Modeling, Analysis and Throughput Optimization of a Generational Garbage Collector

David Vengerov

Sun Microsystems Laboratories
Mailstop UMPK16-160
16 Network Circle
Menlo Park, CA 94025
david.vengerov@sun.com

Abstract

One of the garbage collectors in Sun's HotSpot Java^{TM} Virtual Machine is known as the generational throughput collector, which was designed to have a large throughput (fraction of time spent on application's work rather than on garbage collection). This paper derives an analytical expression for the throughput of this collector in terms of the following key parameters: the sizes of the Young and Old memory spaces and the value of the tenuring threshold. Based on the derived throughput model, a practical algorithm ThruMax is proposed for tuning the collector's parameters so as to formally maximize its throughput. This algorithm was implemented in a custom Java Virtual Machine (JVM) based on the HotSpot sources currently shipped in JDK^{TM} 6, and its performance was evaluated on multiple benchmark workloads. A consistent improvement in throughput was demonstrated when the ThruMax algorithm was enabled in this JVM. More importantly, the ThruMax algorithm was shown to actually converge the collector's parameters to the optimal values that resulted in the highest benchmark throughput.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory Management (garbage collection)

General Terms Algorithms, Performance, Theory

Keywords Garbage collection, throughput model, parameter optimization, tenuring threshold

1. Introduction

The Java^{TM} platform is used for a wide variety of applications, from small applets on desktops to web services on large servers. The popularity of Java is due, in part, to its capability of doing automatic memory management. That is, the programmer does not need to explicitly de-allocate objects, and the system automatically detects and recycles the space occupied by the unreferenced objects (garbage). Several garbage collection (GC) policies can be used in JDK 6 (the latest release of the Java SE Platform as of the time of the writing of this paper), one of which is the generational throughput collector [1]. This collector divides the available heap into two spaces: Young and Old. The Young space is further divided into Eden, where the new objects are allocated, and two survivor spaces, which contain objects that stay alive (are still used by the application) after being examined once during each "minor" GC for the number of times that is less than a specified value of the TenuringThreshold. If an object is found to be alive after it is examined TenuringThreshold number of times, it gets promoted to the Old space. Once the Old space fills up, a "major" GC occurs. The application is stopped when a minor or a major garbage collection occurs, and so the generational throughput collector in JDK 6 can be classified as a "stop-the-world" collector.

The heap size has a significant effect on performance of a Java application and can be resized dynamically to meet the changing needs of the application. A larger heap results in a larger GC throughput (the garbage collections become less frequent), but the garbage collections can freeze the running application for too long (this can be unacceptable in some cases) and performance of other concurrently running applications can suffer if Java consumes too much of the system's memory. If the Java application is running alone on a server, then the heap size larger than the available RAM will lead to paging, which greatly slows down the application.

Starting with JDK 5, a special policy (called "ergonomics") for dynamically adjusting the heap size, the sizes of the Young and Old memory spaces and the TenuringThreshold can be optionally enabled in the HotSpot generational throughput collector [1]. Some other examples of rules for adaptive heap sizing are given in [4, 11]. The ThruMax algorithm does not address the problem of sizing the total heap and can be used in conjunction with any of the above schemes. Instead, the ThruMax algorithm addresses the problem of dynamically adapting the *relative* sizes of Young and Old spaces (also called generations) for a given heap size so as to maximize the GC throughput, which has not been addressed before to the best of our knowledge.

The ThruMax algorithm also adapts the tenuring threshold (the number of times an object is examined and confirmed to be alive before promoting it to the Old space) so as to maximize the GC throughput. Another adaptive policy for setting the tenuring threshold was proposed in [10], which works by keeping track of the number of minor GCs each object has survived and then computing the tenuring threshold that would result in enough data promoted out of the Young space to ensure that the next minor GC will meet the pause time goal. Some researchers (e.g., [3, 6]) have investigated the use of a dynamic "threatening boundary," which separates the memory objects into those that are "old" and those that are still collected by minor GCs. The HotSpot ergonomics policy is also capable of dynamically adapting the tenuring threshold and decreases it by 1 if minor-gc-cost $>$ 1.1(major-gc-cost) and increases it by 1 if

ISMM'09, June 19–20, 2009, Dublin, Ireland.
ACM 978-1-60558-347-1/09/06.

Eden	Survivor Space	Survivor Space	Old
S	$X_N(S)$	$X_N(S)$	H-S-$2X_N(S)$

Figure 1. Partition of the Java heap, total size = H

major-gc-cost $>$ 1.1(minor-gc-cost), where major-gc-cost = (average time spent on a major GC)/(average time interval between successive major GCs) and minor-gc-cost = (average time spent on a minor GC)/(average time interval between successive minor GCs).

In the spirit similar to that of adapting the tenuring threshold, some researchers have investigated ways to improve the GC efficiency through the use of adaptive pre-tenuring, where objects that are expected to be long-lived are allocated directly in the Old space (e.g., [9, 8, 5]). Other researchers have investigated a complimentary approach of allocating objects that are expected to be short-lived in special small memory spaces and then de-allocating all such objects together when they are all dead (e.g., [2]). All of the above works, including the HotSpot ergonomics policy, suggest some heuristic rules that do not guarantee that the GC throughput will actually be maximized as a result of these policies. The policy presented in this paper for dynamically adapting the tenuring threshold is based on the first (to the best of our knowledge) mathematical model of the GC throughput and is actually designed to formally maximize it.

The outline of this paper is as follows. Section 2 describes the heap layout of a generational garbage collector and its basic dynamics. Section 3 derives an analytical expression for the throughput for such a garbage collector. Section 4 derives the expression for the steady state amount of data in the survivor spaces (in terms of the function $d_n(S)$ – the total size of objects surviving at least n minor GCs out of those that filled up Eden of size S between two successive minor GCs) and then analyzes the properties of $d_n(S)$, which are critical for determining how the GC throughput is going to change as the key GC parameters (the size of Eden and the tenuring threshold) are varied. Section 5 analyzes the dependence of the GC throughput on the tenuring threshold. Section 6 analyzes the dependence of the GC throughput on the size of Eden. Section 7 presents the ThruMax algorithm for dynamically modifying Young and Old generation sizes and tenuring threshold so as to maximize the GC throughput for a workload whose properties may change over time. This algorithm is evaluated in Section 8 on multiple benchmark workloads (with the focus being on SPECjbb2005) and its performance is compared with the HotSpot ergonomics policy.

