



Pauseless Garbage Collection

WHITEPAPER

IMPROVING APPLICATION SCALABILITY AND PREDICTABILITY
VERSION 2.0

Azul Systems, Inc.
July 2008
AWP-005-020

Copyright © 2008 Azul Systems, Inc. All rights reserved.



EXECUTIVE OVERVIEW

This paper describes the Azul[®] garbage collection algorithm and its benefits as compared to traditional garbage collection approaches.

Using a combination of software techniques and hardware-assisted features, the Azul garbage collector allows existing and future Java applications to scale to tens of gigabytes of memory without degrading response-time or application throughput. The parallel nature of Azul collector allows a single application instance to scale to tens and hundreds of processor cores without sacrificing efficiency.

INTRODUCTION

“Garbage collection,” a time consuming task that is required in virtual machine based applications, has been the subject of innovative optimization strategies in recent years. Much of these improvements focus on the Java™ platform but the underlying principles promise to be applicable to other application virtual machine technologies such as the Microsoft® .Net™ framework or the SAP® Netweaver™ environments.

These improvements, however, have not kept pace with the continued growth in application scale and memory footprint. Long pause times, poor throughput, and inconsistent response are real problems as Java applications push the practical limits of today's garbage collected runtime environments.

As a result, taking an instance of a Java application beyond the “comfort zone” of sub-gigabyte heaps and a handful of processors is an approach that many IT organizations consider highly desirable but do not believe is possible. Reduced efficiency, heavy cross-JVM communication, increased overall memory footprint, and vastly underutilized servers are all results of the common inability to run applications with multi-GB heaps in practical environments. In addition, data center operations and application developers often resort to peculiar techniques to live with the problem. Examples of this include forced system reboots, complex memory management, and shifting the load to the database tier.

The strong correlation between heap size and garbage collection pause time is one of the main limitations to Java application scalability. Application architecture and deployment often reflects this limitation and attempts to work around it. Deploying applications across many JVM instances and limiting the size of each is a common technique that tries to keep response times and garbage collection pause time within acceptable levels.

Increases in transaction loads cause a corresponding increase in the rate at which garbage is generated (memory heap is consumed.) This is a common additional scaling limitation. Many garbage collectors cannot keep up with the increase in application throughput. The workaround typically involves artificially limiting Java instances to a handful of processors in an attempt to avoid dramatic efficiency degradation. Such workarounds often come at great expense in the form of increased complexity and wasted resources.

By using network attached processing from Azul Systems, existing servers and operating systems can transparently tap the benefits of “pauseless garbage collection.” Java applications and application servers running on these servers can immediately gain scale and improve response times.

Azul Systems[®] pauseless garbage collection uses a combination of software techniques and unique hardware capabilities to directly address these problems. The garbage collector is parallelized, runs concurrently with the application, and is deterministic. Java applications can now scale to tens of gigabytes of memory and tens or hundreds of processors without degrading response-time, application throughput, or efficiency. Complex workaround and architectural restrictions are no longer necessary.

This paper describes the Azul garbage collection algorithm and its benefits as compared to traditional garbage collection approaches.

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.

CHANGING GARBAGE COLLECTION CAPABILITIES AND REQUIREMENTS - WHY DO GARBAGE COLLECTORS HAVE PAUSES AND SCALING PROBLEMS?

Java and J2EE™ server environments have changed dramatically over the last decade. Today's applications are deployed in multi-core, multi-GB 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.

Benefit Scales with CPU Keeps up with garbage creation Larger heap sizes Same response time as load increases	Parallel	Parallel, Stop the World	Parallel, Concurrent
Challenge Provides no benefit in a single processor environment			
Benefit Simple, efficient on single CPU	Single Thread	Single Thread, Stop the World	Single Thread, Concurrent
Challenge Does not scale Limits application scale and drops in efficiency on multiprocessor systems			
		Stop the World	Concurrency
Benefit	Challenge	Benefit	Challenge
Simple, efficient on single CPU	Pauses Pause times grow with heap size Pause frequency grows with load multiprocessor systems	Predictable response times	Requires significant resources to achieve effective implementation

Figure 1 Benefits and challenges of parallelism and concurrency

Although single CPU, small heap systems were prevalent in early Java application deployments, today's applications push the limits of 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 current and future 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. There are 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.

