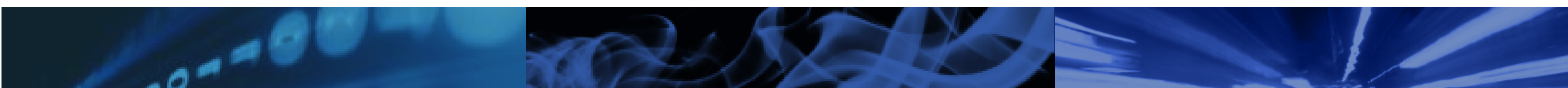
TECHNOLOGY
WHITE PAPER

Azul Pauseless Garbage Collection

Providing continuous, pauseless operation for Java applications



Executive Summary

Conventional garbage collection approaches limit the scalability of Java applications. At large memory heap sizes (> than a few GB), garbage collection (GC) pause times become too long, causing noticeable delays in processing and impacting business. The Azul C4 (Continuously Concurrent Compacting Collector) garbage collection algorithm improves performance and removes barriers to Java scalability by eliminating pause times even at very large heap sizes. Using a combination of software techniques and hardware emulation features, the Azul C4 uses 'concurrent compaction' to allow applications to continue processing while remapping memory. The result is Java applications that scale to tens of cores and hundreds of gigabytes of heap memory without degrading response time or application throughput.

This unique technology, which was part of Azul's Vega hardware appliances, is now available as part of Zing, an innovative JVM. Using C4 garbage collection, Zing® provides:

- Predictable, consistent garbage collection (GC) behavior
- Predictable, consistent application response times
- Smoothly scale to memory heap sizes over 300 GB with no business-interruptive pauses
- Unmatched Java application scalability
- Greatly simplified deployments with fewer, larger instances
- Reduced garbage collection (GC) tuning and JVM tuning

Table of Contents

Executive Summary 2

Java Scalability Challenges 4

Solution: The ZING Elastic Runtime 5

- Zing Components 5

The Azul C4 Garbage Collection Algorithm 6

- Concurrent Marking 7
- Concurrent Relocation 8
- The Concurrent Marking Problem 8
- Solving the Concurrent Marking Problem 10
- The Concurrent Relocation Problem 11
- Solving the Concurrent Relocation Problem 11
- C4 Garbage Collection 11
  - Mark Phase 12
  - Relocate Phase 13
  - Details of the Relocate Phase 14
  - Remap Phase 15

GC Tuning and JVM Tuning 16

Benefits 16

Conclusion 17

Contact Us 17

Java Scalability Challenges

Large transaction processing applications with heavy resource requirements, such as ecommerce and web portals, have become mainstream due to the growth of the Internet. These business-critical applications, most based on the Java platform, face challenges that were inconceivable just a few years ago. They demand access to large amounts of data and memory at runtime to support high user and transaction loads. However, because of the resulting lengthy garbage collection pauses, Java applications quickly reach scalability barriers.

The garbage collection process automatically frees the heap space used by objects that are no longer referenced by the application. This makes the programmer's life easier, since he or she no longer has to keep track and free allocated memory. Automating this process reduces the number of bugs in the final program and saves developers plenty of headaches. In conventional Java Virtual Machines (JVMs), memory heap space is allocated per instance when the application is launched and is, therefore, rigidly limited. IT organizations spend hours tuning JVMs to the right amount of heap. Too much, and garbage collection pauses interrupt processing, too little and the instance is 'out-of-memory' and crashes.

The Java garbage collector also has to spend time preventing memory heap fragmentation, where blocks of freed memory are left between blocks in use. When too much fragmentation occurs, new objects may not fit in the largest continuous space and have to be created by extending the heap, even though overall enough unused memory exists. On a virtual memory system, heap growth results in extra paging which can also negatively impact performance. (Venners, 1996) Heap fragmentation eventually and inevitably leads to compaction, where the garbage collector reclaims unused memory between live objects. Moving the live objects requires scanning and fixing all references to reflect the new location. In today's JVMs, compaction is universally done as a 'stop the world' event that suspends application processing. The amount of time it takes to perform compaction grows linearly with memory heap size, so if it takes two seconds to compact a 1GB heap it will take 20 seconds to compact 10 GB. At 20-30 seconds, the garbage collection delay becomes a crash or an outage, which is unacceptable for business-critical applications. Long compaction pauses create a practical limit on memory heap size per Java instance of about 2-4 GB.

