Java technology, IBM style, Part 1: Garbage collection policies
Differing policies provide flexibility

Mattias Persson

One of the great benefits of the Java™ platform is that it takes care of much of the work of garbage collection for you, but there are occasions when you still want to tweak the way garbage collection takes place. With the latest Java technology implementation from IBM®, you can choose among several garbage collection policies to help you get the most out of your application. In this second article in the series Java technology, IBM style, Java developer Mattias Persson explores the available options and details the situations in which each might be appropriate.

View more content in this series

Garbage collection (GC) in the IBM Developer Kit for the Java 5.0 Platform (IBM SDK) can be configured using four different policies. This article, the first of two parts on GC, introduces the different garbage collection policies and discusses their general characteristics. You should have a basic understanding of garbage collection in the Java platform before you begin. Part 2 presents a quantitative approach to choosing a policy, along with with some examples.

About the series
The Java technology, IBM style series takes a look at the latest releases of the IBM implementations of the Java platform. You'll learn how IBM has implemented some of the advances built into version 5.0 of the Java platform, and you'll find out how to use some of the value-added features built into new releases from IBM.

Contact the authors individually with comments or questions about their articles. To comment on the series as a whole, you may contact series lead Chris Bailey. For more on the concepts discussed here and links where you can download the latest IBM releases, see Related topics.

Why different GC policies?
The availability of differing GC policies gives you increased capabilities. There are many different algorithms available for GC, each of which has advantages and drawbacks, depending on the type of workload. (If you are unfamiliar with the general topic of GC algorithms, see Related topics.)
for further reading.) In the IBM SDK 5.0, you can configure the garbage collector with one of four policies, each of which uses its own algorithm. The default policy is sufficient for most applications. If you do not have any particular performance requirements for your application, then the content of this article (and the next) may not be interesting to you; you can run the IBM SDK 5.0 without changing the GC policy. However, if your application requires optimal performance or if you are generally concerned with the length of GC pauses, then read on. You will see that the latest release from IBM has even more choice than its predecessors.

So why doesn't the IBM implementation of the Java runtime make the choice automatically for you? This isn't always possible. It's difficult for the runtime to understand your needs. In some situations, you may want to run the application with high throughput; in others, you might prefer to reduce the pause times.

Table 1 lists the available policies and explains when you should use each one. The following sections describe the characteristics of each policy in more detail.

**Table 1. GC policies in the IBM SDK 5.0**

<table>
<thead>
<tr>
<th>Policy</th>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimize for throughput</td>
<td>-Xgcpolicy:optthruput (optional)</td>
<td>The default policy. It is typically used for applications where raw throughput is more important than short GC pauses. The application is stopped each time that garbage is collected.</td>
</tr>
<tr>
<td>Optimize for pause time</td>
<td>-Xgcpolicy:optavtpause</td>
<td>Trades high throughput for shorter GC pauses by performing some of the garbage collection concurrently. The application is paused for shorter periods.</td>
</tr>
<tr>
<td>Generational concurrent</td>
<td>-Xgcpolicy:gencon</td>
<td>Handles short-lived objects differently than objects that are long-lived. Applications that have many short-lived objects can see shorter pause times with this policy while still producing good throughput.</td>
</tr>
<tr>
<td>Subpooling</td>
<td>-Xgcpolicy:subpool</td>
<td>Uses an algorithm similar to the default policy's but employs an allocation strategy that is more suitable for multiprocessor machines. We recommend this policy for SMP machines with 16 or more processors. This policy is only available on IBM pSeries® and zSeries® platforms. Applications that need to scale on large machines can benefit from this policy.</td>
</tr>
</tbody>
</table>

**Some terms defined**

- **Throughput** is the amount of data processed by an application. Throughput must be measured with an application-specific metric.
- **Pause time** is the duration of time in which the garbage collector has paused all application threads to collect the heap.

In this article, I use the abbreviations from the command-line options detailed in Table 1 to denote these policies: `optthruput` for Optimize for throughput, `optavtpause` for Optimize for pause time, `gencon` for Generational concurrent, and `subpool` for Subpooling.
When should I consider a non-default GC policy?

I advise that you always start with the default GC policy. Before you move away from the default, you need to understand the circumstances under which you should explore an alternative policy. Table 2 presents a set of reasons that could apply:

**Table 2. Reasons for switching to an alternative GC policy**

<table>
<thead>
<tr>
<th>Switch to</th>
<th>Reasons</th>
</tr>
</thead>
</table>
| optavgpause | • My application cannot tolerate the length of the GC pauses. A degradation in performance is acceptable as long as the GC pause time is reduced.  
• I am running on a 64-bit platform and use a very large heap -- more than 3 or 4GB.  
• My application is a GUI application and I'm concerned about the user response times. |
| gencon | • My application allocates many short-lived objects.  
• The heap space is fragmented.  
• My application is transaction-based (that is, objects in the transaction don't survive beyond the transaction commit). |
| subpool | • I have scalability problems on large multiprocessor machine. |

Let me stress that the reasons mentioned in Table 2 are not sufficient to conclude that the alternative policies will perform better; they are simply hints. In all cases, you should run the application and measure throughput and/or response times in combination with GC pause times. The next part of this series shows examples of this sort of testing.