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. 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 in the Azul VM is unique in its ability to concurrently mark the entire heap in a single pass.

CONCURRENT RELOCATION

Objects in the 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 perform heap compaction during 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. Among commercially available JVMs, the Azul GC collector 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.

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)

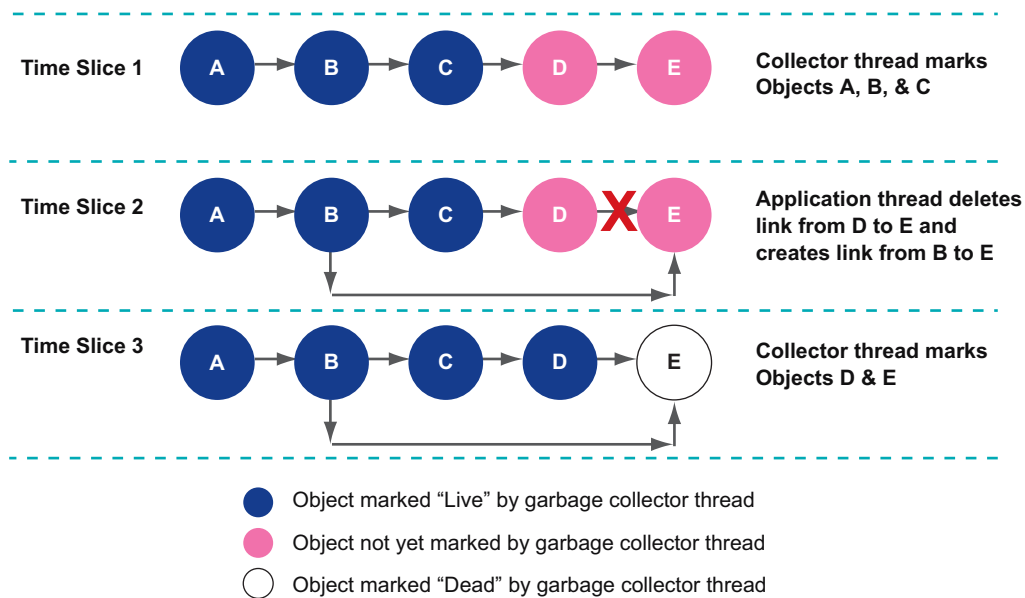


Figure 2 Hidden objects can occur unless the collector thread is made aware of changed references

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 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 Azul pauseless garbage collector performs fully concurrent marking which always completes in a single pass. Azul uses hardware assisted 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.

What makes this problem especially difficult is that there is no easy way 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.