2. Heap layout of a generational garbage collector

The generational garbage collector uses the following partitions of the heap H: Eden, Old (Tenured) and two equally sized survivor spaces (see Figure 1). The Java application allocates new objects in Eden. When Eden fills up, a *minor* garbage collection (GC) occurs and the live data is copied from Eden to the *survivor* space. When Eden fills up again, all the live data from Eden and from the previously used survivor space is copied into the other survivor space. In this fashion, one survivor space always maintains the *tenuring* data, while the other survivor space is empty. Let N be the tenuring threshold parameter in the garbage collector. This means that when a newly generated object has survived $N + 1$ minor GCs (was copied from Eden into a survivor space and then was copied N times back and forth between the two survivor spaces), it gets copied into Old at the next minor GC. Therefore, Old contains the $tenured$ objects that are expected to be long-lived. When Old fills up, a $major$ garbage collection of the whole heap occurs, during which all the surviving data is placed into Old. Let S and $X_N(S)$ be the sizes of Eden and a survivor space. If H is the size of the Java heap, then the size of Old is constrained to be $H - S - 2X_N(S)$. The layout of the Java heap is presented visually in Figure 1.

3. Throughput model for a generational garbage collector

3.1 The need for a throughput model

When a minor or a major GC occurs, the application stops its work. The *throughput* of such a "stop-the-world" garbage collector can be defined as the fraction of time the application spends working as opposed to being stopped while waiting for a GC to complete.

In theory, one does not need a throughput model to optimize the GC parameters – if throughput can be measured directly, then the gradient of throughput with respect to parameters can be computed and parameters can then be adjusted so as to maximize throughput. In order to measure the new GC throughput after a certain change in parameters is made, the Java application needs to be observed for at least two major garbage collections, so as to measure the number of minor collections between two successive major collections. Unfortunately, major collections occur very infrequently for some applications. As will be shown in Section 3.2, the throughput model derived in this paper allows the ThruMax algorithm to adjust parameters after every few *minor* garbage collections, thereby significantly increasing the adaptation speed. Therefore, the throughput model presented in equation (1) below and refined in equations (2) – (4) is not just an academic interest but is central to the algorithm we implemented.

The throughput model of a generational garbage collector will be derived under the assumption that the Java application is in a steady state – the probabilities of various events occurring do not change over time and the live data does not overflow the total heap space. This assumption will allow us to generate a valuable intuition about the dependence of the GC throughput on the key tunable parameters (Eden size S and the tenuring threshold N), as described in Sections 5 and 6. However, the ThruMax algorithm presented in Section 7 does not require the application to be in a steady state, as it continually estimates the values of the key variables in equation (4) and gradually adjusts S and N in the direction of increasing throughput (which is predicted under the assumption that the latest observations of the key variables represent their expected future values). The ThruMax algorithm can be activated once the utilization of the survivor spaces stabilizes (when the amount of data surviving a minor GC stays between 75% and 90% of $X_N(S)$ for several minor GCs in a row). If the survivor space utilization drops below 75% or rises above 90%, then $X_N(S)$ can be resized to bring its utilization inside the 75-90% range. Once S is adjusted using the ThruMax algorithm, $X_N(S)$ can be scaled proportionately, which is done during the experiments described in Section 8.

3.2 Expression for the GC throughput

Let $A(S)$ be the average time required to fill up Eden with new objects (when the Java application is doing useful work). Let $t_N(S)$ be the average time spent on a minor GC. Let $K_N(S)$ be the average number of minor GCs between two successive major GCs. Let T be the average time spent on a major GC. Then, assuming that a minor GC happens right before a major GC (which is the case in the HotSpot generational throughput collector), the steady state GC throughput $f_N(S)$ can be expressed as:

$$f_N(S) = \frac{A(S)K_N(S)}{t_N(S)K_N(S) + T + A(S)K_N(S)}, \tag{1}$$

where $A(S)K_N(S)$ is the average time spent by the Java application on useful work between two major GCs (which is composed of $K_N(S)$ periods of length $A(S)$) and $t_N(S)K_N(S)$ is the total time spent on minor garbage collection between two major GCs.

Consider the objects that are in Eden right before a minor GC, and let's denote by $d_n(S)$ the total size of those that are expected to survive at least n minor GCs. Then, the expected amount of data passing into Old at each minor GC is $d_{N+1}(S)$. Also, let's denote by J the expected total size of objects surviving a major GC. We can express $K_N(S)$ as free space in Old after a major GC divided by the amount of data passing into Old after each minor GC:

$$K_N(S) = \frac{H - J - 2X_N(S) - S}{d_{N+1}(S)}. \quad (2)$$

Let's rewrite equation (1) as $f_N(S) = 1/(1 + 1/u_N(S))$, where

$$u_N(S) = \frac{A(S)}{t_N(S) + T/K_N(S)} \quad (3)$$

and say that our goal is to maximize $u_N(S)$. Using equation (2) we can re-write this expression as

$$u_N(S) = \frac{A(S)}{t_N(S) + \frac{T \cdot d_{N+1}(S)}{H-J-2X_N(S)-S}}. \quad (4)$$

The above equation shows that once T and J have been measured after the first major GC, the impact of changing S or N on the GC throughput can be evaluated after several minor GCs (which are needed to obtain estimates of $t_N(S)$ and $d_{N+1}(S)$ as average values of the actually observed minor GC times and promoted data amounts). Therefore, the adaptation process can be performed much quicker using the throughput model presented above, for in the absence of such a model, several *major* GCs would need to be performed in order to measure accurately the GC throughput (which depends on the number of minor GCs between two major GCs).

At this point, a reader who is not interested in mathematical analysis can read the "introductions" to Sections 4 and 5 and then jump to Section 6 that discusses the optimal setting of the Eden size S. A reader can also jump directly to Section 7 that describes how the ThruMax algorithm adjusts S or N after every few minor GCs by continually estimating $t_N(S)$ and $d_{N+1}(S)$.

4. Steady state amount of live data in the survivor spaces

The function $d_n(S)$ was introduced in Section 3.2 as the total size of the objects that are expected to survive at least n minor GCs out of those that filled Eden right before a minor GC. This function plays the central role in the analysis performed in Sections 5 and 6 of how GC throughput depends on S and N in steady state. This section derives an analytical expression for $d_n(S)$ and then studies its mathematical properties. It also shows that the steady state amount of live data in the survivor spaces is given by $g_N(S) = \sum_{n=1}^{N} d_n(S)$.

4.1 Deriving the expression for $d_n(S)$ and $g_N(S)$

Let $\{\tau_1, ..., \tau_n\}$, be a sequence of random variables indicating n successive values of the time required to fill Eden with new objects. Note that the definition of $A(S)$ in Section 3.2 implies that the expectation of τ_i, denoted by $E[\tau_i]$, is equal to $A(S)$. Let G be the probability distribution function of object lifetimes, so that $G(x)$ is the probability that the lifetime of a newly generated object will be less than x. By its definition, $G(x)$ rises monotonically from 0 to 1 as x increases from 0 to infinity.