The remaining sections of this article describe in more detail the differences between the GC policies.

**optthruput**

**Signs of heap fragmentation**

The most common sign of heap fragmentation is the presence of premature allocation failures. This is seen in verbose garbage collection (`-Xverbose:gc`) as GC being triggered even though there is free space on the heap. Another sign is the presence of small thread allocation caches (see the subsection "Heap lock and thread allocation caches"). You can use `-Xtgc:freelist` to determine the average size. Both the options are explained in the IBM SDK 5 Diagnostics Guide available in Related topics.

**optthruput** is the default policy. It is a tracing collector, referred to as a mark-sweep-compact collector. The mark and the sweep phases always run during GC, but compaction only happens under certain circumstances. The mark phase finds and marks all live objects. The sweep phase removes all unmarked objects. A third and optional step is compaction. There are various circumstances under which compaction might occur; the most common one is the system's inability to reclaim enough free space.

Fragmentation occurs when objects are allocated and freed so often that only small chunks of free space are left in the heap. There may be a large amount of free space overall in the heap, but the contiguous regions are small, causing allocation failures. Compaction moves all the objects down
to the beginning of the heap and aligns them so that no space exists between them. This removes fragmentation from the heap but is an expensive task, so it is only performed if necessary.

Figure 1 shows an outline of the heap layout after the different phases: mark, sweep, and compaction. The dark regions represent objects and the bright regions are free space.

**Mark and sweep**

The *mark* phase traverses all objects that can be referenced from thread stacks, statics, interned strings, and JNI references. As part of this process, we create a mark bit vector that defines the beginning of all live objects.

The *sweep* phase uses the mark bit vector generated by the mark phase to identify the chunks of heap storage that can be reclaimed for future allocations; these chunks are added to the free list.

**Figure 1. Heap layout before and after garbage collection**

The details of how the different GC phases work is beyond the scope of this article; my focus is on ensuring you understand the runtime characteristics. I encourage you to read the Diagnostics Guide (see Related topics) for more information.

Figure 2 illustrates how the execution time is distributed between the application thread (or *mutator*) and GC threads. The horizontal axis is elapsed time and the vertical axis contains the threads, where *n* represents the number of processors on the machine. For this illustration, assume that the application is using one thread per processor. GC is illustrated by the blue boxes, which show stopped mutators and GC threads running. These collections consume 100 percent of the CPU's resources, and the mutator threads are idle. The figure overgeneralizes a bit so that we can compare this policy to other policies in this article. In actuality, GC duration and frequency vary depending on application and workload.
Heap lock and thread allocation caches

The **optthruput** policy uses a contiguous heap area that all threads in the application share. A thread needs exclusive access to the heap for reserving space for new objects. This lock, called the **heap lock**, ensures that only one thread at a time can allocate an object. On machines with multiple CPUs, this lock can cause scaling problems because multiple allocation requests can occur at the same time, but each needs to have exclusive access to the heap lock.

To reduce this problem, each thread reserves a small chunk of memory called a **thread allocation cache** (also referred to as **thread local heap**, or TLH). This piece of storage is exclusive to a thread, so while allocating from it, the heap lock is not used. When the allocation cache is full, the thread goes back to the heap and asks for a new one, using the heap lock.

Fragmentation of the heap can prevent the thread from getting large TLHs, so a TLH fills quickly, causing the application thread to come back frequently to the heap for a new one. In this situation, the heap lock becomes a bottleneck; in such situations, the **gencon** or **subpool** policies can provide good alternatives.

**optavgpause**

For many applications, throughput is not as important as fast response time. Consider an application that processes work items that cannot take longer than 100 milliseconds to complete. With GC pauses in the 100 millisecond range, you get items that cannot be processed within this timeframe. A problem with garbage collection is that the pause time increases the maximum time it takes to process an item. Large heap sizes (available on 64-bit platforms) increase this effect because more objects are processed.

**optavgpause** is an alternative GC policy designed to keep pauses to a minimum. It does not guarantee a particular pause time, but pauses are shorter than those produced by the default GC
policy. The idea is to perform some garbage collection work concurrently while the application is running. This is done in two places:

- **Concurrent mark and sweep**: Before the heap is filled up, each mutator helps out and mark objects (concurrent mark). There is still a stop-the-world GC, but the pause is significantly shorter. After GC, the mutator threads help out and sweep objects (concurrent sweep).

- **Background GC thread**: One (or more) low-priority background GC threads perform marking while the application is idle.

There is a throughput performance penalty of 5 to 10 percent over the default GC policy, depending on the application.