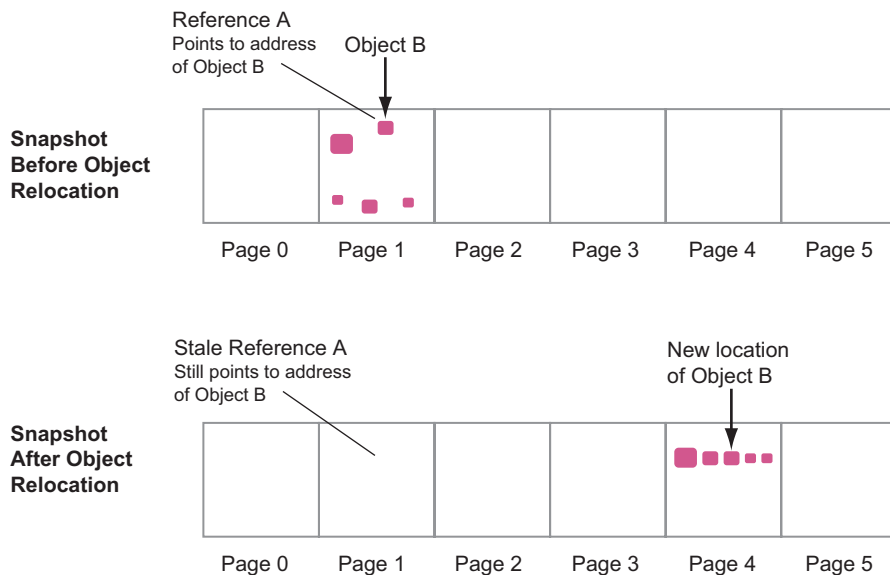


Figure 3 Stale references must be updated before application threads try to access invalid data

Without concurrently relocation 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 hard enough that 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 Azul Pauseless Garbage collector performs fully concurrent object relocation, moving and compacting objects in the heap as part of its normal course of operation. Azul uses hardware assisted barriers to intercept any attempts by the application to use stale pointers referring to old locations of relocated objects, and fixing those stale object pointers as they are encountered such that they point to the correct location. All this is done without pausing the application.

PAUSELESS GARBAGE COLLECTION IN THE AZUL VIRTUAL MACHINE

The Azul garbage collection algorithm relies on a fairly sophisticated read barrier to solve both the concurrent marking and concurrent relocation problems. The read barrier is directly supported by Azul hardware, since it is prohibitively expensive to perform a similar task in software. The Azul algorithm is both concurrent and parallel. It also has the highly desirable quality of continually compacting and defragmenting the heap.

While the Azul garbage collection algorithm actually requires zero pauses in applications, the actual 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 Azul algorithm completely decouples pause time from memory size, allowing applications to scale and use large amounts of memory without repercussions on user response time. The result is a highly scalable environment that offers predictable performance for large Java applications.

The Azul algorithm 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.

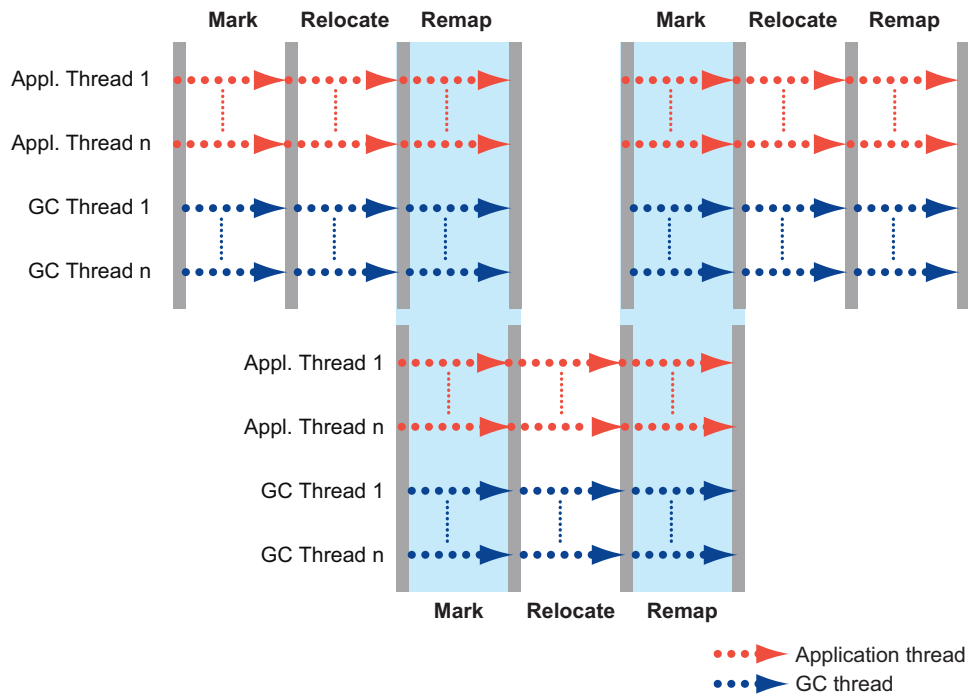


Figure 4 The Azul garbage collection cycle

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 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” and can be safely discarded, and their memory space 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, is used to designate 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 a read barrier.

