Garbage collection in WebSphere Application Server V8, Part 2: Balanced garbage collection as a new option

Ryan Sciampacone
Peter Burka
Aleksandar Micic

August 03, 2011

IBM® WebSphere® Application Server V8 introduces the new "balanced" garbage collection policy. This technology is optimized for large heaps and aims to even out pause times that are associated with garbage collection. The information in this article will help you find out if the balanced collector is a good fit for your applications, and explain how you can tune it for maximum performance.

View more content in this series

Introduction

In a Java™ Virtual Machine (JVM), the garbage collector reduces the application developer’s memory management burden. Garbage collection (GC) is an automated system that handles both the allocation and reclamation of memory for Java objects. This helps reduce the complexity of application development, but comes at a cost that can manifest itself as uneven performance through an application’s lifetime, as well as disruptive long pauses that can impact application responsiveness.

Part 1 of this series described the garbage collection policies available in IBM WebSphere Application Server V8 and information for configuring the new default generational policy. This article discusses the balanced garbage collection policy, a new GC technology available in WebSphere Application Server V8, available through the command line option, -Xgcpolicy:balanced. The balanced collector aims to even out pause times and reduce the overhead of some of the costlier operations typically associated with garbage collection, as shown in Figure 1.
The cost of garbage collection can be attributed to a number of factors. Some of the most important are:

- **Total size of live data:** As the number and size of objects that are live (that is, accessible to the application through a series of object references) in the system increases, the cost of tracing, or discovering, those live objects increases.

- **Heap fragmentation:** As objects become reclaimable, the associated memory must be managed to satisfy allocation. The cost associated with tracking this memory, making it available for allocation, and relocating live objects to coalesce free memory, can be high.

- **Rate of allocation:** The speed at which objects are allocated, along with their size, will dictate the consumption rate of free memory, and, in turn, the frequency of garbage collection. An application with a high rate of allocation might experience disruptive GC pauses more frequently.

- **Parallelism:** The garbage collector might use multiple threads for garbage collection. As more threads work to complete the collection in parallel, garbage collection pause times decrease.

A common pattern in recent Java applications is the use of **in-heap data stores**. Examples include in-memory databases or low latency caches, often seen in NoSQL solutions. These deployments typically have a relatively low number of available cores compared to the large amount of memory being managed. Responsiveness in these applications is key -- long GC pauses can interrupt keep-alive heartbeats in data grid systems, leading to false conclusions that nodes within the data grid have failed and forcing a node restart. This causes poor responsiveness, wasted bandwidth as grid nodes are restarted and repopulated, and increased pressure on surviving nodes as they struggle to service increased load.

Existing approaches to garbage collection use powerful techniques to effectively address many of today’s GC pause time issues. However, increasing heap sizes and changes in data patterns are beginning to expose weaknesses in these traditional models.

**Traditional garbage collection**

There are a number of existing solutions to garbage collection. Two of the more prevalent are:

**Whole heap collection:** In this approach, a JVM typically waits until the heap is completely consumed and then performs a garbage collection of the whole heap. The primary cost is directly proportional to the size of the live set of the application. Additional costs can include other global operations, such as whole heap compaction in order to defragment the heap.
Generational: The hypothesis "Objects die young" states that, in typical applications, most allocated objects have an extremely short life span and can be collected shortly after they’ve been allocated. Generational collectors leverage this by designating areas in the heap, typically referred to as new space, that are used for object allocation and are collected by specialized collectors. This seeks the best return-on-investment of time and effort to reduce total pause times relative to whole heap collection. Eventually, as objects survive long enough to leave new space, the remainder of the heap, referred to as old space, fills up and must be collected with a whole heap collector.

Figure 2. New space collection areas and size vs. whole heap (global) collections

There are many other techniques to help alleviate the pressures of pause times in garbage collection, including:

- **Parallelism:** Using multiple GC threads on multi-core systems to complete GC operations faster and reduce pause times.
- **Concurrency:** Performing GC operations as Java threads execute, through a combination of dedicated GC helper threads and by recruiting Java threads to help with the work.
- **Incremental collection:** Reducing the average GC pause time by dividing work across many shorter pauses, eventually completing a full GC. This approach typically trades shorter average pause times for longer total pause time as the bookkeeping required to incrementalize the operation has overhead.

As noted in Part 1 of this series, there are a number of powerful GC technologies available to application developers, depending on the required performance characteristics of a particular deployment.

