuning parameters. Because it marks and compacts concurrently in both the young and old generations the only important parameter is heap size.

When you set the heap size in the Zing runtime, the C4 collector computes the GC time it needs in order to keep up with allocation rates. You do not need to set ratios or evacuation times. Zing's C4 GC will fire when needed using threads that are separate from your application threads. This allows worst case pause times to drop by orders of magnitude compared to other GC strategies.

With GC out of the way, time to safepoint emerges as the next dominant source of delays, and even lower latencies may be achieved using other controls, typically in the OS, or in the JVM through JIT compiler adjustment.

Sample HotSpot JVM CMS Settings

```
Java -Xmx12g -XX:MaxPermSize=64M -XX:PermSize=32M-XX:MaxNewSize=2g
       -XX:NewSize=1g -XX:SurvivorRatio=128 -XX:+UseParNewGC
       -XX:+UseConcMarkSweepGC -XX:MaxTenuringThreshold=0
       -XX:CMSInitiatingOccupancyFraction=60 -XX:+CMSParallelRemarkEnabled
       -XX:+UseCMSInitatingOccupancyOnly -XX:ParallelGCThreads=12
       -XX:LargePageSizeInBytes=256m ...

Java -Xms8g -Xmx8g -Xmn2g -XX:PermSize=64M -XX:MaxPermSize=256M
       -XX:-OmitStackTraceInFastThrow -XX:SurvivorRatio=2 -XX:-UseAdaptiveSizePolicy
       -XX:+UseConcMarkSweepGC -XX:+CMSConcurrentMTEnabled
       -XX:+CMSParallelRemarkEnabled -XX:+CMSParallelSurvivorRemarkEnabled
       -XX:+CMSMaxAbortablePrecleanTime=10000 -XX:+UseCMSInitiatingOccupancyOnly
       -XX:CMSInitiatingOccupancyFraction=63 -XX:+UseParNewGC -Xnoclassgc ...
```

Sample Azul Zing C4 Collector Settings

```
Java -Xmx40g
```

Summary

Garbage Collection (GC) is an integral part of application behavior on Java platforms, yet it is often misunderstood. Java developers can improve application performance, scalability and reliability by understanding how GC works and by making better garbage collector choices.

The main drawback of most garbage collectors is the need for long application pauses. These pauses are the result of an inevitable requirement to compact the heap to free up space. Collectors use different strategies to delay these events, but compaction is inevitable for all commercial available collectors except Azul C4, which employs a Continuously Concurrent Compacting Collector that avoids pauses altogether.

About Azul Systems

Azul Systems delivers Java solutions that meet the needs of today's real-time business. Designed and optimized for x86 servers and enterprise-class workloads, Azul's Zing is the only Java runtime that supports highly consistent and pauseless execution for throughput-intensive and QoS-sensitive Java applications. Azul's enhanced Java technologies also enable organizations optimize the value of their capital investments while ensuring that their human capital is free to generate new features, unlocking creativity and driving competitive advantage.

Contact Azul

To discover how Zing can improve scalability, consistency and performance of all your Java deployments, contact:

Azul Systems, Inc.

1173 Borregas Ave

Sunnyvale, CA 94089

+1.650.230.6500

www.azulsystems.com

info@azulsystems.com

Appendix A: Garbage Collection Terminology

Compaction The garbage collection phase that defragments the heap, moves objects in memory, remaps all affected references and frees contiguous memory regions.

Concurrent A type of garbage collection algorithm that where GC is done while the application is running.

Copying A garbage collector that copies that performs mark/sweep/compact all at once by copying live objects to a new area in memory.

Dead object An object that is no longer being referenced by the application.

GC safepoint A point or range in a thread's execution where the collector can identify all the references in the thread's execution stack.

Generational Objects in memory are split between a young generation and old generation and garbage collected separately.

Incremental Garbage collects only a portion of the heap at a time.

Live object One that is still being referenced by the application.

Marking The garbage collection phase that identifies all live objects in the heap.

Monolithic Garbage collects the entire heap at once.

Mutator Your application, which is changing (mutating) references in memory.

Parallel A collector that uses multiple threads.

Pause Time period when the application is stopped while garbage collection is occurring.

Precise A precise collector knows exactly where every possible object reference is.

Promotion Allocating an object from the young generation to the old generation of the heap.

Remembered Set Tracks all references into the young generation from the outside so the collector doesn't have to scan for them.

Roots Starting points for the garbage collector to find live objects.

Stop-the-World Indicates that the garbage collector stops application processing to collect the heap.

Sweeping The garbage collection phase that locates dead objects.