Real-time Java, Part 4: **Real-time garbage collection**

Benjamin Biron  
Ryan Sciampacone  
May 02, 2007

Nondeterministic pauses in traditional garbage collection (GC) have inhibited Java™ technology from being a suitable environment for real-time (RT) development. Metronome GC -- part of IBM® WebSphere® Real Time -- provides deterministic GC behavior that, when combined with other features, enables developers to write hard RT applications in the Java language. The authors describe the approach that Metronome uses for deterministic GC, technical issues involved in developing Metronome, and the tools and facilities available for tuning GC.

[View more content in this series](#)

### Real-time systems and garbage collection

Real-time (RT) application development distinguishes itself from general-purpose application development by imposing time restrictions on parts of the runtime behavior. Such restrictions are typically placed on sections of the application such as an interrupt handler, where the code responding to the interrupt must complete its work in a given time period. When hard RT systems, such as heart monitors or defense systems, miss these deadlines, it's considered a catastrophic failure of the entire system. In soft RT systems, missed deadlines can have adverse effects -- such as a GUI not displaying all results of a stream it's monitoring -- but don't constitute a system failure.

In Java applications, the Java Virtual Machine (JVM) is responsible for optimizing the runtime behavior, managing the object heap, and interfacing with the operating system and hardware. Although this management layer between the language and the platform eases software development, it introduces a certain amount of overhead into programs. One such area is GC, which typically causes nondeterministic pauses in the application. Both the frequency and length of the pauses are unpredictable, making the Java language traditionally unsuitable for RT application development. Some existing solutions based on the Real-time Specification for Java (RTSJ) let developers side step Java technology's nondeterministic aspects but require them to change their existing programming model.

Metronome is a deterministic garbage collector that offers bounded low pause times and specified application utilization for standard Java applications. The reduced bounded pause times result from an incremental approach to collection and careful engineering decisions that include...
fundamental changes to the VM. Utilization is the percentage of time in a particular time window that the application is permitted to run, with the remainder being devoted to GC. Metronome lets users specify the level of utilization an application receives. Combined with the RTSJ, Metronome enables developers to build software that is both deterministic with low pause times and pause-free when timing windows are critically small. This article explains the limitations of traditional GC for RT applications, details Metronome's approach, and presents tools and guidance for developing hard RT applications with Metronome.

**Traditional GC**

Traditional GC implementations use a stop-the-world (STW) approach to recovering heap memory. An application runs until the heap is exhausted of free memory, at which point the GC stops all application code, performs a garbage collect, and then lets the application continue.

Figure 1 illustrates traditional STW pauses for GC activity that are typically unpredictable in both frequency and duration. Traditional GC is nondeterministic because of the amount of effort required to recover memory depends on the total amount and size of objects that the application uses, the interconnections between these objects, and the level of effort required to free enough heap memory to satisfy future allocations.

**Figure 1. Traditional GC pauses**

![Figure 1. Traditional GC pauses](image)

**Why traditional GC is nondeterministic**

You can understand why GC times are unbounded and unpredictable by examining a GC's basic components. A GC pause usually consists of two distinct phases: the mark and sweep phases. Although many implementations and approaches can combine or modify the meanings of these phases, or enhance GC through other means (such as compaction to reduce fragmentation within the heap), or make certain phases operate concurrently with the running application, these two concepts are the technical baselines for traditional GC.

The mark phase is responsible for tracing through all objects visible to the application and marking them as live to prevent them from having their storage reclaimed. This tracing starts with the root set, which consists of internal structures such as thread stacks and global references to objects. It then traverses the chain of references until all (directly or indirectly) reachable objects from the root set are marked. Objects that are unmarked at the end of the mark phase are unreachable by the application (dead) because there's no path from the root set through any series of references to find them. The mark phase's length is unpredictable because the number of live objects in an application at any particular time and the cost of traversing all references to find all live objects in the system can't be predicted. An oracle in a consistently behaving system could predict time...
requirements based on previous timing characteristics, but the accuracy of these predictions would be an additional source of nondeterminism.