The probability that an object j created at a certain time t_j survives a total of at least n minor GCs is the probability that the object's lifetime is greater than $\sum_{i=1}^{n} \tau_i - t_j$, the time until the nth minor GC conditional on $t_j < \tau_1$. This probability can be expressed as $E[1 - G(\sum_{i=1}^{n} \tau_i - t_j) | \tau_1 > t_j]$, where $E[x|\tau_1 > t_j]$ denotes expectation of x over the random variables τ_i conditional on $\tau_1 > t_j$. If we don't know the exact time when an object was created but know only that it was created before the first GC event (that is, $t_j < \tau_1$), then an *a priori* probability P of it surviving a total of at least n minor GCs can be computed by taking the expectation over object generation times, which can be assumed to be uniformly distributed on the interval $[0, \tau_1]$:

$$P = E[\frac{1}{\tau_1}\int_0^{\tau_1} (1 - G(\sum_{i=1}^{n} \tau_i - x))dx]. \quad (5)$$

Assuming that the size of each object is small relative to S, the probability P in equation (5) can be interpreted as a $fraction$ of objects that will survive at least n minor GCs out of those that filled up Eden (of size S). Therefore, the expected total size of objects surviving at least n minor GCs (denoted by $d_n(S)$ in Section 3) is S multiplied by P:

$$d_n(S) = S \cdot E[\frac{1}{\tau_1}\int_0^{\tau_1} (1 - G(\sum_{i=1}^{n} \tau_i - x))dx]. \quad (6)$$

The survivor spaces are emptied during a major GC and all live data is copied into Old. At the next minor GC, the amount of data copied into the survivor space is $d_1(S)$. After one more minor GC, some of that data dies, leaving $d_2(S)$ of live data (based on the definition of $d_n(S)$). However, $d_1(S)$ of new live data is copied into the survivor space from Eden. Therefore, the amount of data in survivor spaces after the major GC would increase as follows with each minor GC: $d_1(S), d_1(S) + d_2(S), d_1(S) + d_2(S) + d_3(S)$, and so on, until it reaches

$$g_N(S) = \sum_{n=1}^{N} d_n(S), \quad (7)$$

where N is the tenuring threshold.

4.2 Analyzing the properties of $d_n(S)$

We will now analyze the mathematical properties of $d_n(S)$. The key results of this subsection are that $d_1(S)$ and $g_N(S)$ are concave increasing functions of S while $d_n(S)$ for $n > 1$ is hump-shaped AND is most likely concave in the region where it is increasing. This will be used by the ThruMax algorithm in Section 7 for setting the initial Eden size.

As a starting point, observe that as n increases, $G(\sum_{i=1}^{n} \tau_i - x)$ monotonically approaches 1, implying that $d_n(S)$ is a monotonically decreasing function of n and approaches 0 as $n \to \infty$.

The dependence of $d_n(S)$ on S is more complicated. First, observe that it is reasonable to assume that in steady state, the *expected* time to fill Eden with objects is a linear function of the Eden size. Then, we can express $E[\tau_i] = S/r$ for some r. The simplest expression for $d_n(S)$ would be the one where each τ_i is replaced with its expected value of S/r:

$$d_n(S) \approx r \int_{(n-1)S/r}^{nS/r} (1 - G(x))dx. \quad (8)$$

This expression shows that $d_n(0) = 0$. Also, since $1 - G(x)$ decreases as x increases, $d_1(S)$ is basically a summation of a decreasing sequence of non-negative terms, implying that $d_1(S)$ is a concave increasing function of S. Equation (7) also shows that $g_N(S) = \sum_{n=1}^{N} d_n(S) \approx r \int_0^{NS/r} (1 - G(x))dx = d_1(NS)$, implying that $g_N(S)$ is a concave increasing function of S just like $d_1(S)$.

An important question that needs to be answered is whether $d_n(S)$ (for $n > 1$) can be a convex function of S in the region where it is increasing. Let's define $g(x) = 1 - G(x)$. Then, equation (8) becomes $d_n(S) \approx r\int_{(n-1)S/r}^{nS/r} g(x)dx$. Recall that the Leibniz's rule states that $\frac{d}{dS}\int_{a(S)}^{b(S)} G(x)dx = G(b(S))\frac{db(S)}{dS} - G(a(S))\frac{da(S)}{dS}$, which can be used to differentiate both sides of this expression for $d_n(S)$ to obtain

$$d_n'(S) \approx ng(nS/r) - (n-1)g((n-1)S/r). \qquad (9)$$

Differentiating the above equation we get an expression for the second derivative of $d_n(S)$:

$$d_n''(S) \approx \frac{1}{r}[n^2 g'(nS/r) - (n-1)^2 g'((n-1)S/r)]. \qquad (10)$$

Equation (9) shows that in order for $d_n(S)$ to be increasing (i.e., $d_n'(S) > 0$), we need to have $\frac{g(nS/r)}{g((n-1)S/r)} > \frac{n-1}{n}$. Equation (10) shows that in order for $d_n(S)$ to be convex (i.e. $d_n''(S) > 0$), we need $g'(nS/r) > \frac{(n-1)^2}{n^2}g'((n-1)S/r)$. One can see that if $g(x)$ starts decreasing steeply but then makes a sharp turn around the point x_0 and becomes very flat for $x > x_0$, then the above two conditions might hold for $(n-1)S/r < x_0 < nS/r$. Such sharp turns, however, are unlikely to happen in "smooth" functions that are infinitely differentiable. For example, one can check analytically that the above two conditions do not hold simultaneously for the family of functions $G(x) = 1 - e^{-\lambda x^k}$ where $k > 0$.

This family of functions covers many scenarios we expect to see in reality. For example, if we let $k = 1$, we get a subfamily $G(x) = 1 - e^{-\lambda x}$ that models the scenario where the same fraction λ of live objects is expected to die at any moment in time. For this family of functions one can analytically compute $d_n(S) = \frac{r}{\lambda}e^{-\lambda nS/r}(e^{\lambda S/r} - 1)$ and $g_N(S) = \frac{r}{\lambda}(1 - e^{-\lambda NS/r})$. These expressions show that as $S \to \infty$, $d_1(S) \to r/\lambda$ and $g_N(S) \to r/\lambda$ (because $g_N(S) = d_1(NS)$ as was shown previously in the general case and as can be seen by comparing the above expressions for $d_1(S)$ and $g_N(S)$). This might be surprising because one might think that as the Eden size becomes very large, the amount of live data in the survivor space, $g_N(S)$, should also become very large. Section 5.1 sheds some light on why $g_N(S)$ might approach an asymptote as S increases for a Java application that is in a steady state. The above expressions also show that as $S \to \infty$, $d_n(S) \to 0$ for $n > 1$, implying that $d_n(S)$ first rises and then falls back to 0 as S increases. Intuitively, this means that as S becomes very large, all objects that survived one minor GC die out before the next minor GC because the wait time until the next minor GC keeps increasing.

The analysis in this section showed that the steady state amount of live data in the survivor spaces, $g_N(S)$, is a concave increasing function of S. Also, it showed that $d_n(S)$ (and hence the amount of data promoted into Old, which is given by $d_{N+1}(S)$) is most likely a concave function of S in the region where it is increasing, but it CAN be convex increasing over one or several small intervals of the Eden size S (each interval corresponding to a sharp change in $G(x)$, where it changes quickly from being very steep to being very flat).