The impact to business, IT organization and developers is significant. Businesses rely on consistently fast application response times to create an outstanding user experience that drives revenue. Garbage collection pauses create inconsistent and slow response times that impact customers. As a result, systems administrators deploy tens or hundreds of instances using just a few GBs each to keep pause times low and spend hundreds of additional hours tuning memory heap size, the JVM and garbage collection to find ways to delay compaction. Developers create complex memory management schemes and shift load to the database tier to keep memory heap size low. This creates a complex, difficult to manage environment that suffers from reduced efficiency; heavy, cross-JVM communication; increased overall memory footprint and vastly under utilized servers.

To simplify deployments, remove scalability limitations and ensure consistently fast response times, a new deployment platform is needed that eliminates garbage collection pauses.

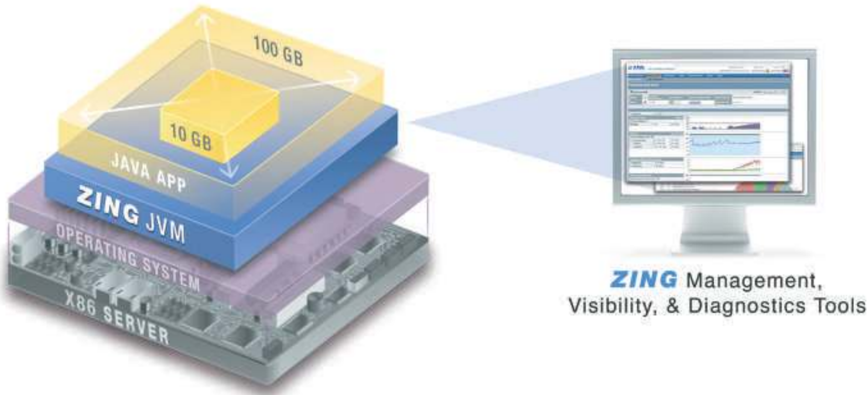
Solution: ZING

Zing shatters Java scalability barriers and enables existing Java applications to smoothly and reliably scale to dozens of CPU cores and hundreds of gigabytes of heap memory. Zing automatically scales resources for individual Java application instances up and down based on real-time demands.

Zing Components

Zing consists of a Java Virtual Machine (JVM), which is a full-featured JDK installed on the host server and Zing Vision (ZVision), a true, zero-overhead production JVM monitoring, visibility and diagnostics tool.

Zing seamlessly replaces your existing JVM with no coding changes. It allows existing Java applications to transparently tap the benefits of Azul's C4 pauseless garbage collection technology. C4 uses a combination of software techniques and hardware emulation to directly address unacceptable pause times. The garbage collector is parallelized, runs concurrently with the application, and is deterministic. Java application scalability can now reach hundreds of gigabytes of heap memory without degrading response-time consistency, application throughput, or efficiency. Complex workarounds, architectural restrictions and extensive JVM and GC tuning are no longer necessary.



The Azul C4 Garbage Collection Algorithm

The Azul C4 algorithm has multiple benefits compared to conventional garbage collection approaches.

Why Do Conventional Garbage Collectors Have Pauses That Limit Scalability and Performance?

Java and J2EE server environments have changed dramatically over the last decade. Today's applications are deployed in multi-core, multi-GB heap memory environments and need to deliver predictable service levels to a wide variety of end users. Figure 1 shows the two main characteristics that are evolving to match application requirements: parallelism and concurrency.

Although single CPU, small heap size systems were prevalent in early Java application deployments, today's applications push the limits of memory heap size and can run on servers capable of executing many threads concurrently. Single threaded collectors and non-concurrent collectors that stop the world are clearly no longer a good match for modern Java applications. The availability of parallel collectors on most commercial JVMs reflects this reality as does the emergence of partially concurrent or mostly concurrent collectors that increase the portions of garbage collection work that can be done concurrently with the application's execution.

While parallel and mostly parallel collectors have been delivered by most commercial JVMs, concurrency has proven to be a harder problem to solve using existing computer architectures due to two particularly difficult challenges to achieving concurrent collection: concurrent marking, and concurrent relocation. Without solving both problems, the best a collector can do is to reduce the frequency of stop-the-world pause events.