**Goals of the balanced garbage collector**

The primary goal of the balanced collector is to amortize the cost of global garbage collection across many GC pauses, reducing the effect of whole heap collection times. At the same time, each pause should attempt to perform a self contained collection, returning free memory back to the application for immediate reuse.

To achieve this, the balanced collector uses a dynamic approach to select heap areas to collect in order to maximize the return-on-investment of time and effort. This is similar to the gencon policy approach, but is more flexible as it considers all parts of the heap for collection during each pause, rather than a statically defined new space.
Figure 3. Balanced ability to dynamically select areas of the heap for collections

By removing restrictions on what areas of the heap to collect, the balanced collector is able to dynamically adapt to a wide range of object usage patterns. For example, applications with high object mortality after 3 minutes, or applications which allocate 100MB per transaction, or applications that have high fragmentation at certain object lifetimes, can all be addressed by focusing on the appropriate areas of the heap at the appropriate time.

The balanced collector evens out pause times across GC operations based on the amount of work that is being generated. This can be affected by object allocation rates, object survival rates, and fragmentation levels within the heap. Be aware that this smoothing of pause times is a best effort rather than a real-time guarantee. Pause times are not guaranteed to be bound to a certain maximum nor does the technology provide utilization guarantees.

The balanced collector builds upon the strengths of existing GC technologies within the IBM JDK, including optavgpause, gencon, and metronome. The remainder of this article will describe the general approach to garbage collection taken by the balanced collector, some of the key pieces of technology used to achieve these goals, scenarios under which balanced should be used, and advice on tuning the collector for best results.

How the balanced collector works

This section describes the operation of the balanced collector, first explaining how the heap is organized, and then describing the techniques used by the balanced collector to collect the heap and return free memory to the application.

Heap organization

A fundamental aspect of the balanced collector’s architecture, which is critical to achieving its goals of reducing the impact of large collection times, is that it is a region-based garbage collector. A region is a clearly delineated portion of the Java object heap which categorizes how the associated memory is used and groups related objects together. During the JVM startup, the garbage collector divides the heap memory into equal-sized regions, and these region delineations remain static for the lifetime of the JVM.

Regions are the basic unit of garbage collection and allocation operations. For example, when the heap is expanded or contracted, the memory committed or released will correspond to a number of regions. Although the Java heap is a contiguous range of memory addresses, any region within that range can be committed or released as required. This enables the balanced collector to contract the heap more dynamically and aggressively than other garbage collectors, which typically require the committed portion of the heap to be contiguous.

It is also important to note that GC operations, such as tracing live objects or compacting, operate on a set of regions. Because the heap is partitioned into well defined regions, collection operations...
can act on different sets of regions over time as the balanced collector analyzes the available data about the heap.

Objects in a single region share certain characteristics, such as all being of similar age (Figure 4). In particular, newly allocated objects (objects that have been allocated since the last GC cycle) are all kept together in so-called **eden regions**. Eden regions are noteworthy because they are always included in the next collection cycle.

Regions impose a maximum object size. Objects are always allocated within the bounds of a single region. Arrays which cannot fit within a single region are represented using a discontiguous format known as **arraylets**, which is described later. With the exception of arraylets, objects are never permitted to span regions.

The region size is always a power of two (for example, 512KB, 1MB, 2MB, 4MB, and so on). The region size is selected at startup based on the maximum heap size. The collector chooses the smallest power of two which will result in less than 2048 regions, with a minimum region size of 512KB. Except for small heaps (less than about 512MB) the JVM aims to have between 1024 and 2047 regions.

**Figure 4. Region structure and characteristics found in the object heap**

Collecting the heap

Like the gencon collector, the balanced collector leverages the observation that recently allocated objects are likely to quickly become garbage. It targets newly created objects with a stop-the-world cycle, meaning all Java threads are suspended from execution while the GC proceeds. This is supported by a more global-oriented operation to help handle objects outside of the set of eden regions (which are always collected).

There are three different types of garbage collection cycles (Figure 5):

- **Partial garbage collection (PGC)** collects a subset of regions known as the **collection set**. PGCs are the most common GC cycles when using the balanced collector. It is used to collect garbage from regions with a high mortality rate and to collect and defragment regions outside of the eden set.