5. Optimizing the tenuring threshold N

The conventional wisdom for generational garbage collectors suggests that the tenuring threshold should be increased if it would result in a noticeable decrease in the amount of data passing into Old in steady state. However, the formal throughput analysis performed in this section shows that the decision of whether or not to increase the tenuring threshold cannot be made correctly without also considering the ratio of the free space in Old after a major GC to the amount of data promoted into Old at every minor GC. The analysis in Section 5.3 also shows that the optimal value of the tenuring threshold will be finite if $H - S < 3J$ (where J is the expected total size of objects surviving a major GC) and infinite otherwise. In the unlikely case when the average object lifetime is VERY large, the optimal tenuring threshold is $N = 0$.

5.1 Deriving the expressions for J and T

First, let's derive the expression for the amount of data that survives a major GC (that collects all spaces) in a steady state, which was denoted by J in Section 3.2. It turns out that J is independent of S, $X_N(S)$ and N, and depends only on the shape of G – the probability distribution function of object lifetimes (provided $H > J$). To see that this indeed is the case, note that after the application was working for T_0 units of time in a steady state (T_0 does not include the times when the application was stopped due to garbage collections), the probability that an object j created at time $t_j < T_0$ will still be alive at time T_0 is $1 - G(T_0 - t_j)$. If we don't know the exact time t_j, then the *a priori* probability of an object created before T_0 still being alive at time T_0 is the average value of the function $h(x) = 1 - G(T_0 - x)$ over the interval $[0, T_0]$, which is

$$P = \frac{1}{T_0}\int_0^{T_0}(1 - G(T_0 - x))dx = \frac{1}{T_0}\int_0^{T_0}(1 - G(x))dx. \quad (11)$$

Note that in a steady state, the *expected* amount of data generated over the time interval of length T_0 is going to be a linear function of T_0, say rT_0. Then, the expected amount of data surviving a major GC after the application has been working for T_0 units of time is $rT_0P = r\int_0^{T_0}(1 - G(x))dx$. Finally, the quantity J can be computed by letting $T_0 \to \infty$:

$$J = r\int_0^{\infty}(1 - G(x))dx. \qquad (12)$$

A comparison of equations (8) and (12) shows that as $S \to \infty$, the expected amount of data surviving a minor GC, $d_1(S)$, approaches J and the expected amount of live data in the survivor spaces, $g_N(S)$, also approaches J (because $g_N(S) = d_1(NS)$). Hence, if a Java application is in a steady state (and the distribution function of object lifetimes is well modeled by $G(x) = 1 - e^{-\lambda x}$ as explained in Section 4.2), then the amount of data surviving the major GC is approximately equal to r/λ.

The expected major GC time T is linear in the amount of data surviving the major GC. It is also linear in the total size of memory spaces that have some live data. Since a minor GC is assumed to occur right before a major GC (as is the case in the HotSpot generational throughput collector), the live data is found only in Old and in one survivor space when a major GC starts. Therefore, the major GC time can be approximated as $T \approx c_2 + c_3(H - S - X_N(S)) + a_2J$. This expression shows that the maximum percentage change in T is bounded as S or N are varied (and our experiments show that T stays almost constant). Therefore, we will denote the major GC time as T for the rest of the analysis, and a careful reader can check that its dependence on S and N does not change any of our conclusions.

5.2 Deriving an expression for $t_N(S)$

The time required to perform a minor GC (when N or more minor GCs have already happened since the last major GC and the amount of live data in the survivor spaces has reached its steady state amount of $g_N(S)$) can be approximated as $t_N(S) \approx c + a(g_N(S) + d_{N+1}(S)) + B(S)$, where $B(S)$ is the time required to determine which data is alive by tracing all the data pointers in Old (which is well-modeled by a linear function of the amount of data in Old before the nth minor GC), and $a(g_N(S) + d_{N+1}(S))$ is the time required to copy the live data (a linear function of the amount of live data copied). Our experimental studies showed that $B(S)$ is almost constant, which is not surprising because most pointers go from the younger objects to the older objects, and the few pointers that go from Old into Eden are mostly from the objects that were placed in Old most recently [7]. Hence, the number of these pointers stays relatively constant, and since they are tracked explicitly in Java's generational garbage collector, the whole Old space does not need to be examined to find them. Therefore, assuming that $K_N(S)$ is much greater than N, the average time required to perform a minor GC can be (approximately) expressed as:

$$\begin{aligned} t_N(S) &= c + a(g_N(S) + d_{N+1}(S)) \\ &= c + a\sum_{n=1}^{N+1} d_n(S) \\ &= c + a g_{N+1}(S) \end{aligned} \qquad (13)$$

5.3 Analyzing the impact of changing N on the GC throughput

The GC throughput model presented in equation (3) suggests that any particular value of N should be increased to $N + 1$ if the denominator of equation (3) evaluated at $N + 1$ minus the denominator evaluated at N is less than 0:

$$t_{N+1}(S) - t_N(S) + T(\frac{1}{K_{N+1}(S)} - \frac{1}{K_N(S)}) < 0. \qquad (14)$$

Equation (7) shows that the amount of live data in the survivor space would increase by $d_{N+1}(S)$ if the tenuring threshold is increased from N to $N+1$. Therefore, in order to avoid an overflow of the survivor space, its size should be increased as the tenuring threshold is increased from N to $N+1$. Assuming that the minimal increase by $d_{N+1}(S)$ is made, inequality (14) can be rewritten (using equations (2) and (13)) as

$$a d_{N+2}(S) + T(\frac{d_{N+2}(S)}{W_N(S) - 2d_{N+1}(S)} - \frac{d_{N+1}(S)}{W_N(S)}) < 0, \qquad (15)$$

where $W_N(S) = H - J - S - 2X_N(S)$ gives the free space in Old right after a major GC when the tenuring threshold is N. We can further rewrite (15) as

$$\frac{aW_N(S)}{T} < \frac{d_{N+1}(S)}{d_{N+2}(S)} - \frac{W_N(S)}{W_N(S) - 2d_{N+1}(S)}. \qquad (16)$$

Equations (8) and (12) imply that $J = \sum_{n=1}^{\infty} d_n(S)$. Hence, if we assume that the amount of live data does not overflow the heap as time goes on (i.e., the Java application reaches a steady state), we must have $\sum_{n=1}^{\infty} d_n(S) < \infty$. As the average object lifetime increases, the series $\sum_{n=1}^{\infty} d_n(S)$ begins to converge slower and slower. A standard convergence test states that $\sum_{n=1}^{\infty} d_n(S)$ converges if $\lim_{n\to\infty} \frac{d_n(S)}{d_{n+1}(S)} > 1$ and diverges if $\lim_{n\to\infty} \frac{d_n(S)}{d_{n+1}(S)} < 1$. Since $d_n(S)$ is a monotonically decreasing function of n (as shown in Section 4.2), then for a fixed N, the ratio $\frac{d_{N+1}(S)}{d_{N+2}(S)}$ approaches 1 as the average object lifetime increases (almost all of the objects that survive $N+1$ minor GCs will also survive $N+2$ minor GCs). Since $\frac{W_N(S)}{W_N(S)-2d_{N+1}(S)} > 1$, the right hand side of (16) eventually becomes negative as the average object lifetime increases, and the optimal tenuring threshold becomes 0 (since inequality (16) stops being true).