ZING C4 GARBAGE COLLECTOR BENEFITS

- Predictable, consistent garbage collection (GC) behavior
- Predictable, consistent application response times
- Smoothly scale to over 300 GB memory heap sizes
- Unmatched Java application scalability
- Greatly simplified deployments with fewer, larger instances
- Reduced garbage collection (GC) tuning and JVM tuning

GARBAGE COLLECTOR TERMINOLOGY

Concurrent Garbage collection is performed simultaneously with the execution of the application threads so that applications do not have to pause while the heap is garbage collected.

Parallel More than one garbage collection thread can be executing at the same time so that garbage collection can scale with the number of active application threads.

Compacting To avoid heap fragmentation, objects are moved by the collector to create more contiguous free space for faster new object allocation.

Figure 1 Benefits and challenges of parallelism and concurrency

Parallel

Benefit

Scales with CPU

Keeps up with garbage creation

Larger heap sizes

Same response time as load increases

Challenge

Provides no benefit in a single processor environment

Single Thread

Benefit

Simple, efficent on single CPU

Challenge

Does not scale

Limits application scale and drops in efficiency on multiprocessor systems

	Stop the World	Concurrency
Parallel	Parallel, Stop the World	Parallel, Concurrent
Single Thread	Single Thread, Stop the World	Single Thread, Concurrent

Stop the World

Benefit

Simple, efficient on single CPU

Challenge

Pauses

Pause times grow with heap size

Pause frequency grows with load multiprocessor systems

Concurrency

Benefit

Predictable response times

Challenge

Requires significant resources to achieve effective implementation

Concurrent Marking

Identifying live objects in a memory heap that is to go through garbage collection requires some form of identifying and "marking" all reachable objects. While marking is simple when the application is stopped, marking concurrently with a running program presents challenges. Multi-pass marking is sensitive to the mutation rate of the heap, and weak, soft and final references are all difficult to deal with concurrently. If concurrent marking cannot be completed in a bound period of time, a collector must resort to pauses to circumvent concurrency race conditions. These pauses may mark the entire heap while the program is paused. Among commercially available JVMs, the collector used by Zing is unique in its ability to concurrently mark the entire heap in a single pass for optimal GC performance.

Concurrent Relocation

Objects in the memory heap must also be concurrently relocated in the process of compacting and defragmenting the heap. In long-running Java applications compaction is unavoidable as the heap gets fragmented over time to a point where further object allocation is impossible without defragmentation. For this reason, all commercial JVMs except Azul perform memory heap compaction that pause application execution. Semi-concurrent collectors that do not concurrently compact delay the pause that they need to perform compaction, and will periodically pause to perform this task. This is typically the longest pause found in any garbage collector and has the most impact on performance. Among commercially available JVMs, Zing C4 garbage collection is unique in its ability to concurrently relocate objects and compact the heap without a stop-the-world pause.

The Concurrent Marking Problem

For a collector to identify garbage that can be collected, it must mark or traverse the entire set of live objects. Objects not marked as live in this traversal are "dead" and can be safely collected. By pausing the application and freezing all objects, a stop-the-world collector can mark and follow the static set of live objects quite simply. A concurrent collector, on the other hand, must coexist with a running application and a continually changing object graph.

Concurrent collectors must somehow cooperate with the application to ensure that no live objects are missed in the marking operation. Without such cooperation, hidden pointers to live objects can occur if an application thread rearranges the pointers to an object such that a live object that has already been processed by the collector now holds the only pointer to an object that has not yet been processed. Such hidden pointers result in the erroneous collection of a live object, and incorrect program execution. To further complicate matters, references to objects keep changing as the application executes. This makes multi-pass marking sensitive to the mutation rate of the heap. Weak, soft and final references are also very difficult to deal with concurrently.

Figure 2 Hidden objects can occur unless the collector thread is made aware of changed references