- **Global mark phase (GMP)** incrementally traces all live objects in the heap. GMPs are common operations, but are normally less frequent than PGCs. GMP cycles are not responsible for returning free memory to the heap or compacting fragmented areas of the heap (those are responsibilities of PGCs). The role of the GMP is primarily to support the PGC by refining the information used to determine which regions of the heap are best suited for different collection operations.

- **Global garbage collection (GGC)** marks, sweeps, and compacts the whole heap in a stop-the-world fashion. GGC is normally used only when the collector is explicitly invoked through
a System.gc() call by the application. It can also be performed in very tight memory condition as a last resort to free up memory before throwing an OutOfMemoryError. A primary goal of the balanced collector is to avoid global garbage collects; unless explicitly invoked, these should be viewed as problems in the tuning of the balanced collector.

Know that although PGCs are self-contained stop-the-world operations, GMP cycles can span many increments, and are performed partially concurrently while the Java application runs. The goal of each PGC cycle is to reclaim memory -- it will choose regions in the heap to collect, perform the collection, and return the resulting free memory back to the application for allocation purposes. Figure 5 shows a typical time line describing the mode of operation of the balanced collector.

**Figure 5. Time-line representation of typical balanced collector behavior**

Understanding the role of a partial garbage collection

Each partial garbage collection (PGC) is responsible for ensuring that enough free memory is available for the application to continue. To do this, every PGC selects a number of regions to include in the collection set. There are three major factors in deciding whether a region belongs in the collection set (Figure 6):

- Eden regions, which contain objects allocated since the previous PGC, are always included, primarily due to their generally high mortality rate. This also enables the GC to analyze and record metadata concerning the reference graph and liveness demographics of these objects.
- Regions discovered by the GMP phase that are highly fragmented and would help return free memory to the application if they were collected, compacted, and coalesced. This is known as defragmentation. Continuous defragmentation is required to create free regions for use as eden regions.
- Regions outside of the eden region set which are expected to have a high mortality rate. The GC gathers statistics on average mortality of objects as a function of object age. Based on these statistics each PGC dynamically selects regions in which it expects to find sufficient garbage (and consequently free memory) relative to the amount of work required to discover live objects.

**Figure 6. Collection set selection by region within the object heap**

A PGC typically employs a copying collector similar to that used by the gencon policy. This copying approach requires the PGC to reserve a number of free regions as destinations for live objects.
evacuated from the collection set. Unlike gencon the balanced collector does not pre-allocate memory for the surviving objects. Instead, it gauges the expected survival rate and consumes a subset of free regions in the heap large enough so that all live objects from the collection set are successfully copied.

In application workloads with high variability in allocation and mortality rate the projected size of the survivor area might be larger than what is actually available. In such cases, the PGC will switch from a copying mechanism to an in-place trace-and-compact approach. The compact pass uses sliding compaction, which is guaranteed to successfully complete the operation without requiring any free heap memory, unlike the preferred copying approach.

Besides switching collection modes between PGCs, the balanced collector is also capable of switching modes during a PGC. In cases where the estimate of survivor memory requirements proves insufficient, the copying collector fills up the remaining free space and then dynamically transitions to the in-place approach. Any objects copied before the transition remain copied; objects which were not successfully copied are collected in place.

In all cases, all operations within the PGC are stop-the-world (STW). This means that all Java thread execution is halted while the GC cycle completes.

**Understanding the role of a global mark phase**

PGCs can recover a high percentage of garbage while incurring relatively low pause times through strategic collection set decisions. A PGC does not have global knowledge of heap liveness and the data it uses to make decisions becomes increasingly unreliable over time. The global mark phase (GMP) is responsible for refreshing the view of the entire heap, enabling the PGC to make better decisions about collection set selection and the level of effort needed to keep pace with the application’s heap consumption rate.

A GMP is triggered when the PGC is unable to keep up with live data being injected into the system. Garbage not discovered by PGCs slowly accumulates, and the heap fills up. When the efficiency of PGCs deteriorates, a GMP cycle starts. The GMP marks all live objects in the heap through a combination of parallel STW increments and concurrent processing.

In order to determine when a GMP cycle should be initiated, the GC projects the rate at which the heap is being depleted, the size of the global live set, and the cost of tracing through the live set. Based on this information the GC schedules the initial kick-off point, the number of increments required to complete the GMP, and the amount of work to be done for each increment. This schedule is intended to:

- Minimize the impact on the application.
- Complete the GMP before the heap is completely depleted of free memory. (More precisely, to complete such that sufficient free space remains for PGCs to complete efficiently.)