The sweep phase is responsible for examining the heap after marking has completed and reclaiming the dead objects' storage back into the free store for heap, making that storage available for allocation. As with the mark phase, the cost of sweeping dead objects back into the free memory pool can't be completely predicted. Although the number and size of live objects in the system can be derived from the mark phase, both their position within the heap and their suitability for the free memory pool can require an unpredictable level of effort to analyze.

Traditional GC suitability for RT applications

RT applications must be able to respond to real-world stimuli within deterministic time intervals. A traditional GC can't meet this requirement because the application must halt for the GC to reclaim any unused memory. The time taken for reclamation is unbounded and subject to fluctuations. Furthermore, the time when the GC will interrupt the application is traditionally unpredictable. The time during which the application is halted is referred to as pause time because application progress is paused for the GC to reclaim free space. Low pause times are a requirement for RT applications because they usually represent the upper timing bound for application responsiveness.

Metronome GC

Metronome’s approach is to divide the time that consumes GC cycles into a series of increments called quanta. To accomplish this, each phase is designed to accomplish its total work in a series of discrete steps, allowing the collector to:

1. Preempt the application for very short deterministic periods.
2. Make forward progress in the collection.
3. Let the application resume.

This sequence is in contrast to the traditional model where the application is halted at unpredictable points, the GC runs to completion for some unbounded period of time, and GC then quiesces to let the application resume.

Although splitting the STW GC cycle into short bounded pauses helps reduce GC's impact, this isn't sufficient for RT applications. For RT applications to meet their deadlines, a sufficient portion of any given time period must be devoted to the application; otherwise, the requirements are violated and the application fails. For example, take a scenario where GC pauses are bounded at 1 millisecond: If the application is allowed to run for only 0.1 millisecond between every 1-millisecond GC pause, then little progress will be made, and even marginally complex RT systems will likely fail because they lack time to progress. In effect, short pause times that are sufficiently close together are no different from a full STW GC.

Figure 2 illustrates a scenario where the GC runs for the majority of the time yet still preserves 1-millisecond pause times:
Utilization

A different measure is required that, in addition to bounded pause times, provides a level of determinism for the percentages of time allotted to both the application and GC. We define application utilization as the percentage of time allotted to an application in a given window of time continuously sliding over the application's complete run. Metronome guarantees that a percentage of processing time is dedicated to the application. Use of the remaining time is at the GC's discretion: it can be allotted to the application or it can be used by the GC. Short pause times allow for finer-grained utilization guarantees than a traditional collector. As the time interval used for measuring utilization approaches zero, an application's expected utilization is either 0% or 100% because the measurement is below the GC quantum size. The guarantee for utilization is made strictly on measurements the size of the sliding window. Metronome uses quanta of 500 microseconds in length over a 10-millisecond window and has a default utilization target of 70%.

Figure 3 illustrates a GC cycle divided into multiple 500-microsecond time slices preserving 70% utilization over a 10-millisecond window:

In Figure 3, each time slice represents a quantum that runs either the GC or the application. The bars below the time slices represent the sliding window. For any sliding window, there are at most 6 GC quanta and at least 14 application quanta. Each GC quantum is followed by at least 1 application quantum, even if the target utilization would be preserved with back-to-back GC quanta. This ensures the application pause times are limited to the length of 1 quantum. However, if target utilization is specified to be below 50%, some instances of back-to-back GC quanta will occur to allow the GC to keep up with allocation.