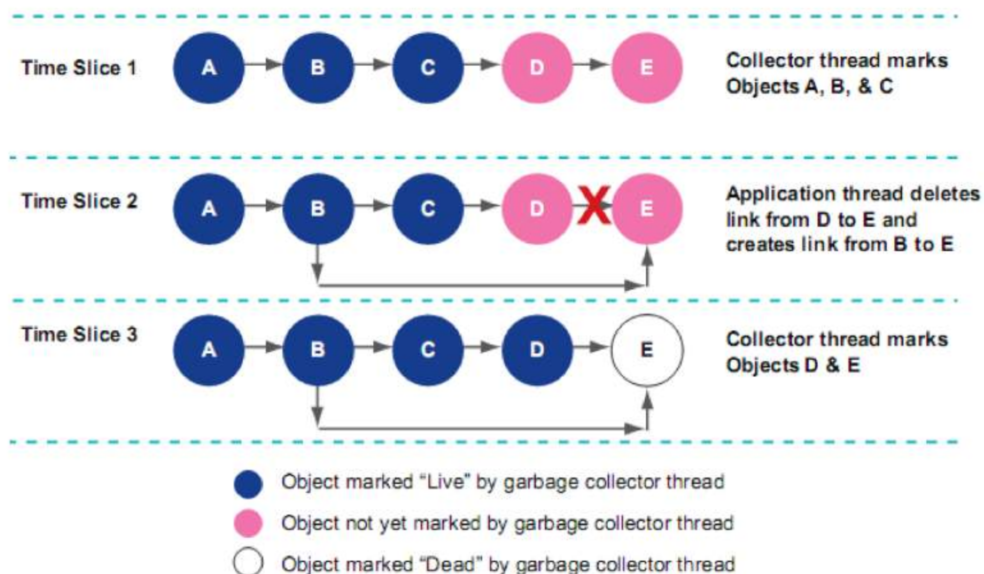
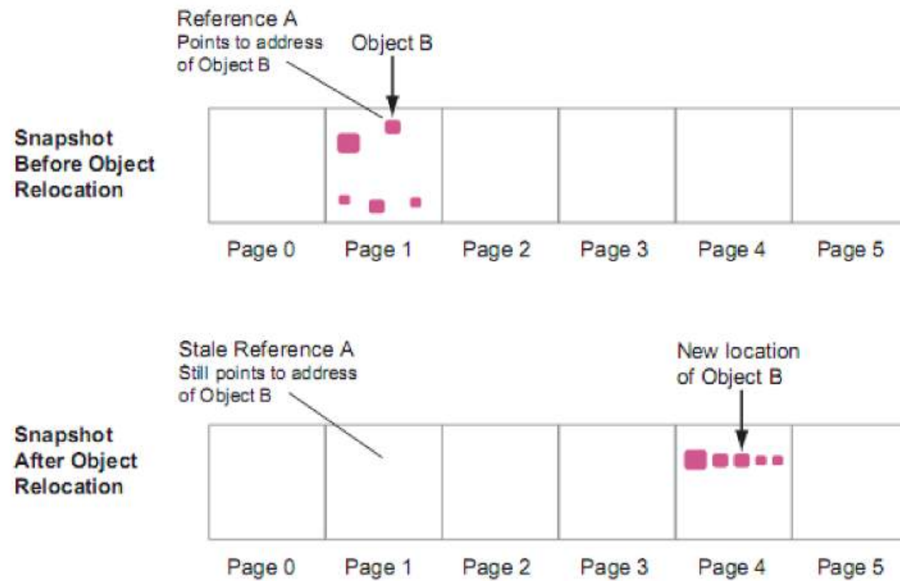


Figure 2 shows the sequence that would result in a hidden object being missed by the collector unless the collector is made aware of the changed references:

- Collector thread marks Objects A, B, and C as alive, then its time slice ends
- During Time Slice 1, Object B has no link to Object E and the collector thread has not yet processed Object D which does point to Object E, so Object E is not yet found by the collector
- Application thread loads Object D, finds the link to Object E, and keeps it in a register
- Application thread stores link to Object E in the Object B pointer as shown in Time Slice 2
- Object B now points to Object E and the garbage collector has already checked Object B so it won't check again to find Object E
- Application thread stores a null over Object D's pointer to Object E
- Collector thread continues marking and marks object D as alive and it marks Object E as dead because it has no knowledge that Object E is referenced by Object B (Time Slice 3)

Figure 3 Stale references must be updated before application threads try to access invalid data



Solving the Concurrent Marking Problem

A common approach to avoiding the concurrent marking race condition described above is to track all pointer modifications performed by the program as it runs. The collector then revisits all new or modified pointers after completing a first concurrent pass through the Java heap to mark objects. Some "mostly concurrent" collectors perform this revisit pass under stop-the-world full-pause conditions. A revisit pass can be performed concurrently and repeatedly as long as pointer modifications are still tracked by the application, but an eventual stop-the-world final pass is always necessary to complete the marking.