GMP increments are scheduled approximately half-way between PGC cycles. In addition, if there is available processor time (such as idle cores) at the conclusion of a PGC cycle threads will be dispatched to help complete the GMP increment concurrently while the application executes. If the
concurrent threads are able to complete the work for the next GMP increment before its scheduled point, no additional work will be performed.

**Under the hood: Key mechanisms that help make it all happen**

Now that you understand the core infrastructure and approach used by the balanced collector, you should know about two important mechanisms that support the collector, and enable it to achieve its goals: the remembered set and arraylets.

**The remembered set**

For PGCs to be able to accurately discover all live objects, the collector must scan all object roots (for example, thread stacks, permanent class loaders, JNI references). Additionally, all references to objects within the collection set from objects external to the collection set must be discovered. This could be accomplished by scanning all objects in all regions not included in the collection set, but this would be terribly inefficient. Instead, references between objects in different regions are tracked and recorded in a data structure known as the remembered set. The remembered set is used to track all in-bound references on a per-region basis (Figure 7).

![Figure 7. Basic structure of the remembered set](image)

References are created and discovered during program execution through a write barrier. This process is handled by the JVM and is invisible to the Java application -- no changes to Java code are required.

Off-heap memory is reserved for remembered set storage, and is typically no more than 4% of the current heap size. There are also limits on the number of in-bound references tracked per region. If either the global limit (4%) or the local limit for a region is hit, then any further addition to a region's remembered set causes the region to be flagged as "popular." This renders the region uncollectible by PGCs. It might once again become a candidate for collection after the next GMP cycle, which strips stale information from the remembered set and updates it.

**Arraylets**

Most objects are easily contained within the minimum region size of 512KB. However, some large arrays might require more memory than is available in a single region. In order to support such large arrays the balanced collector uses an arraylet representation for large arrays.

Figure 8 shows that large array objects appear as a spine -- which is the central object and the only entity that can be referenced by other objects on the heap -- and a series of arraylet leaves, which contain the actual array elements:
Figure 8. Basic structure of an arraylet

The arraylet leaves are not directly referenced by other heap objects and can be scattered throughout the heap in any position and order. Each leaf is an entire region, allowing for a simple calculation of element position, and requiring a single additional indirection to reach any element. As Figure 8 illustrates, memory-use overhead due to internal fragmentation in the spine has been optimized by including any trailing data for the final leaf into the spine.

Because array representation is hidden by the JVM, the apparent complexity created by the shape of arraylets is invisible to the Java application. No code modification or knowledge that arraylets are present is required.

There are a number of advantages of using arraylets. Due to heap fragmentation over time, other collector policies might be forced to run a global garbage collection and defragmentation (compaction) phase in order to recover sufficient contiguous memory to allocate a large array. By removing the requirement that large arrays be allocated in contiguous memory, the balanced garbage collector is more likely to be able to satisfy such an allocation without requiring unscheduled garbage collection, and more likely still to be able to do so without a global defragmentation operation. Additionally, the balanced collector never needs to move an arraylet leaf once it has been allocated. The cost of relocating an array is limited to the cost of relocating the spine, so large arrays do not contribute to higher defragmentation times.

Figure 9. Allocation of an array as an arraylet in a fragmented heap

The arraylet representation is only used for very large arrays. Small arrays have the same representation in the balanced collector as they do in the other IBM garbage collectors, such as the gencon collector. There is no additional space overhead for small arrays. However, as JIT compiled code needs to include logic for both small and large arrays in most cases, the use of arraylets might result in larger compiled code.

The most visible consequence of using arraylets is seen in Java Native Interface (JNI) code. The JNI provides APIs called `GetPrimitiveArrayCritical`, `ReleasePrimitiveArrayCritical`, `GetStringCritical`, and `ReleaseStringCritical` which provide direct access to array data where
possible. The balanced collector offers direct access to contiguous arrays, but is required to copy the array data into a contiguous block of off-heap memory for discontiguous arraylets, as the representation of arraylets is inconsistent with the contiguous representation required by these APIs.