As the average object lifetime decreases, $\frac{d_{N+1}(S)}{d_{N+2}(S)}$ can become arbitrarily large (almost all of the objects that survive $N+1$ minor GCs will die after $N + 2$ minor GCs), and hence at some point it becomes greater than the left hand side of (16). At that point, the decision of whether or not to increase the tenuring threshold begins to depend on the value of $W(S)$ relative to $d_N(S)$. That is, if the free heap space after a major GC is large relative to the amount of data promoted at each minor GC, then $\frac{W_N(S)}{W_N(S)-2d_{N+1}(S)}$ is close to 1 and it is beneficial to increase the tenuring threshold (since inequality (16) would hold in this case). However, as N increases, the amount of data in the survivor space approaches J (as can be seen by comparing equations (8) and (12)), and hence if $H - J - S < 2J$, then $W_N(S)$ (which is equal to $H - J - S - 2X_N(S)$) will eventually be reduced to 0, pushing $\frac{W_N(S)}{W_N(S)-2d_{N+1}(S)}$ toward infinity and the right side of (16) toward negative infinity. Therefore, for a sufficiently small object lifetime, the optimal tenuring threshold will be some finite value of N – the largest N for which (16) still holds. If, on the other hand, $H-J-S$ is much larger than $2J$, then the optimal tenuring threshold will be infinitely large, since the inequality (16) would hold for any N, assuming that the average object lifetime is small enough for $\frac{d_{N+1}(S)}{d_{N+2}(S)}$ to be greater than the left hand side of (16).

Note that if $G(x) = 1$ for x greater than some constant M, then for some value of n, $d_{n+2}(S)$ will be 0 while $d_{n+1}(S)$ will be greater than 0 (we have observed this in practice for some workloads). In this case, the above analysis shows that it would be beneficial to set the tenuring threshold N equal to $n + 1$ if it would not trigger an immediate major GC.

In the general case of the average object lifetime being not too large and not too small, the optimal value of N depends on the shapes of the left side of (16) and the right side of (16) as a function of N. Hence, as stated in the beginning of this section, we have shown that the decision of whether or not to increase the tenuring threshold cannot be made without considering the relative sizes of $W_N(S)$ and $d_{N+1}(S)$. That is, all components of the inequality (16) should be considered, and an iterative algorithm for adjusting the tenuring threshold should be used (since one does not know ahead of time the ratio $d_{N+1}(S)/d_{N+2}(S)$). Such an algorithm is presented in Section 7.

6. Optimizing the Eden size S

The current belief in the engineering community is that if the Eden size S can be increased without affecting the size of Old, then the GC throughput will increase. The analysis in Section 4 shows that this indeed is the case for applications that have a smooth steady state behavior and have a distribution function of object lifetimes that is well modeled by $G(x) = 1 - e^{-\lambda x}$. To see this, let's rewrite equation (3) as

$$\tilde{u}_N(S) = \frac{A(S)}{t_N(S) + Td_{N+1}(S)/L} \qquad (17)$$

where L is the fixed size of Old. Then, as S increases, the expected time $A(S)$ required to fill Eden in steady state increases linearly, but the amount of data in the survivor space (given by $g_N(S)$) increases slower than S because $g_N(S)$ is a concave function of S (as was determined in Section 4 for a general shape of $G(x)$). Section 5.2 has established that the minor GC time $t_N(S) \approx c + a \cdot g_N(S)$, and hence $t_N(S)$ is also a concave increasing function of S. Also, as S increases, the amount of data passing into Old in steady state (given by $d_{N+1}(S)$) is either a concave increasing or a decreasing function of S (as was determined in Section 4 for $G(x) = 1 - e^{-\lambda x^k}$ with $k > 0$). Therefore, as S increases linearly, the denominator of equation (17) increases slower than linearly, implying that the GC throughput increases IF the application has a smooth steady state behavior. However, the analysis in section 4 also showed that $d_{N+1}(S)$ can be convex increasing over a small region of S (for some rare shapes of $G(x)$), and hence the term $Td_{N+1}(S)/L$ in the denominator of equation (17) can increase faster than linearly as S increases, possibly increasing the whole denominator of (17) faster than linearly as S increases, thereby decreasing the GC throughput.

Note that as S increases and the size of Old does not change, then the application's footprint and the maximum GC pause times (which occur during major GCs) also increase, which is undesirable in some circumstances. In order to keep the application's footprint and the maximum pause time constant, one can increase S while decreasing the size of Old by the same amount. In this case, even if $t_N(S)$ and $d_{N+1}(S)$ are concave increasing functions of S, the GC throughput can still decrease as S increases because the term $1/(H - J - S - 2X_N(S))$ in the denominator of equation (4) is a convex increasing function of S. For example, when $S + 2X_N(S) \geq (H - J)/2$, then $1/(H - J - S - 2X_N(S))$ grows faster than linearly as a function of S. One can check that when $S+2X_N(S) = (H-J)/2$, then a 10% increase in $S+2X_N(S)$ results in an 11% increase in $T/(H - J - S - 2X_N(S))$. Therefore, one can safely increase the size of the young generation and decrease the size of Old up to the point when the young generation is equal to one half of the free heap space after a major GC. At that point, one should look more carefully at the percentage change in $t_N(S)$ and $T/K_N(S)$ in response to each incremental increase in S so as to determine when to stop increasing the young generation size because the optimal value has been reached. An iterative algorithm that dynamically estimates the values of $t_N(S)$ and $T/K_N(S)$ so as to determine the optimal change in S is presented in the next section.

7. A practical algorithm for adjusting generation sizes and tenuring threshold

7.1 Overview

Below is a simplified outline of the ThruMax (Throughput Maximization) algorithm we propose for adjusting the generation sizes and the tenuring threshold.

Adjusting S:

1. After every change in S, adjust $X_N(S)$ so that the survivor space utilization will stay within some reasonable boundaries (say 75% to 90%).
2. Observe the new values of $t_N(S)$ and $d_{N+1}(S)$ and estimate numerically the derivatives $t'_N(S)$ and $d'_{N+1}(S)$.
3. Use the above derivatives to compute the predicted value of $u_N(S + \Delta S)$ and $u_N(S - \Delta S)$.
4. If $u_N(S + \Delta S) > \max[u_N(S - \Delta S), u_N(S)]$, then increase S.
5. If $u_N(S - \Delta S) > \max[u_N(S + \Delta S), u_N(S)]$, then decrease S.
6. If $u_N(S) > \max[u_N(S + \Delta S), u_N(S - \Delta S)]$, then stop changing S and switch to changing the tenuring threshold N.
7. Repeat steps 1-6 a maximum of 3 times and then switch to adjusting N.