Figures 4 and 5 illustrate a typical application-utilization scenario. In Figure 4, the region where utilization drops to 70% represents the region of an ongoing GC cycle. Note that when the GC is inactive, application utilization is 100%.
Figure 4. Overall utilization

Figure 5. GC cycle utilization

Section A of Figure 5 is a staircase graph where the descending portions correspond to GC quanta and the flat portions correspond to application quanta. The staircase demonstrates the GC respecting low pause times by interleaving with the application, producing a step-like descent toward the target utilization. Section B consists of application activity only to preserve utilization targets across all sliding windows. It's common to see a utilization pattern showing GC activity only at the beginning of the pattern. This occurs because the GC runs whenever it is allowed to (preserving pause times and utilization), and this usually means it exhausts its allotted time at the beginning of the pattern and allows the application to recover for the remainder of the time window. Section C illustrates GC activity when utilization is near the target utilization. Ascending portions represent application quanta, and descending portions are GC quanta. The sawtooth nature of this section is again because of the interleaving of the GC and application to preserve low pause times. Section D represents the portion after which the GC cycle has completed. This section's ascending
nature illustrates the fact that the GC is no longer running and the application will regain 100% utilization.

The target utilization is user-specifiable in Metronome; you can find more information in this article's Tuning Metronome section.

**Running an application with Metronome**

Metronome is designed to provide RT behavior to existing applications. No user code modification should be required. Desired heap size and target utilization must be tuned to the application so target utilization maintains the desired application throughput while letting the GC keep up with allocation. Users should run their applications at the heaviest load they want to sustain to ensure RT characteristics are preserved and application throughput is sufficient. This article's Tuning Metronome section explains what you can do if throughput or utilization is insufficient. In certain situations, Metronome's short pause-time guarantees are insufficient for an application's RT characteristics. For these cases, you can use the RTSJ to avoid GC-incurred pause times.

**The Real-time Specification for Java**

The RTSJ is a "specification for additions to the Java platform to enable Java programs to be used for real-time applications." Metronome must be aware of certain aspects of the RTSJ -- in particular, RealtimeThreads (RT threads), NoHeapRealtimeThreads (NHRTs), and immortal memory. RT threads are Java threads that, among other characteristics, run at a higher priority than regular Java threads. NHRTs are RT threads that can't contain references to heap objects. In other words, NHRT-accessible objects can't refer to objects subject to GC. In exchange for this compromise, the GC won't impede the scheduling of NHRTs, even during a GC cycle. This means NHRTs won't incur any pause times. Immortal memory provides a memory space that's not subject to GC; this means NHRTs are allowed to refer to immortal objects. These are only some aspects of the RTSJ; see Related topics for a link to the complete specification.

**Technical issues involved in deterministic GC**

Metronome uses several key approaches within the J9 virtual machine to achieve deterministic pause times while guaranteeing GC's safety. These include arraylets, time-based scheduling of the garbage collector, processing of root structures for tracing live objects, coordinating between the J9 virtual machine and GC to ensure all live objects are found, and the mechanism used for suspending the J9 virtual machine for a GC quantum.

**Arraylets**

Although Metronome achieves deterministic pause times through breaking the collection process up into incremental units of work, allocation can cause hiccups in the GC in some situations. One area is in the allocation of large objects. For most collector implementations, the allocation subsystem keeps a pool of free heap memory, consumed by the application through allocating objects and replenished by the collector through sweeping. After the first collection, free heap memory is primarily the result of objects that were once live but are now dead. Because there's no predictable pattern to how or when these objects die, the resulting free memory on the heap is a collection of fragmented chunks of varying sizes, even if coalescence of adjacent dead objects takes place. Further, each collection cycle can return a different pattern of free chunks. As a result,
the allocation of a sufficiently large object can fail if no free chunk of memory is large enough to satisfy the request. Typically, these large objects are arrays; standard objects are generally no larger than a few dozen fields, often resulting in less than 2K in size for most JVMs.

To alleviate the fragmentation issue, some collectors implement a compaction, or defragmentation, phase to their collection cycle. After the sweep is complete, if an allocation request can't be met, the system tries to move existing live objects around in the heap in an effort to coalesce two or more free chunks into a single larger chunk. This phase is sometimes implemented as an on-demand feature, embedded into collector's fabric (semispace collectors being an example), or in an incremental fashion. Each of these systems has its trade-offs, but generally the compaction phase is an expensive one in terms of time and effort.

