SAPinsider

This article appeared in the Jul▪Aug▪Sep 2005 issue of *SAP Insider* and appears here with permission from the publisher, Wellesley Information Services (WIS), **www.WISpubs.com**.

WIS

Taking Out the Trash: Avoid Performance Bottlenecks from Java Garbage Collection

Regular Feature

Performance and Data Management Corner

Susanne Janssen, SAP AG

Rudolf Meier, SAP AG

The integration of Java into the SAP landscape has reaped considerable benefits — standardized integration of Web services technology and third-party solutions, and a global, leading-edge community of developers — but also has introduced to the SAP world a whole new set of programming and performance challenges. One of these is Java garbage collection.

For many years, Java developers had difficulty writing effective applications that required high throughput or minimal pause time during execution because of Java's automatic garbage collection (GC). Garbage collection runs at regular intervals to free up memory by removing unwanted or outdated programming objects, but at the same time has the nasty habit of stopping all application threads on the same virtual machine, leading to increased wait time and diminished user productivity.

For non-Java professionals, this article will introduce you to the influence of garbage collection on CPU consumption. For Java developers, the article will familiarize you with our recently conducted research on predicting the impact of garbage collection on system performance,[1] as well as provide you with some important guidelines for efficient programming, which we will introduce at the end of this article.

Garbage Collection's Performance Slowdown

How well a system performs from an end user's perspective is dominated by the time that elapses between screen changes or, in more technical terms, *interaction response time*. If not managed properly, Java's garbage collection can increase interaction response time and wear on users' patience.

Imagine an end user doing something very simple, perhaps keying a job applicant's home address into an e-recruiting scenario and then trying to save this entry. All of a sudden nothing happens — the system just waits. Although the user doesn't know it, their CPU is blocked because of garbage collection. Imagine that your CPU is like a small street, and every so often a huge garbage collection truck comes to take away the rubbish. All the cars that come in behind the truck have to stop. That's just how garbage collection works in Java.

A garbage collection run in Java can lead to increased wait times for users and increased memory consumption of the underlying software components, as shown in **Figure 1**. Even if your system has a fairly low number of requests coming in, these requests can quickly back up and come to a very steep peak because of GC blocking all other processes.

A Manageable Condition

Over the years, garbage collection in Java has improved, with the increasing number of Java-based business applications and the development of garbage collection strategies by different virtual machines. An example of such a strategy is minor GCs, as opposed to the full GCs described earlier. Think of a minor GC as a smaller garbage truck that allows a few cars to pass by during collection time.

However, there are still some cumbersome features of garbage collection

[1] For more detailed information on our research, please visit our Web log at www.sdn.sap.com/sdn/weblogs.sdn?blog=/pub/wlg/1631.

that affect the response time of Java applications, even at relatively low overall CPU utilization. For system administrators and developers alike, it is very important to understand exactly what causes this behavior and to take preventive measures from a development and tuning perspective.

> ✔ **Note!**
> To understand the performance behavior of any given application, you must use different models to simulate real system behavior, and then validate these models against measurements of actual coding. In our research, we observed performance bottlenecks in Java applications, and then stepped back to analyze the source of what we believed impeded the performance. We then tested our model against reality.

Modeling Response Time in a Java Environment

In all software applications, the response time per user interaction increases with the utilization of the CPU. For applications based on ABAP — the language of the majority of applications in the SAP world — we can apply relatively simple Markovian queuing models[2] to predict response time behavior. These models attempt to show interdependencies between CPU utilization, response times, services times, and other factors.

When we add Java into the equation, our research demonstrates that garbage collection must be factored in when tallying up performance. When we first included stop times caused by garbage collection into our performance model, however, we found significant deviations when we tried to validate it. Therefore, stopping threads are not the only contributor to potentially high response times.

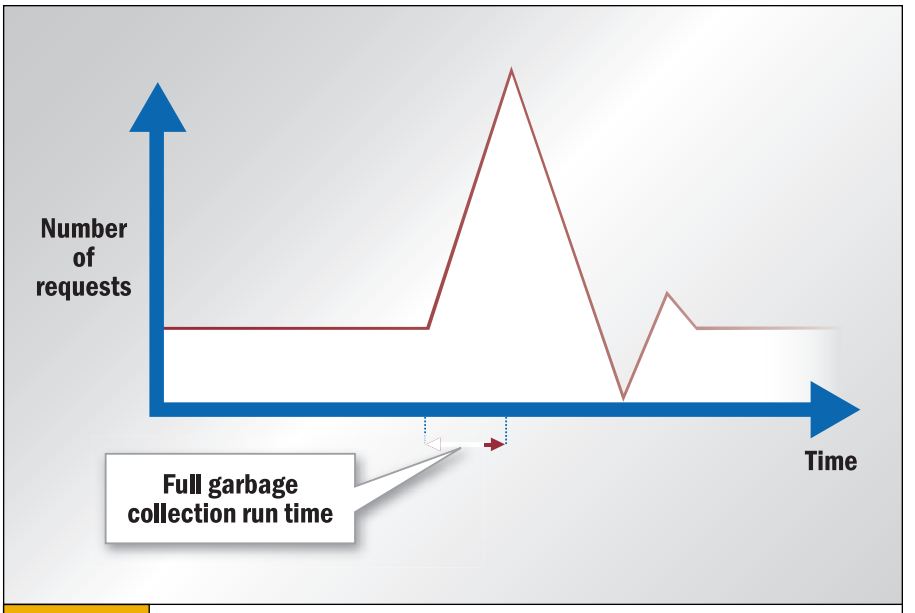

Figure 1 Garbage Collection Time Causing Blocked CPU and a Spike in Incoming System Requests