Figure 3 illustrates how the execution time can be divided between GC threads and mutator threads using `optavgpause`. The background tracing thread is not shown because it should not affect application performance.

**Figure 3. Distribution of CPU time between mutators and GC threads in the optavgpause policy**

The gray areas in the figure imply that concurrent tracing is enabled and that each mutator thread has to give up some of its processing time. Each concurrent phase is followed by a full garbage collection that completes the marking and sweeping that did not happen during the concurrent phase. The pause that results from this should be much shorter than the normal GC seen in `optthruput`, as indicated by a smaller box on the timescale in Figure 3. The gap between the end of the GC and the start of the concurrent phase vary, but during this phase, there is no significant performance impact.

**gencon**

A generational garbage collection strategy considers the lifetime of objects and places them in separate areas of the heap. In this way, it tries to overcome the drawbacks of a single heap in applications where most objects *die young* -- that is, where they do not survive many garbage collections.

With generational GC, objects that tend to survive for a long time are treated differently from short-lived objects. The heap is split into a *nursery* and a *tenured area*, as illustrated in Figure 4. Objects are created in the nursery and, if they live long enough, are promoted to the tenured area. Objects
are promoted after having survived a certain number of garbage collections. The idea is that most objects are short-lived; by collecting the nursery frequently, these objects can be freed up without paying the cost of collecting the entire heap. The tenured area is garbage collected less often.

Figure 4. New and old area in gencon GC

As you can see in Figure 4, the nursery is in turn split into two spaces: allocate and survivor. Objects are allocated into the allocate space and, when that fills up, live objects are copied into the survivor space or into the tenured space, depending on their age. The spaces in the nursery then switch use, with allocate becoming survivor and survivor becoming allocate. The space occupied by dead objects can simply be overwritten by new allocations. Nursery collection is called a scavenge; Figure 5 illustrates what happens during this process:

Figure 5. Example of heap layout before and after GC

When the allocate space is full, garbage collection is triggered. Live objects are then traced and copied into the survivor space. This process is really inexpensive if most of the objects are dead. Furthermore, objects that have reached a copy threshold count are promoted into the tenured space. The object is then said to be tenured.

As the name Generational concurrent implies, the gencon policy has a concurrent aspect to it. The tenured space is concurrently marked with an approach similar to the one used in the optavgpause policy, except without concurrent sweep. All allocations pay a small throughput tax during the concurrent phase. With this approach, the pause time incurred from the tenure space collections is kept small.

Figure 6 shows how the execution time maps out when running gencon GC:
A scavenge is short (shown by the small red boxes). Gray indicates that concurrent tracing starts followed by a collection of the tenured space, some of which happens concurrently. This is called a global collection, and it includes both a scavenge and a tenure space collection. How often a global collection occurs depends on the heap sizes and object lifetimes. The tenured space collection should be relatively quick because most of it has been collected concurrently.

**subpool**

The subpool policy can help increase performance on multiprocessor systems. As I mentioned earlier, this policy is available only on IBM pSeries and zSeries machines. The heap layout is the same as that for the optthruput policy, but the structure of the free list is different. Rather than having one free list for the entire heap, there are multiple lists, known as subpools. Each pool has an associated size by which the pools are ordered. An allocation request of a certain size can quickly be satisfied by going to the pool with that size. Atomic (platform-dependent) high-performing instructions are used to pop a free list entry off the list, avoiding serialized access. Figure 7 shows how the free chunks of storage are organized by size:

When the JVMs start or when a compaction has occurred, the subpools are not used because there are large areas of the heap free. In these situations, each processor gets its own dedicated mini-heap to satisfy requests. When the first garbage collection occurs, the sweep phase starts populating the subpools, and subsequent allocations mainly use subpools.
The `subpool` policy can reduce the time it takes to allocate objects. Atomic instructions ensure that allocations happen without acquiring a global heap lock. Mini-heaps local to a processor increase efficiency because cache interference is reduced. This has a direct effect on scalability, especially on multiprocessor systems. On platforms where `subpool` is not available, generational GC can provide similar benefits.

**Conclusion**

This article has highlighted the different GC policy choices available in the IBM SDK 5.0 and some of their characteristics. The default policy is sufficient for most applications; however, in some situations, other policies perform better. I have described some general scenarios in which you should consider switching to `optavgpause`, `gencon`, or `subpool`. Measuring application performance when evaluating a policy is very important, and Part 2 demonstrates this evaluation process in more detail.
Related topics

- "A Parallel, incremental and concurrent GC for servers," Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, Avi Owshanko (IBM Haifa Research Laboratory, 2002): In-depth information about concurrent GC.
- **Garbage collection (computer science)**: Wikipedia's article on the subject is a good starting point.
- **IBM Java 5.0 Development Package for Eclipse**: For use with the popular open source IDE.
- **IBM Java SDKs**: Download the SDKs for AIX, Linux, and z/OS, among other IBM developer kits for Java technology, from this page.

© Copyright IBM Corporation 2006
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)