The current version of Metronome in WebSphere Real Time does not implement a compaction system. To prevent fragmentation from being a problem, Metronome uses arraylets, which breaks the standard linear representation up into several discrete pieces that can be allocated independently of one another.

Figure 6 shows that array objects appear as a spine -- which is the central object and only entity that can be referenced by other objects on the heap -- and a series of arraylet leaves, which contain the actual array contents:

**Figure 6. Arraylets**

The arraylet leaves are not referenced by other heap objects and can be scattered throughout the heap in any position and order. The leaves are of a fixed size to allow simple calculation of element position, which is an added indirect. As Figure 6 illustrates, memory-use overhead that's because of internal fragmentation in the spine has been optimized by including any trailing data for a leaf into the spine.

Note that this format can mean that an array spine can grow to unbounded sizes, but this hasn't yet been found to be a problem in the existing system.

**Scheduling a GC quantum**

To schedule deterministic pauses for GC, Metronome uses two different threads to achieve both consistent scheduling and short, uninterrupted pause times:
• The **alarm thread**. To schedule a GC quantum deterministically, Metronome dedicates the alarm thread to act as the heartbeat mechanism. The alarm thread is a very high priority thread (higher than any other JVM thread in the system) that wakes up at the same rate as the GC quantum time period (500 microseconds in Metronome) and is responsible for determining whether or not a GC quantum should be scheduled. If so, the alarm thread must suspend the running JVM and wake the GC thread. The alarm thread is active for a very short period (typically under 10 microseconds) and should go unnoticed by the application.

• The **GC thread**. The GC thread performs the actual work during a GC quantum. The GC thread must first complete the suspension of the JVM that the alarm thread initiated. It can then perform GC work for the remainder of the quantum, scheduling itself back to sleep and resuming the JVM when the quantum end time approaches. The GC thread can also preemptively sleep if it can't complete its upcoming work item before the quantum end time. In relation to the RTSJ, this thread's priority is higher than all RT threads except NHRTs.

### Cooperative suspend mechanism

Although Metronome uses a series of small incremental pauses to complete a GC cycle, it must still suspend the JVM for every quantum in a STW fashion. For each of these STW pauses, Metronome uses the **cooperative suspend mechanism** in the J9 virtual machine. This mechanism doesn't rely on any special native-thread capability for suspending threads. Rather, it uses an asynchronous-style messaging system to notify Java threads that they must release their access to internal JVM structures, including the heap, and sleep until they are signaled to resume processing. Java threads within the J9 virtual machine periodically check if a suspend request has been issued, and if so, they proceed as follows:

1. Release any held internal JVM structures.
2. Store any held object references in well-described locations.
3. Signal the central JVM suspend mechanism that it has reached a **safe point**.
4. Sleep and wait for a corresponding **resume**.

Upon resumption, threads reread object pointers and reacquire the JVM-related structures they previously held. The act of releasing JVM structures lets the GC thread process these structures in a safe fashion; reading and writing to partially updated structures can cause unexpected behavior and crashes. By storing and then reloading object pointers, the threads allow the GC the opportunity to update the object pointers during a GC quantum, which is necessary if the object is moved as part of any compaction-like operation.

Because the suspend mechanism cooperates with Java threads, it's important that the periodic checks in each thread be spaced apart with the shortest possible intervals. This is the responsibility of both the JVM and Just-in-time (JIT) compiler. Although checking for suspend requests introduces an overhead, it allows structures such as stacks to be well defined in terms of the GC's needs, letting it determine accurately whether or not values in stacks are pointers to objects.

This suspend mechanism is used only for threads currently participating in JVM-related activities; non-Java threads, or Java threads that are out in Java Native Interface (JNI) code and not using the JNI API, are not subject to being suspended. If these threads participate in any JVM activities,
such as attaching to the JVM or calling the JNI API, they will cooperatively suspend until the GC quantum is complete. This is important because it lets threads that are associated with the Java process continue to be scheduled. And although thread priorities will be respected, perturbing the system in any noticeable way in these other threads can affect the GC’s determinism.