If you believe that this is affecting your application, there are a number of possible solutions. First, determine if array data is being copied. The JNI APIs mentioned include an `iscopy` return parameter. If the API sets this to `JNI_TRUE`, then the array was discontiguous and the data was copied. Examine your JNI code to determine if the relevant native functions can be rewritten in Java, or changed to use different APIs, for example `Get<Type>ArrayRegion`. Ensure that any callees of `ReleasePrimitiveArrayCritical` use the `JNI_ABORT` mode if the data is not modified, as this eliminates the need to copy the data back to the Java heap. Finally, a larger region size (controlled by increasing the heap size) could reduce or eliminate arraylets.

### Tuning the balanced collector

The balanced collector uses a similar approach to collection as the gencon collector (as described in *Part 1*) so many of the same techniques are applicable. It is important, however, to enumerate some of the differences that need to be considered:

- The balanced notion of eden space is similar to the gencon new space but not identical. Eden space contains newly created objects that are always involved in the collection process, but, once collected, become part of the general heap. In contrast, objects can remain in the gencon new space across several collections before moving to old space.
- Although the notion of a global tracing phase exists, one of the goals of balanced is to completely avoid regular global collections (in particular global compactions) by incrementally collecting and defragmenting the heap outside of the eden space. This is in contrast to gencon, which focuses only on new space, ultimately leading to global collections when old space is exhausted.
- Long lived objects such as classes and string constants are never allocated in gencon's new space (they are directly allocated into old space). Balanced allocates these objects in eden space and consequently includes them as part of the collection cycle.

The basic tuning options for balanced are the same as gencon, with eden space replacing the tunable new space. To recap:

- `-Xmn<size>` sets the recommended size of eden space, effectively setting both `-Xmns` and `-Xmnx`.
- `-Xmns<size>` sets the recommended initial size of eden space to the specified value.
- `-Xmnx<size>` sets the recommended maximum size of eden space to the specified value.

Be aware that the options are recommendations; balanced will adhere to the options as long as the system is capable of accommodating any restrictions. For example, if there is insufficient heap memory for the recommended eden size, balanced will reduce the eden space to a size that is achievable. The default recommended eden size is 1/4 of the current heap size.
The primary goal of tuning eden space should be to contain the objects allocated for a set of transactions within the system. Since most systems have many transactions in flight (due to multi-threaded processing) eden space should be able to accommodate all of these transactions at any time. From a tuning perspective, this means that the amount of data surviving from eden space in a system under regular load should be significantly less than the actual size of eden space itself. As a general rule, for optimal performance the amount of data surviving from eden space in each collection should be kept to approximately 20% or less. Some systems might be able to tolerate parameters outside these boundaries, based on total heap size or number of available GC threads in the system. The key is to use these guidelines as a starting point when deploying applications. In general, any -Xmn setting that was used for the gencon policy is applicable to balanced as well.

Although you can freely size eden space with -Xmn, there are some important points to keep in mind when doing so:

- **Keeping eden size small:** By sizing eden space down to the bare minimum based on transaction size and turn around time, your system may be pausing for GC more frequently than it needs to, reducing performance. Also, if the eden size is too tightly tuned, any change in behavior (for example, taking on more work from a suddenly unavailable system in order to satisfy high availability requirements) will exceed the eden space and result in sub-optimal performance.

- **Keeping eden size large:** By sizing the eden space higher, you reduce the number of collection pauses (potentially increasing performance), but you reduce the amount of memory available to the general heap. If the general heap is too small relative to the total live set, it could force the balanced collector to incrementally collect and defragment large portions of the heap in each PGC cycle in an effort to keep up with demand, resulting in long GC pauses and sub-optimal performance.

In all cases, there are trade-offs to be made, and the tuning process is systematic and iterative:

1. Adjust heap maximum (-Xmx) and initial (-Xms) memory as needed.
2. Run the application under a normal stress load.
3. Gather verbose GC logs (-verbose:gc, -Xverbosegclog:<filename>) for analysis
4. Use appropriate tooling to determine adjustments to eden size (-Xmn), if necessary.
5. Return to step 2 until performance is satisfactory.

Although verbose GC logs can be inspected manually, the volume of data can often be overwhelming. The Garbage Collection and Memory Visualizer (GCMV) is a tool that consumes verbose GC logs, visualizes the results, and provides analysis of the data (Figure 10). GCMV reports what went right, what went wrong, what needs attention, and, most importantly, provides recommendations to help improve performance.
If you prefer to get your hands dirty, and in some cases mine information from the logs that might not be readily available through GCMV, direct inspection of the verbose GC logs is an acceptable avenue to problem determination and tuning. An example of verbose GC stanzas appear in Listing 1, which describe a typical PGC cycle.