Adjusting N:

1. Increase N by 1 unless a decrease was suggested in step 4 below during the previous episode of adjusting N.
2. Let several minor GCs pass, observe the new values of $t_N(S)$ and $d_{N+1}(S)$ and use them to compute $u_{N+1}(S)$.
3. If $u_{N+1}(S) > u_N(S)$, then the increase was successful and another increase is made. Repeat steps 2 and 3 a maximum of 3 times, then switch to adjusting S.
4. If $u_{N+1}(S) < u_N(S)$, then undo the increase and switch to changing S. The next episode of adjusting N will start by decreasing N by 1.

7.2 Adjusting the Eden size S

We will now describe in a greater detail how adjustment of the Eden size S should be performed. The algorithm starts with $S = S_0$ for which Eden and Old are of an equal size (based on the insights developed in Section 6) and wait for a major GC to occur to estimate T, J and the other variables in the denominator of (4), which we will call $Z_N(S)$. Then set $S_1 = pS_0$ (where p is slightly larger than 1, say 1.1), perform several minor GCs and estimate (e.g., as an arithmetic average) the new values of the minor pause time, $t_N(S_1)$, and of amount of data promoted into Old at each minor GC, $d_{N+1}(S_1)$. At this point, one can (locally) approximate the derivatives of these functions with $t'_N(S_1) \approx (t_N(S_1) - t_N(S_0))/(S_1 - S_0)$ and $d'_{N+1}(S_1) \approx (d_{N+1}(S_1) - d_{N+1}(S_0))/(S_1 - S_0)$.

After that, one can compute an estimate for the optimal value of S that will maximize $u_N(S)$ under the assumption that the Java application is in a steady state (which implies that the time $A(S)$ required to fill Eden with live objects can be approximated as S/r for some r). In this case, the optimal Eden size S^* will satisfy $[u_N(S^*)]' = 0$ or equivalently $Z_N(S^*) - S^* Z'_N(S^*) = 0$. Let $S^* = S + \Delta S$ and rewrite the previous condition as $Z_N(S+\Delta S) - (S+\Delta S) Z'_N(S+\Delta S) = 0$. Since $Z'_N(S+\Delta S) \approx [Z_N(S + \Delta S) - Z_N(S)]/\Delta S$, the previous condition for the optimal ΔS can be rewritten as $Z_N(S + \Delta S)\Delta S = [Z_N(S + \Delta S) - Z_N(S)](S + \Delta S)$, and after a little algebra as

$$\frac{(S + \Delta S)}{S} = \frac{Z_N(S + \Delta S)}{Z_N(S)}, \tag{18}$$

which is equivalent to requiring the percentage change in the numerator and in the denominator of (4) to be equal in response to a given change in S, which intuitively makes sense when we want to maximize a ratio of two functions of S. Using the derivative approximation mentioned above, we can express

$$\begin{aligned} Z_N(S + \Delta S) \quad &\approx \quad [t_N(S) + \Delta S t'_N(S)] \\ &+ \quad T\frac{[d_{N+1}(S) + \Delta S d'_{N+1}(S)]}{\Delta W}, \end{aligned} \tag{19}$$

where $\Delta W = H - J - 2(X_N(S)(1 + \Delta S/S)) - (S + \Delta S)$. Therefore, one can use $S = S_1$, substitute the expression for $Z_N(S_1 + \Delta S)$ given above into (18) and use the previously stated approximations for $t'_N(S_1)$ and $d'_{N+1}(S_1)$ to solve for the optimal value of ΔS.

The above procedure should then be repeated, since the new value $S_2 = S_1 + \Delta S$ will allow one to generate new derivative estimates for $t_N(S_2)$ and $d_{N+1}(S_2)$ based on the change in these functions between S_1 and S_2. Also, the application's behavior may have changed since the last evaluation of $t_N(S_1)$ and $d_{N+1}(S_1)$, which gives an additional reason for re-estimating $t'_N(S_2)$ and $d'_{N+1}(S_2)$. Therefore, one can keep repeating the above procedure for obtaining $S_2, S_3, ...$ and adjusting the sizes of Eden, Old, and survivor spaces accordingly, in order to keep GC throughput close to being optimal.

If ΔS turns out to be very large at some iteration, it should be restricted to fall between, say, -10% and +10% of S. This will avoid large jumps in S in response to abrupt changes in the workload, when derivative estimates will change significantly. On the other hand, if the suggested ΔS is very small, then there is a high chance that the derivative estimates $t'_N(S_i)$ and $d'_{N+1}(S_i)$ will differ from the true values by very large amounts simply because a slight error in $(t_N(S_i) - t_N(S_{i-1}))$ and $(d_{N+1}(S_i) - d_{N+1}(S_{i-1}))$ due to the stochastic nature of the sampling process will be divided by a very small quantity $(S_i - S_{i-1})$ and hence greatly magnified. In order to avoid this problem, a minimum percentage change δ in S should be specified, and if $|1 - (S_i + \Delta S)/S_i| < \delta$, then the ΔS change in S is not implemented and $t'_N(S_i)$ and $d'_{N+1}(S_i)$ are estimated again after some number of minor GCs.

7.3 Adjusting the tenuring threshold N

A similar procedure to the one described in Section 7.2 can be used to dynamically adjust the tenuring threshold N. Such adjustments can happen in "episodes," each containing several changes to N. Each episode starts with estimating the latest values of $t_N(S_i)$ and $d_{N+1}(S_i)$ and using them to compute $Z_N(S_i)$. After that, a "trial" change is made in N in some direction (i.e., by increasing it by 1 or decreasing it by 1, using the direction opposite to the one used at the beginning of the previous episode) and several more minor GCs are allowed to happen. Then, the new values of $t_N(S_i)$ and $d_{N+1}(S_i)$ are estimated and $Z_N(S_i)$ is re-computed. If the new value of $Z_N(S_i)$ is found to be smaller than the old value, then the performed change in N resulted in an increase to $u_N(S)$ in equation (4), and hence another change can be made in the same direction. If at some point the new value of $Z_N(S_i)$ is found to be larger than the old value, then the previous change to N is undone and the current episode of adjustments to N is terminated. Since the optimal value of S is dependent on N (and vice versa), some adjustments to S should be performed before resuming adjustments to N. We thus obtain a co-evolutionary framework for alternating adjustments to S and N, which should converge to the optimal values for maximizing throughput if the workload remains constant.

7.4 Implementation details

The implementation details for the ThruMax algorithm were as follows. The minor pause time $t_N(S_i)$ and the amount of promoted data $d_{N+1}(S_i)$ were estimated as arithmetic averages of the corresponding values observed during the last 3 minor GCs, so as to allow for S and N to quickly reach the neighborhood of the optimal values and also quickly adapt to changing workload conditions, while still "averaging away" some noise.