Write barriers

Full STW collectors have the benefit of being able to trace through object references and JVM internal structures without the application perturbing the links in the object graph. By splitting the GC cycle into a series of small STW phases and interleaving its execution with the application’s, Metronome does introduce a potential problem in keeping track of the live objects in a system. Unexpected behavior or crashes can occur because the application, after processing an object, can modify the object’s references such that unprocessed objects are hidden from the collector. Figure 7 illustrates the hidden-object problem:

**Figure 7. Hidden-object problem**

Assume an object graph exists in the heap as described in Figure 7 by section I. The Metronome collector is active and is scheduled to perform tracing work in this quantum. In its allotted time period, it manages to trace through the root object as well as the object that it references, before running out of time and needing to schedule the JVM back in section II. During the application run, the references between the objects are changed such that object A now points to an unprocessed object, which is no longer referred to by any other location in section III. The GC is then scheduled back in for another quantum and continues to process, missing this hidden object pointer. The result is that during the sweep phase of the GC that returns unmarked objects to the free list, a live object will be reclaimed, resulting in a dangling pointer, causing incorrect behavior or even crashes in the JVM or GC.

To prevent this type of error, the JVM and Metronome must cooperate in tracking changes to the heap and JVM structures such that the GC will keep all relevant objects alive. This is achieved through a write barrier, which tracks all writes to objects and records the creating and breaking of references between objects so that the collector can track potential hidden live objects. The type of
barrier that Metronome uses is called a **snapshot at the beginning** (SATB) barrier. It conceptually records the heap's state at the beginning of a collection cycle and preserves all live objects at that point as well as those allocated during the current cycle. The concrete solution involves a Yuasa-type barrier (see Related topics), where the overwritten value in any field store is recorded and treated as if it had a root reference associated with it. Preserving a slot's original value before overwriting enables the live object set to be preserved and processed.

This type of barrier processing is also required for internal JVM structures, including the JNI Global Reference list. Because the application can add and remove objects from this list, a barrier is applied to track both removed objects to avoid a hidden-object problem (similar to a field overwrite) and added objects to eliminate the need to rescan the structure.

**Root scanning and processing**

To begin tracing through live objects, garbage collectors start from a set of initial objects obtained from roots. Roots are structures within the JVM that represent hard references to objects that the application creates either explicitly (for example, JNI Global References) or implicitly (for example, stacks). Root structures are scanned as part of the initial function of the mark phase in the collector.

Most roots are malleable to some degree during execution in terms of their object references. For this reason, changes to their reference set must be tracked, as we discussed in **Write barriers**. However, certain structures, such as the stack, can't afford the tracking of pushes and pops without significant penalties incurred on performance. Because of this, certain limitations and changes to scanning stacks are made for Metronome in keeping with the Yuasa-style barrier:

- **Atomic scanning of stacks.** Individual thread stacks must be scanned atomically, or within a single quantum. The reason for this is that during execution, a thread can pop any number of references from its stack -- references that could have been stored elsewhere during execution. Pausing at mid-scan of a stack could cause stores to be lost track of or missed during two partial scans, creating a dangling pointer within the heap. Application developers should be aware that stacks are scanned atomically and should avoid using very deep stacks in their RT applications.

- **Fuzzy barrier.** Although a stack must be scanned atomically, it would be difficult to keep determinism if all stacks were scanned during a single quantum. The GC and JVM are allowed to interleave execution while scanning Java stacks. This could result in objects being moved from one thread to another through a series of loads and stores. To avoid losing references to objects, threads that have not been scanned yet during a GC have the barrier track both the overwritten value and the value being stored. Tracking the stored object, should it be stored into an already processed object and popped off the stack, preserves reachability through the write barrier.