As long as the rate at which reference pointers are changed is low enough, the algorithm can eventually converge to a small enough list of changed pointers that stopping the world to revisit it would be unnoticeable. However, it is impossible to assure that this will ever happen. Program behavior that exhibits a high rate of pointer change, or continually changing behavior can easily present situations where such "mostly concurrent" marking passes never converge. The marking phase takes so long that the application exhausts all free heap space before the collector is able to free more objects, resulting in a full and lengthy garbage collection pause.

The Zing C4 collector performs fully concurrent marking which always completes in a single pass. Azul uses hardware emulated read barriers to intercept attempts by the application to use pointers that have not yet been seen by the garbage collector. The barrier logs such pointers to assure the collector visits them, tags the pointers as "seen by the collector", and continues the application's execution without a pause.

The Concurrent Relocation Problem

Relocating objects without stopping application execution is also challenging. Until now, no commercial JVM has provided such concurrent object relocation because of the complexity and costs involved in ensuring that all pointers to moved objects get updated correctly and data consistency is maintained. Figure 3 illustrates the problem of outdated references that result when objects are moved. After Object B is moved, it is easy to see that if Reference A is loaded into memory in an attempt to fetch data held in Object B, it would retrieve data from a location in memory Page 1 whereas Object B now resides in a different location in memory Page 4. The data retrieved from the wrong location compromises data integrity resulting in incorrect application execution. This problem is especially difficult, since no easy way exists to find all of the references to Object B and update them before an application thread tries to access the data in Object B through an old reference.

Without concurrently relocating objects, the heap gets continually fragmented, requiring an eventual defragmentation operation that would relocate objects under a stop-the-world pause. "Mostly concurrent" collectors will eventually perform such a defragmentation pause, which is typically the longest pause found in any garbage collector.

Solving the Concurrent Relocation Problem

The concurrent relocation problem is very difficult, and until now no commercial JVMs have attempted to solve it. Instead, "mostly concurrent" collectors simply succumb to a long pause when fragmentation gets too high to allocate objects in the heap.

The Zing C4 garbage collector (GPGC) performs fully concurrent object relocation, moving and compacting objects in the heap as part of its normal course of operation. Azul uses hardware emulated barriers to intercept any attempts by the application to use stale pointers referring to old locations of relocated objects. It then fixes those stale object pointers as they are encountered such that they point to the correct location without pausing the application.

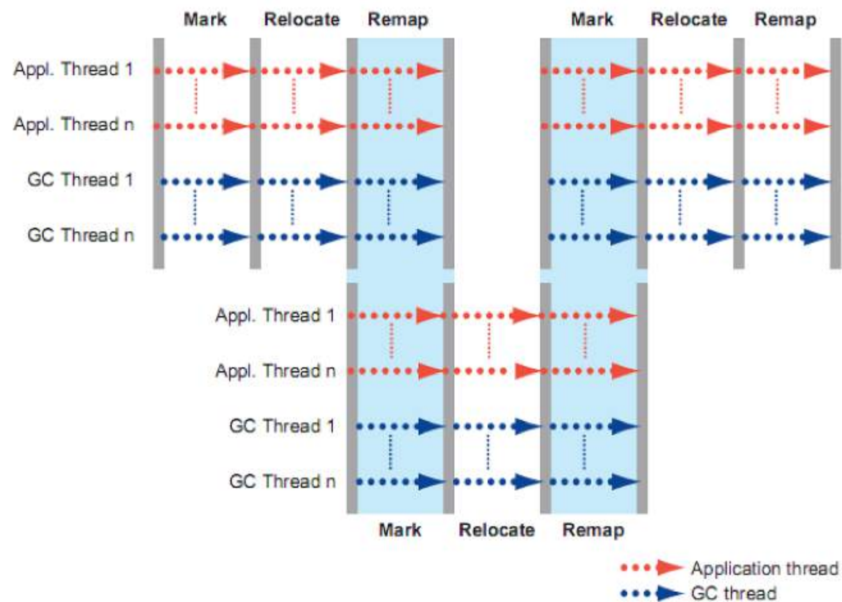
C4 Garbage Collection in Zing

The Azul C4 garbage collection algorithm relies on a fairly sophisticated read barrier implemented using hardware emulation to solve both the concurrent marking and concurrent relocation problems. The C4 algorithm is both concurrent and parallel and has the highly desirable quality of continually compacting and defragmenting the heap.

Central to the algorithm is a Loaded Value Barrier (LVB). Every Java reference is verified as 'sane' when loaded and any 'non-sane' references are fixed in a self-healing barrier. (A 'sane' reference has the correct behavior and state.) References that have not yet been marked through or that point to relocated are caught. This allows a guaranteed single pass concurrent marker and the ability to lazily and concurrently remap references.