The next candidate for the cause of these resource constraints is the fact that during GC, regular requests of the CPU accumulate. Once GC is completed, the CPU moves on to the backlog of requests. This request "bunching" keeps the CPU busy and influences the system response time, even at moderate overall system usage. We therefore included another simulation to describe the effects both of GC stop times *and* GC request bunching. As a result, predictions now came very close to measured reality.

Implications for Java Programming and Tuning

Now that you're equipped with a basic understanding of the bottleneck effects of Java garbage collection, let's introduce you to some generally applicable guidelines for improving Java performance, addressing the key issues discovered in our investigation.

First and foremost, **avoid full GCs**. Of course this is a simple, understood notion for any Java developer, but a concept so well known that it is often overlooked. As a developer you can avoid full GCs by adhering to the following basic rules for memory-conscious programming in Java:

- *Only create objects that you will need later on.* Unlike other programming languages, unused objects in Java have an impact, as they are included in the garbage collection.
- *Keep object sizes to a minimum.* The smaller the object, the fewer CPU resources required to process it during garbage collection.
- *Be careful when designing caches.* Memory and CPU, thanks to garbage collection, are dependent resources in the Java environment. Caches as inherently long-lived, and potentially large objects will reduce the available memory and lead to an increase of full garbage collection frequency.

[2] A Markovian queuing model is used to analyze waiting situations by determining how key "service centers" handle incoming requests. For more information, visit www2.uwindsor.ca/~hlynka/queue.html.

For efficient cache design, this overhead must be weighed against the direct performance gain introduced by the cache.

- And, needless to say, *avoid any memory leaks*, which prevent programs from freeing discarded memory.

As a virtual machine (VM) administrator, it also is incumbent upon you to **minimize the duration of full GCs**. This is a task that should be addressed by appropriate tuning of the Java virtual machine heap size[3] and garbage collection properties:

- *Find an optimal heap size.* If our only concern were to minimize GC frequency (first conclusion), then configuring a large heap size would seem an appropriate reaction. However, large heaps have a negative effect on the full GC duration, which in turn significantly increases the average response time. As the average response time depends nonlinearly on the full GC duration, an optimal heap size can be found (depending on the application characteristics).
- *Tune garbage collection properties.* Choose garbage collection settings that minimize the GC duration. For example, the Sun Java Virtual Machine offers a variety of configuration options, including parallel and concurrent garbage collection (see **http://java.sun.com/docs/hotspot/gc1.4.2** for more information). Other virtual machines may have other guidelines for minimizing the duration of garbage collection runs.

Summary

To ensure scalable, high-performing Java applications in your enterprise, and to avoid slowing down your users with lagging systems and long interaction response times, you must avoid poorly tuned garbage collection. By understanding how garbage collection works, you can better understand how to prevent the performance delays it can cause.

Since Java was not originally designed for multi-user applications, the careful design of Java objects and appropriate VM tuning is of paramount importance to SAP developers and system administrators. We encourage everyone to visit our Web log on the SAP Developer Network at **www.sdn.sap.com/sdn/weblogs.sdn?blog=/pub/wlg/1631** to take a more detailed look at our research, from which the above performance guidelines are derived. ■

[3] A Java heap is a repository for live objects, dead objects, and free memory.

Susanne Janssen joined SAP in 1997, after finishing her studies in applied linguistics and cognitive science at the Universities of Mainz and Edinburgh. She has been a member of the SAP Performance, Data Management, and Scalability team since 1998, where she manages sizing processes, projects, and customer contacts, as well as supports the rollout of the performance standard. She is also responsible for the cooperation of SAP and technology partners and hardware vendors in the area of sizing. One of her major projects for 2005 is the rollout of the new Quick Sizer, SAP's online sizing tool.

Rudolf Meier began his IT experience in data acquisition, analysis, and simulation of physics experiments at accelerator labs around the globe. He now participates in the exciting task of accelerating SAP applications. He joined the SAP Performance, Data Management, and Scalability team in 2004. In addition to working on ABAP performance topics, he also focuses on the performance of new UI technologies in the Java world. Rudolf received his Ph.D. in Experimental Physics from the University of Karlsruhe in 1993, and is a private lecturer at the University of Tübingen, Germany.