**Tuning Metronome**

It's important to understand the correlation between heap size and application utilization. Although high target utilization is desirable for optimal application throughput, the GC must be able to keep up with the application allocation rate. If both the target utilization and allocation rate are
high, the application can run out of memory, forcing the GC to run continuously and dropping the utilization to 0% in most cases. This degradation introduces large pause times often unacceptable for RT applications. If this scenario is encountered, a choice must be made to decrease the target utilization to allow for more GC time, increase the heap size to allow for more allocations, or a combination of both. Some situations might not have the memory required to sustain a certain utilization target, so decreasing the target utilization at a performance cost is the only option.

Figure 8 illustrates a typical trade-off between heap size and application utilization. A higher utilization percentage requires a larger heap because the GC isn't allowed to run as much as a lower utilization would allow.

**Figure 8. Heap size versus utilization**

![Heap size versus utilization](image)

The relationship between utilization and heap size is highly application dependent, and striking an appropriate balance requires iterative experimentation with the application and VM parameters.

**Verbose GC**

Verbose GC is a tool that logs and outputs GC activity to a file or screen. You can use it to determine if the parameters (heap size, target utilization, window size, and quantum time) support the running application. Listing 1 shows an example of verbose output:

**Listing 1. Verbose GC sample**

```xml
<?xml version="1.0" ?>
<verbosegc version="200702_15-Metronome">
  <gc type="synchgc" id="1" timestamp="Tue Mar 13 15:17:18 2007" intervalms="0.000">
    <details reason="system garbage collect" />
    <duration timems="30.023" />
    <heap freebytesbefore="535265280" />
    <heap freebytesafter="535838720" />
    <immortal freebytesbefore="15591288" />
    <immortal freebytesafter="15591288" />
    <synchronousgcpriority value="11" />
  </gc>
  <gc type="trigger start" id="1" timestamp="Tue Mar 13 15:17:45 2007" intervalms="0.000" />
</verbosegc>
```
Each Verbose GC event is contained within a `<gc></gc>` tag. Various event types are available, but the most common are included in Listing 1. A `synchgc` type represents a synchronous GC, which is a GC cycle that ran uninterrupted from beginning to end; that is, interleaving with the application didn't happen. These can occur for two reasons:

- `System.gc()` was invoked by the application.
- The heap filled up, and the application failed to allocate memory.

The reason for the synchronous GC, contained in the `<details>` tag, consists of `system garbage collect` for the first case and `out of memory` for the second. The first case has no bearing on the sustainability of the application with the specified parameters. However, invoking `System.gc()` from the user application causes the application utilization to drop to 0% in many cases and causes long pause times; it should therefore be avoided. But if a synchronous GC occurs because of the second case -- an out-of-memory error -- this means the GC was unable to keep up with the application allocation. Therefore, you should consider increasing the heap or decreasing the application utilization target to avoid the occurrence of synchronous GCs.

`trigger` GC event types correspond to the GC cycle's start and end points. They're useful for delimiting batches of `heartbeat` GC events. `heartbeat` GC event types roll up the information of multiple GC quanta into one summarized verbose event. Note that this is unrelated to the alarm-thread heartbeat. The `quantumcount` attribute corresponds to the amount of GC quanta rolled up in the `heartbeat` GC. The `<quantum>` tag represents timing information about the GC quanta rolled up in the `heartbeat` GC. The `<heap>` and `<immortal>` tags contain information about the free memory at the end of the quanta rolled up in the `heartbeat` GC. The `<gcthreadpriority>` tag contains information about the priority of the GC thread when the quanta began.

The quantum time values correspond to the pause times seen by the application. Mean quantum time should be close to 500 microseconds, and the maximum quantum times must be monitored to ensure they fall within the acceptable pause times for the RT application. Large pause times can occur when the GC is preempted by other processes in the system, preventing it from completing.
its quanta and allowing the application to resume, or when certain root structures in the system are abused and grown to unmanageable sizes (see Issues to consider when using Metronome).