**Listing 1. Example verbose GC stanza**

```xml
<exclusive-start id="137" timestamp="2011-06-22T16:18:32.453" intervalms="3421.733">
  <response-info timems="0.146" idlems="0.104" threads="4" lastid="0000000000D97A00"
    lastname="XYZ Thread Pool : 34" />
</exclusive-start>

<allocation-taxation id="138" taxation-threshold="671088640"
  timestamp="2011-06-22T16:18:32.454" intervalms="3421.689" />
<cycle-start id="139" type="partial gc" contextid="0" timestamp="2011-06-22T16:18:32.454"
  intervalms="3421.707" />
<gc-start id="140" type="partial gc" contextid="139" timestamp="2011-06-22T16:18:32.454">
  <mem-info id="141" free="8749318144" total="10628366336" percent="82">
    <mem type="eden" free="0" total="671088640" percent="0" />
    <numa common="10958264" local="1726662224" non-local="0" non-local-percent="0" />
    <remembered-set count="352640" freebytes="422080000" totalbytes="424901120"
      percent="99" regionsoverflowed="0" />
  </mem-info>
</gc-start>

<allocation-stats totalBytes="665373480">
  <allocated-bytes non-tlh="2591104" tlh="662765360" arrayletleaf="0" threadName="WXYConnection[192.168.1.1,port=1234]"
    threadId="0000000000C6ED00" bytes="1408341176" />
</allocation-stats>
<gc-op id="142" type="copy forward" timems="71.024" contextid="139">
  <memory-copied type="eden" objects="171444" bytes="103905272" bytesdiscarded="5289504" />
  <memory-copied type="other" objects="75450" bytes="96864448" bytesdiscarded="4600472" />
  <memory-cardclean objects="88738" bytes="5422432" />
  <remembered-set-cleared processed="338248" cleared="53760" durationms="3.018" />
  <references type="soft" candidates="2" cleared="0" enqueued="0" dynamicThreshold="28"
    maxThreshold="32" />
  <references type="weak" candidates="1" cleared="0" enqueued="0" />
</gc-op>

<gc-op id="143" type="classunload" timems="0.021" contextid="139">
  <classunload-info classloadercandidates="178" classloadersunloaded="0"
    quiescems="0.000" setupms="0.188" scanms="0.000" postms="0.001" />
</gc-op>
<gc-end id="144" type="partial gc" contextid="139" durationms="72.804">
  <mem-info id="145" free="9311354880" total="10628366336" percent="87" />
</gc-end>
```
When to use the balanced collector

The balanced collector is a suitable replacement for the older policies, including gencon, if the environment and the needs of the application fit with the associated trade-offs. In general, balanced is recommended if:

- The application is running on a **64 bit platform** and deploys a **heap greater than 4GB**. Balanced is best suited to reducing large global GC pause times on large heaps. Due to the overhead associated with the techniques used in balanced, small heaps might not provide ideal deployment scenarios.
- The application gets excellent results with gencon, but still experiences **occasional excessively long global GC pauses**, including long global compaction times. Balanced might offer slightly higher average pause times than gencon, but maintains the advantages of gencon by focusing on newly created objects, and will ultimately work to avoid the cost of global collections by incrementally collecting and compacting the global heap. Note that global collections and compactions might also be overly frequent in other GC policies due to the allocation of large objects, in particular arrays.
- The application is **willing to accept a slight degradation in performance**. As noted earlier, the technical approach of balanced is much more complicated than gencon or other GC policies, and, as such, the cost of GC is higher both in terms of pause times and Java application overhead. Although this overhead is often no more than 10%, it still represents a measurable trade off, in particular when comparing against gencon.

### NUMA support

Non-uniform memory access (NUMA) is a hardware architecture in which the processors and memory are organized into groups called nodes. Processors can access memory local to their own node faster than memory associated with other nodes. NUMA is available on System x® and System p® using current versions of AIX®, Linux® or Windows®.

**Figure 11. Sample NUMA configuration**

The balanced collector organizes memory and garbage collection work to take advantage of NUMA. On startup, the collector binds each heap region to one of the system’s NUMA nodes.
The heap is divided as evenly as possible so that all nodes have approximately the same number of regions.