Azul processors include a read barrier instruction 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 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 hardware assisted NMT 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 it's 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 Azul garbage collection enable the mark phase to have a highly deterministic quality. An entire heap (large or small) can consistently be marked without falling behind and resorting to long garbage collection pauses because Azul garbage collection 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 data set sizes

RELOCATE PHASE

The relocate phase reclaims 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 such 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 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 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.

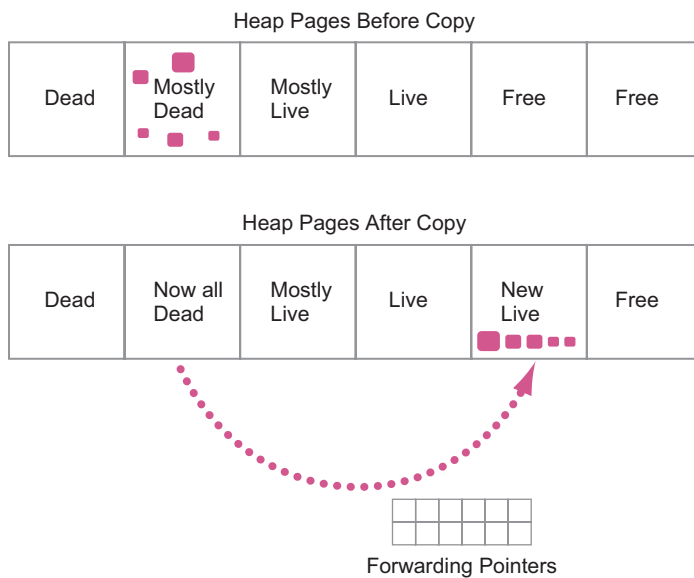


Figure 5 Live objects are copied out of sparsely populated or “mostly dead” pages

After a memory page has been scrubbed by the relocate algorithm, the following situation exists:

- All live objects have been copied to a new memory page
- A temporary array contains pointers to the relocated objects
- Physical memory is reclaimed, but 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 hardware assisted 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 a read barrier.

The same read barrier instruction used by application threads to detect attempts to load “not marked through” references during the mark phase is 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 hardware 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 the object reference was loaded from, 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 hardware assisted 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 marking 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 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.

As each reference is traversed in the combined mark/remap phase, the mark phase marks objects as live and sets the 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.

CONCLUSION

By utilizing a synergistic hardware and software approach, Azul Systems has created an efficient and safe 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 virtual machine scalability and immediate response to changes in load. The extensive read-barrier support provided by Azul hardware platform makes this type of collector practical. Such a collector would be overwhelmingly expensive on other existing processor architectures and platforms.

Network attached processing technology makes this unique garbage collection capability available to new and existing Java and J2EE programs running on all major operating system platforms and processor architectures. All major application server platforms can execute unmodified using Azul technology and readily benefit from the predictability, consistency, and scalability of the solution.

ABOUT AZUL SYSTEMS

Azul Systems[®] has pioneered the industry's first network attached processing solution designed to enable unbound compute resources for Java and J2EE based enterprise applications. Azul compute appliances eliminate capacity planning at the application level and much of the cost and complexity associated with the conventional delivery of computing resources. More information about Azul Systems can be found at www.azulsystems.com.

Copyright © 2008, Azul Systems, Inc. All rights reserved. Azul Systems and Azul are registered logos in the United States and other countries. The Azul arch logo, Compute Pool Manager, and Vega are trademarks of Azul Systems Inc. in the United States and other countries. Linux is a registered trademark of Linus Torvalds. RedHat is the property of Red Hat, Inc. Sun, Sun Microsystems, Solaris, J2EE, J2SE, Java and all Java based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. Microsoft and Windows are registered trademarks of Microsoft Corporation in the United States and/or other countries. Other marks are the property of their respective owners and are used here only for identification purposes. Products and specifications discussed in this document may reflect future versions and are subject to change by Azul Systems without notice.