Immortal memory is a resource required by the RTSJ that is not subject to GC. For this reason, it's normal to see the immortal free memory in the verbose GC log drop without ever recovering. It's used for objects such as string constants and classes. You need to be aware of your program's behavior and adjust the size of immortal memory appropriately.

You should monitor heap usage to ensure the general trend remains stable. A downward trend in heap free space would indicate the existence of a potential leak caused by the application. A number of conditions can cause leaks, including ever-expanding hash tables, large resource objects being held indefinitely, and global JNI references not being cleaned up.

Figures 9 and 10 illustrate stable and downward trends in free heap space. Note that local minima and maxima are normal and expected because the free space only increases during a GC cycle and correspondingly decreases when the application is active and allocating.

Figure 9. Stable free heap

![Figure 9. Stable free heap](image-url)
The `<gc>` tag’s `interval` attribute corresponds to the time elapsed since the last verbose GC event of the same type was output. In the case of the `heartbeat` event type, it can represent the time since the `trigger start` event if it's the first heartbeat for the current GC cycle.

**Tuning Fork**

Tuning Fork is a separate tool for tuning Metronome to suit the user application better. Tuning Fork lets the user inspect many details of GC activity either after the fact through a trace log or during run time through a socket. Metronome was built with Tuning Fork in mind and logs many events that can be inspected from within the Tuning Fork application. For example, it displays the application utilization over time and inspects the time taken for various GC phases.

Figure 11 shows the GC performance summary graph generated by Tuning Fork, including target utilization, heap memory use, and application utilization:
Issues to consider when using Metronome

Metronome strives to deliver short deterministic pauses for GC, but some situations arise both in application code and the underlying platform that can perturb these results, sometimes leading to pause-time outliers. Changes in GC behavior from what would be expected with a standard JDK collector can also occur.

The RTSJ states that GC doesn't process immortal memory. Because classes live in immortal memory, they are not subject to GC and therefore can't be unloaded. Applications expecting to use a large number of classes need to adjust immortal space appropriately, and applications that require class unloading need to make adjustments to their programming model within WebSphere Real Time.

GC work in Metronome is time based, and any change to the hardware clock could cause hard-to-diagnose problems. An example is synchronizing the system time to a Network Time Protocol (NTP) server and then synchronizing the hardware clock to the system time. This would appear as a sudden jump in time to the GC and could cause a failure in maintaining the utilization target or possibly cause out-of-memory errors.

Running multiple JVMs on a single machine can introduce interference across the JVMs, skewing the utilization figures. The alarm thread, being a high-priority RT thread, preempts any other lower-priority thread, and the GC thread also runs at an RT priority. If sufficient GC and alarm threads are active at any time, a JVM without an active GC cycle might have its application threads preempted by another JVM’s GC and alarm threads while time is actually taxed to the application because the GC for that VM is inactive.
Related topics

- **Real-time Java series**: Read the other parts in this series.
- "**A real-time garbage collector with low overhead and consistent utilization**" (David F. Bacon, Perry Cheng, and V.T. Rajan, *Proceedings of the 30th Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 2003): This paper presents a dynamically defragmenting collector that overcomes the limitations of applying GC to hard RT systems.
- **JSR 1: Real-time Specification for Java**: You'll find the RTSJ at the Java Community Process site.
- "**IBM WebSphere Real Time V1.0 delivers predictable response times using Java standards**": Read the product announcement for WebSphere Real Time.
- **Metronome**: Learn more about Metronome, the GC technology incorporated in WebSphere Real Time.
- "**High-level Real-time Programming in Java**": Read about the Staccato research prototype.
- **WebSphere Real Time**: WebSphere Real Time lets applications dependent on a precise response times take advantage of standard Java technology without sacrificing determinism.
- **Real-time Java technology**: Visit the authors' IBM alphaWorks research site to find cutting-edge technologies for RT Java.