The adjustment of the Eden size according to the ThruMax algorithm began after one major GC has occurred (which is needed to obtain the estimates of T and J) and the size of each survivor space $X(S)$ has reached a "steady-state", as defined by the amount of data placed in the survivor space occupying between 75% and 90% of $X(S)$ for two consecutive minor GCs. Once the steady-state was reached, any adjustment to the Eden size S also resulted in the adjustment to the survivor size by the same percentage amount, so as to ensure that with high probability that its utilization will stay between 75% and 90%. The maximum size of S was limited at any point by *eden-limit*, which was dynamically computed as the current Eden size plus the current free space in Old. The change in S was constrained to be at least 1% of S, with the maximal value being 20% if the change was positive and 10% if it was negative. A greater magnitude was allowed for increases because the insights developed in Section 6 suggest that the optimal size of the young generation is expected to be greater than half of the total heap, implying the need for the algorithm to make several quick initial increases in S. If S was within 30% of *eden-limit*, the maximal increase in S was limited to 10%, if S was within 20% of *eden-limit*, the maximal increase in S was limited to 5%, if S was within 10% of *eden-limit*, the maximal increase in S was limited to 2.5%, and if S was within 3% of *eden-limit*, no increases in S were allowed. Such limits were imposed to make sure that changes in S become smaller as S approaches its optimal value.

Adjustments to the tenuring threshold N began after the Eden size S has decreased, which indicated that S has reached its optimal neighborhood and started oscillating around its optimal value. A new episode of adjustments began whenever ΔS was computed and was found to be non-negative (indicating that there was no immediate pressure on decreasing S so as to avoid triggering an unwanted major GC), *finished-adjusting-threshold* variable was FALSE and at least 10 more minor GCs were expected to happen until the next major GC (based on the current estimate $d_{N+1}(S)$ of the amount of data promoted into Old and on the free space currently available in Old). After a change in N resulted in an increase in $Z_N(S)$, the current episode of adjustments to N was terminated and *finished-adjusting-threshold* was set to TRUE (as described in Section 7.3). The variable *finished-adjusting-threshold* was then set to false during each major GC (when new estimates of T and J became available) and also when ΔS was computed to be 0 (which indicated that S is very close to its optimal value or to *eden-limit*).

8. Experimental results

The ThruMax algorithm was implemented in a custom JVM based on the HotSpot sources currently shipped in JDK 6, and its performance was evaluated using multiple benchmarks: SPECjbb2005, SPECjvm2008, SciMark, and Volano 2.5. The experiments were conducted with either the ThruMax algorithm or the HotSpot ergonomics policy enabled, which were treated as two alternative policies for adapting the generation sizes and the tenuring threshold. In the case of SciMark and Volano, the default benchmark duration turned out to be too small to allow for more than one major GC to occur, and therefore the benchmark score was approximately the same for the ThruMax and for the ergonomics policy. In the case of SPECjvm2008, some of the sub-benchmarks had a very erratic behavior and did not allow ThruMax to reach the steady state where approximately the same amount of data survives each minor GC (the survivor space utilization often varied by more than 50% between successive minor GCs, sometimes resulting in survivor overflows), while other benchmarks were simply not sensitive to the garbage collection policy. Therefore, on many SPECjvm2008 sub-benchmarks the final score was almost the same for the ThruMax and for the ergonomics policy. There were some cases in which ThruMax resulted in a much higher score (a greater than 100% improvement was observed relative to the ergonomics policy for compiler.compiler sub-benchmark). This performance improvement, however, was due to the fixed rules for adjusting memory space sizes and the tenuring threshold that were implemented together with the ThruMax algorithm. For example, one set of rules decreased the survivor space size when its utilization was very low, which allowed the JVM to utilize the heap space more efficiently, and increased the survivor space size when its utilization approached 1, which made survivor overflows unlikely (such overflows placed extra data in the Old generation and increased the frequency of major GCs). Another set of rules decreased the Young generation size if less than 10 minor GCs were expected to occur until the next major GC (based on the amount of data promoted into Old at each minor GC). These rules, however, had nothing to do with the GC throughput model that was the main contribution of this paper.

Only the SPECjbb2005 benchmark spent most of its time in a steady state and was therefore appropriate for evaluating performance improvement due to the gradient-based adjustment of key GC parameters implemented in the ThruMax algorithm. The main user-controlled parameter in this multi-threaded benchmark is the number of threads that generate I/O requests to the database created by this benchmark. As the number of threads increases, the amount of garbage generated per unit of time increases and the garbage collections become more frequent. The number of threads can be specified by any non-decreasing sequence $\{n_1, n_2, n_3, ...\}$, which tells SPECjbb2005 to run n_1 threads for 4 minutes, then n_2 threads for 4 minutes, and so on. Experiments were conducted on an Intel©server with the Solaris^{TM} 10 Operating System installed. The server was equipped with 16GB RAM and two quad-core E5462 Xeon©processors running at 2.8GHz on a Bensley platform with 12MB L2 Cache and 1600 MHz FSB.

All experiments below were conducted with the Java heap of 2GB. For the ThruMax algorithm, the Young generation was ini-

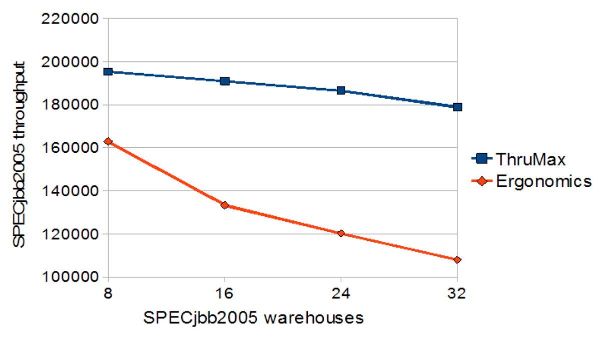

Figure 2. SPECjbb2005 throughput comparison for different load levels

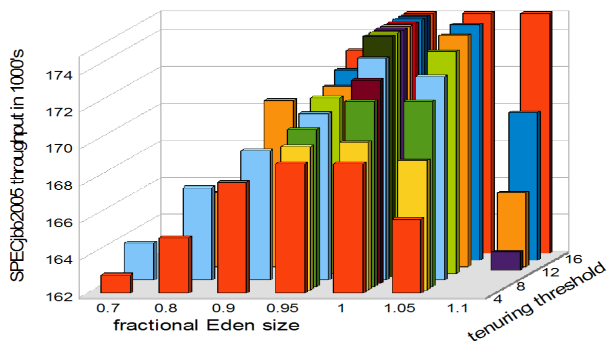

Figure 3. SPECjbb2005 throughput of the HotSpot JVM for different fixed values of the Eden size and tenuring threshold

tialized to be 1/3 of the heap size and the Eden size S as well as the size of each survivor space was initialized to be 1/3 of the Young generation, so as to make sure that no overflow of survivor spaces initially occurs (when almost all data collected from Eden consists of long-term objects that survive the minor GC and get placed in the survivor space). The tenuring threshold N was initialized to 1.