Most threads in the JVM are bound, or affinitized, to NUMA nodes. Threads are bound in a simple round-robin manner: the first thread is bound to the first node, the second thread to the second node, and so forth. Some threads, such as the main thread and threads attached through JNI, might not be bound to a specific node.

As threads allocate objects they attempt to place those objects in their local memory for faster access. If there is insufficient local memory, the thread can borrow memory from other nodes. Although this memory is slower, it is still preferable to an unscheduled GC or an OutOfMemoryError.

During garbage collection, parallel GC threads prefer work associated with their own node. This results in shorter pauses. The collector also moves objects so that each one is stored in memory local to the thread using the object, whenever possible. This can result in improved application throughput.

Some operating systems provide utilities to control which nodes a process can use (for example, numactl on Linux or execrset on AIX). The JVM will detect that a subset of nodes are available and will only use the resources permitted. NUMA support in the JVM can be disabled using the -Xnuma:none command line option.

There are also a number of performance or behavioral benefits that a Java application may be able to leverage, but which are not necessarily first class reasons to move to balanced:

- **The application makes heavy use of class unloading**, either through reflection or other facilities. Unlike gencon, which relies on global collections for class unloading and string constant garbage collection, the balanced collector is capable of collecting class loaders and string constants if they are in the given collection set, including short lived, newly created objects. This is an advantage of the balanced collector as it is able to collect these objects and their associated native memory structures quickly, reducing total memory pressure and overhead.

- **The application is deployed on large hardware** (large amounts of memory; large numbers of cores) which balanced is better able to leverage. The balanced collector will also recognize NUMA systems, deploy GC threads and Java threads, and allocate objects in the heap, in a manner which leverages the differences in memory speeds. In addition, with large numbers of cores, balanced makes use of concurrency (GC work proceeds while Java threads are running) by deploying helper threads to expedite the collection process.

- **The application makes use of very large arrays** which the balanced collector stores using a discontiguous representation. The balanced collector is able to process these arrays more efficiently during garbage collection, and can also avoid compacting to allocate large arrays.

Finally, there are a few points to consider when evaluating balanced that might mean it isn’t appropriate to deploy:

- **Balanced is not a real time garbage collector**; if real-time results are required, you should use IBM WebSphere Real Time for AIX or Linux. Although the balanced collector strives to smooth out GC pause times, it ultimately does not guarantee a maximum pause time or that pauses will be completely uniform. The work generated by the application will dictate both the frequency and length of the GC pauses.
Because balanced uses slightly more heap memory based on how the heap is organized internally, it will not work well with heaps that are already very full (>90% occupancy). Due to the incremental nature of the global collection process, and the possibility of micro fragmentation in regions, it is possible that heaps which are extremely full will not experience good pause time behavior with balanced as it struggles to process the entire heap in relatively short periods of time -- effectively making it act as a global collector such as optthruput or optavgpause.

Because balanced uses a discontiguous representation for very large arrays, JNI access to such arrays might be slower than in other collectors. If your application makes extensive use of JNI and large arrays, you might be able to improve performance by increasing the heap size sufficiently so that the arrays become contiguous. You might also be able to modify your native code to reduce its reliance on large arrays. If neither of these are possible, balanced might not be suitable for your deployment.

Conclusion

This article introduced the balanced GC technology available via the `-Xgcpolicy:balanced` option in the Java Virtual Machine for IBM WebSphere Application Server V8. The technology is intended to reduce disruptive long pause times incurred by global collections and compactions by incrementalizing the whole heap collection process and incorporating it into a generational collection mechanism. Although the technology has some drawbacks, such as higher average pause times, it can prove extremely useful in a variety of deployment scenarios where relatively consistent pause times are desired. Due to technological advancements, such as incremental class unloading, balanced could be suitable for other deployments as well based on need.
Related topics

- Garbage collection in WebSphere Application Server V8, Part 1: Generational as the new default policy
- Wikipedia: Garbage collection definition
- Java technology, IBM style: Garbage collection policies, Part 1
- Java technology, IBM style: Garbage collection policies, Part 2
- Download the IBM SDKs for Java
- IBM Monitoring and Diagnostic Tools for Java - Health Center
- IBM Monitoring and Diagnostic Tools for Java - Garbage Collection and Memory Vizualiser
- WebSphere Application Server V8 product information
- Download WebSphere Application Server V8 trial version

© Copyright IBM Corporation 2011
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)