While the Azul C4 algorithm actually requires zero pauses in applications, the JVM implementation does include brief pauses at phase transitions for simplicity and implementation expedience. These pauses are well below the threshold that would be noticeable to users and are not affected by application scale or data set size. The C4 algorithm completely decouples pause time from memory heap size, allowing applications to scale and use large amounts of heap memory without impacting performance or user response time. The result is a highly scalable environment that offers predictable performance for large Java applications.

Figure 4 The Azul garbage collection cycle



C4 also implements a 'young' and 'old' generation, providing 20x more throughput than a single generation. Both generations are concurrently compacted.

C4 is implemented in three major phases as illustrated in Figure 4. The figure illustrates three complete collection cycles, each cycle including a mark, relocate, and remap phase. It is important to note that the remap phase of each cycle runs concurrently with the mark phase of the next cycle.

The following sections provide an overview of what is accomplished in each of the three major phases of the algorithm.

Mark Phase

The mark phase finds all live objects in the Java heap. It works concurrently while application threads continue to execute, and traverses all reachable objects in the heap. The mark completes in a single pass, marking every live object in the heap as "live". Objects not marked "live" at the end of the mark phase are guaranteed to be "dead" can be safely discarded so their memory space can be reclaimed. In addition to marking all live objects, the mark phase maintains a count of the total live memory in each memory page (each page is currently 1MB). This information is later used by the relocation phase to select pages for relocation and compaction. To avoid the concurrent mark problem described earlier, the mark phase tracks not only marked objects, but also traversed object references (sometimes referred to as pointers). The algorithm tracks the object references that have been traversed by using an architecturally reserved bit in each 64 bit object reference. This bit, called the "not marked through" (NMT) bit, designates an object reference as either "marked through" or "not marked through". The marking algorithm recursively loops through a working list of object references and finds all objects reachable from the list. As it traverses the list, it marks all reachable objects as "live", and marks each traversed object reference as having been "marked through".

The marker's work list is "primed" with a root set which includes all object references in application threads at the start of a mark phase. All application thread stacks are scanned for object references, and all references are queued on the marker's work list for later traversal. After being queued for later traversal, each reference's NMT bit can be safely set to "marked through". Once a mark phase has started, it becomes impossible for the application threads to observe a "not marked through" reference without being intercepted by the LVB.

Zing includes a read barrier instruction contained in hardware emulation software that is aware of the NMT bit's function, and ensures application threads will never see an object reference that was not visited by the marker. The NMT bit is tested by the read barrier instruction that all application threads use when reading Java references from the memory heap. If the NMT bit does not match an expected value of "marked through" for the current GC cycle, the processor generates a fast GC trap. The trap code corrects the cause of the trap by queuing the reference to the marker's work list, setting the NMT bit to "marked through", and correcting the source memory location the object reference was loaded from, to ensure it, too includes the "marked through" indication. It then returns to normal application thread execution, with the application observing only the safely "marked through" reference. By using the read barrier and fast trapping mechanism, the marker is assured of safe marking in a single pass, eliminating the possibility of the application threads causing any live references to escape its reach. By correcting the cause of the trap in the source memory location (possible only with a read barrier that intercepts the source address), the GC trap has a "self-healing" effect since the same object references will not re-trigger additional GC traps. This ensures a finite and predictable amount of work in a mark phase.

The mark phase continues until all objects in the marker work list are exhausted, at which point all live objects have been traversed. At the end of the mark phase, only objects that are known to be dead are not marked as "live", and all valid references have their NMT bit set to "marked through". Several aspects of C4 enable the mark phase to have a highly deterministic quality. An entire Java heap (large or small) can be marked consistently without falling behind and resorting to long garbage collection pauses because the C4 employs:

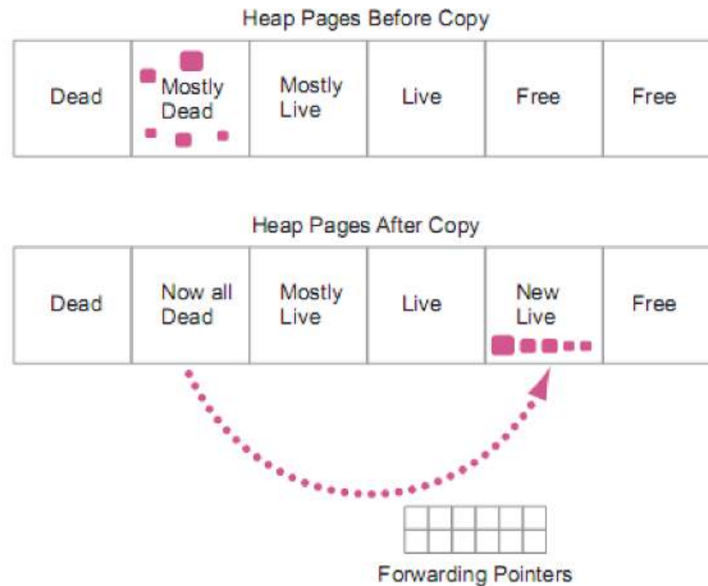
- Single pass completion to avoid the possibility of applications changing references faster than the mark phase can revisit them
- "Self-healing" traps that ensure a finite workload and avoid unnecessary overhead
- Parallel marking threads to provide scalability that can support large applications and large dataset sizes

Relocate Phase

The relocate phase reclaims Java heap space occupied by dead objects while compacting the heap. The relocation phase selects memory pages that contain mostly dead objects, reclaims the memory space occupied by them, and relocates any live objects to newly allocated compacted areas. As live objects are relocated from sparse pages, their physical memory resources can be immediately reclaimed and used for new allocations. In each cycle, the amount of data that is relocated and compacted varies according to program behavior and allocation rates. The Zing C4 algorithm requires only one empty memory page to compact the entire heap.

The relocation phase can continue to free and compact memory until it exhausts all dead objects detected by the previous mark phase. The garbage collector starts a new mark phase with enough time to detect more dead objects well ahead of the need for new allocations to use them. To maintain compaction efficiency the collector focuses on relocating mostly sparse pages, and starts new mark phases when the supply of sparse pages starts to run low.

Figure 5 Live objects are copied out of sparsely populated or "mostly dead" pages



The relocate phase can be summarized as follows:

- The GC protects 'from' pages, and uses the LVB to support lazy remapping by triggering on access to references on 'from' pages
- The GC relocates any live objects to newly allocated 'to' pages
- The algorithm maintains forwarding pointers outside of the 'from' pages
- Physical memory is immediately reclaimed for use in further compaction or allocation, but virtual 'from' space cannot be recycled until all references to allocated objects are remapped

Figure 5 illustrates the impact on the heap when objects are moved from sparsely populated pages into compacted areas. Forwarding pointers are also saved to enable object references to be remapped to the new object locations.

After a memory page has been scrubbed by the relocate algorithm:

- All live objects have been copied to a new memory page
- A temporary array contains pointers to the relocated objects
- Physical memory is reclaimed and is available to feed further compaction or allocation (in a process called 'quick release'), but the virtual memory addresses of the objects are still alive because pointers to those virtual addresses have not been updated
- The page is flagged to indicate that it is in process with garbage collection and any attempts to access its contents should cause a trap

Details of the Relocate Phase

Whenever the collector decides to relocate a set of objects in this phase, it "GC protects" the pages where the objects reside using a virtual memory protection feature. Once the pages are protected, the collector can relocate all live objects to new memory locations. The collector maintains "forwarding pointer" data, tracking each relocated object's original and new address for the duration of the relocation phase. When all live objects have been relocated and forwarding pointer data has been established, the physical memory resources of the relocated pages can be released and used for new allocations.

As each batch of pages is protected in preparation for relocation, all application threads are notified and their stacks are scanned for references to objects in the newly relocated pages. All objects referred to by such references are immediately relocated, and the references are remapped to point to the new object locations. Once this thread stack is complete, it becomes impossible for the application threads to make a reference to a relocated page without being intercepted by the LVB.

The LVB is also used to detect and correct attempts to load references to relocated objects during the relocate and remap phases. The read barrier instruction checks the loaded reference's value against a Translation Look-aside Buffer (TLB), and if the reference points to a page that is GC protected (being relocated), it will generate a fast GC trap.

The trap code corrects the cause of the trap by replacing the reference with a remapped reference that points to the object's new location. This remapping is performed using the "forwarding pointer" data developed during object relocation. The trap code will correct the source memory location for the object reference from which it was loaded, ensuring that it now refers to the object's new location. It will then return to normal application thread execution with the application observing only the safely remapped reference.