Figure 2 shows the SPECjbb2005 throughput for the ThruMax algorithm and for the HotSpot ergonomics policy when the number of SPECjbb2005 warehouses (threads) was specified using the sequence $\{n, n, n, n\}$ with $n = 8$, 16, 24, or 32. This sequence gave 4 throughput values (one value for each 4-minute interval), and the last 3 values were averaged to come up with the final value reported in Figure 2 (the first 4-minute interval was skipped to allow the GC policies to reach a steady state). Since the server used for these experiments had 8 cores, the highest throughput was observed for 8 warehouses and it then decreased because of the thread contention as the number of warehouses was increased.

Despite the large performance improvement of ThruMax over the current HotSpot ergonomics policy shown in Figure 2, a question still remains whether ThruMax indeed performs gradient ascent in the parameter space (with the size of Eden and the tenuring threshold being the parameters) and converges to the parameter values that actually maximize the GC throughput. Figure 3 shows a 3-D bar chart where the x- and the y- coordinates of each bar are the size of Eden and the tenuring threshold and the z-coordinate (height) is the SPECjbb2005 throughput that was observed for the fixed size of Eden and tenuring threshold corresponding to the x- and y- coordinates under the warehouse sequence $\{32, 32, 32, 32\}$. The size of Eden is actually given as a fraction of the size to which ThruMax has converged for this sequence of warehouses during the experiment reported in Figure 2, which was 861 MB (and the corresponding survivor space size was 158 MB). As the size of Eden was changed in this experiment, the survivor space size was changed proportionately. The tenuring threshold to which ThruMax has converged for this sequence of warehouses was 9. Figure 3 shows that this tenuring threshold and the Eden size of 861 MB indeed correspond to the maximum SPECjbb2005 throughput of 175000.

The final experiment focused on the behavior of the ThruMax algorithm for the case when the GC load was increasing. In such a case, the amount of data surviving the major GC increases, and hence the size of the young generation should be continually decreased. Otherwise, the garbage collector will get into a very low-throughput regime where only major GCs are occurring. The load increase was modeled using the sequence of threads (warehouses) $\{1, 2, 3, ..., 32\}$. During this experiment, the ThruMax algorithm first increased the size of Eden until it reached 1460 MB for 12 warehouses, and then started decreasing it until it reached 741 MB for 32 warehouses. The tenuring threshold was steadily increased from 1 until 13. Figure 3 shows that the point specified by Eden = 741MB and tenuring threshold = 13 results in a near-optimal throughput and is very close to the point to which ThruMax converged when the constant sequence of 32 warehouses was used. This suggests that the ThurMax algorithm is robust in its parameter adaptation with respect to different paths the workload can take as it reaches a particular level, which is a very desirable behavior for an algorithm.

9. Conclusions and future work

This paper presented the first mathematical analysis of how the throughput of a generational garbage collector used in the HotSpot JVM depends on key parameters such as Eden size and tenuring threshold. This analysis was then used in Section 5 to establish that the optimal value of the tenuring threshold can be either finite or infinite depending on the average object lifetime and on the free space in Old after a major GC relative to the amount of data surviving a major GC. In the unlikely case of the average object lifetime being VERY large, the optimal tenuring threshold is $N = 0$.

Section 6 showed that if the size of Eden S can be increased without affecting the size of Old, then the GC throughput will increase for applications that have a smooth steady state behavior. For the case when the total heap size is fixed and the Eden size has to be traded off against the size of Old, the derived shapes of key GC functions (such as minor GC time and the amount of data promoted into Old at each minor GC as a function of the Eden size) were used to conclude in Section 6 that the optimal Eden size is expected to be greater than half of the free heap space (heap size minus the amount of data surviving the major GC). Finally, the analytical expression for the GC throughput was used to derive the ThruMax algorithm, which continually performs numerical estimation of the gradient of GC throughput with respect to the tunable parameters (Eden size and tenuring threshold) and takes a step in the gradient direction. Therefore, this algorithm is capable of dynamically adapting generation sizes and tenuring threshold so as to keep maximizing the GC throughput in the face of dynamically changing workload properties.

The ThruMax algorithm was evaluated using the SPECjbb2005 benchmark, which represents a large and important class of customer workloads. The experimental results showed that ThruMax significantly outperforms the current HotSpot ergonomics policy, with the difference between them increasing as the application load increases. Moreover, a plot of the throughput surface for different values of the tunable parameters showed that ThruMax indeed converged to the optimal parameter values. Finally, when the application's load was allowed to increase over time, the ThruMax algorithm was able to quickly reduce the young generation size and avoid the situation when only the major garbage collections are occurring.

Acknowledgments

The author would like to thank Steven Heller and especially Randy Smith from Sun Microsystems Laboratories for helpful comments and suggestions on this paper.

References

[1] "Tuning Garbage Collection with the 5.0 Java^{TM} Virtual Machine," http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html.

[2] D. A. Barrett, B. G. Zorn. "Using lifetime predictors to improve memory allocation performance." In Proceedings of the ACM SIGPLAN 1993 conference on Programming Language Design and Implementation, pp 187 – 196.

[3] D. A. Barrett and B. G. Zorn. "Garbage collection using a dynamic threatening boundary." In Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, pp. 301 – 314.

[4] T. Brecht, E. Arjomandi, C. Li, H. Pham. "Controlling garbage collection and heap growth to reduce the execution time of Java applications." In Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, pp. 353 – 366, 2001.

[5] S. M. Blackburn, M. Hertz, K. S. McKinley, J. E. B. Moss, T. Yang. "Profile-based pretenuring." *ACM Transactions on Programming Languages and Systems*, Vol. 29, Issue 1, January 2007.

[6] A. Demmers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, S. Shenker. "Combining generational and conservative garbage collection: framework and implementations," In Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 261 – 269, 1989.

[7] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley & Sons, 1996.

[8] T.L. Harris. "Dynamic Adaptive Pre-Tenuring." In Proceedings of The 2000 International Symposium on Memory Management (ISMM 2000), pp. 127 – 136.

[9] J. Singer, G. Brown, M. Lujn, I. Watson. "Towards intelligent analysis techniques for object pretenuring." In Proceedings of the 5th international symposium on Principles and practice of programming in Java (PPPJ 2007), pp. 203 – 208.

[10] D. Ungar and F. Jackson. "An Adaptive Tenuring Policy for Generation Scavengers." *ACM Transactions on Programming Languages and Systems*, Vol. 14, No 1, pp. 1 – 27, January 1992.

[11] T. Yang, E. D. Berger, S. F. Kaplan, J. E. B. Moss. "CRAMM: virtual memory support for garbage-collected applications." In Proceedings of the 7th Symposium on Operating Systems Design and Implementation, pp. 103 – 116, 2006.