In the early part of a page's relocation, if a read barrier is triggered in an application thread before the referenced object has been relocated, the trap code will cooperatively relocate the object rather than wait for the collector to complete relocating the entire page. This behavior ensures that application threads will not wait for an entire page to be relocated before continuing their own execution. By using the GC protection read barrier and fast trapping mechanism, the collector overcomes the concurrent relocation problem described previously. The collector can now safely relocate objects, compact pages, and free dead object memory without stopping application execution. By correcting the cause of the trap in the source memory location (again, possible only with a read barrier that intercepts the source address), the GC protection trap has a "self-healing" effect, since the same object reference will not re-trigger traps. This ensures a finite and predictable amount of work in a concurrent relocation phase.

The "self-healing" trap behavior, coupled with the marker's use of parallel making threads, gives the relocation phase a highly deterministic quality, helping to ensure that it will consistently release and compact memory without falling behind and resorting to a long garbage collection pause. The relocation phase can be terminated whenever the collector determines a new mark phase is needed. This occurs when the collector decides to detect more dead objects and sparse pages so that it can keep up with the application's object allocation rate. A remap phase that completes the remapping of references to relocated objects is overlapped with the next mark phase.

Remap Phase

The remap phase is responsible for completing the remapping of all relocated object references. It ensures that references to the old location of previously relocated objects do not remain in the Java heap. Such references can still exist at the beginning of the remap phase, as the heap may contain object references that were never visited by application threads after their target objects were relocated. When the remap phase is complete, the GC page protection and virtual memory resources associated with relocated pages can be safely released since live references to them will no longer exist. The forwarding pointer data is also released, as it is no longer needed. The remap phase performs its task by scanning all live objects in the heap, and remapping them if they point to a relocated object. This scan completely overlaps with the mark phase's live object scan, and the two phases are performed together. For each new garbage collection cycle, the mark phase is combined with the previous cycle's remap phase to form a combined mark/remap phase that is executed in a single pass.

Figure 6: GC Tuning Parameter Example – Conventional JVM

```
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:+UseCMSInitiatingOccupancyOnly -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 …
```

GC Tuning Parameters – Zing

```
Java -Xmx40g
```

As each reference is traversed in the combined mark/remap phase, the mark phase marks objects as live and sets the hardware emulated NMT bit to the "marked through value." At the same time, the remap phase looks for references to previously relocated objects, and remaps them to point to the new object locations. Throughout the mark/remap phase, the GC protection read barrier continues to protect the concurrently executing application threads from encountering references to relocated objects.

GC Tuning and JVM Tuning

For most JVMs, extensive GC and JVM tuning is required. A large Java heap size leads to business-interruptive GC pauses, while a small heap can lead to out-of-memory errors and crashes. Preventing these issues normally entails tweaking a variety of JVM parameters to balance goals for maximum pause time, throughput and memory footprint. Using Zing, the need for JVM and GC tuning are minimized through use of the C4 garbage collector and elastic memory, which allows memory to be shared on-demand between Java instances. Examples of GC tuning parameters for a conventional JVM and for Zing are shown above.

Benefits

By utilizing Azul's unique C4 garbage collection technology, Zing provides:

- Predictable, consistent garbage collection (GC) behavior
- Predictable, consistent response times
- Smooth scalability to over 300 GB memory heap sizes
- Unmatched Java application scalability
- Greatly simplified deployments with fewer, larger instances
- Reduced garbage collection (GC) tuning and JVM tuning

Conclusion

Azul Systems has created an efficient, high-performance garbage collection mechanism that provides both concurrent and parallel collection. The collector provides consistent and contained application response times across a wide range of dynamic workloads. These benefits are coupled with unmatched Java application scalability and immediate response to changes in load.

By eliminating garbage collection pauses, Zing enables existing Java applications to smoothly and reliably scale to dozens of CPU cores and hundreds of gigabytes of heap memory. The result is orders of magnitude increases in throughput to withstand massive usage spikes with consistently fast response times, improving service availability and overall user experience. Available as easy-to-deploy software, Zing brings scalability, reliability and elasticity benefits to all Java deployments.

Contact Us

To discover how the Zing can make your Java deployments more scalable, more elastic, more reliable, and less complex, contact:

Azul Systems®
1.650.230.6500
info@azulsystems.com
www.azulsystems.com

Copyright © 2014 Azul Systems, Inc. 1173 Borregas Ave, Sunnyvale California, USA