TECHNICAL REPORT

Adaptive Optimization of the Java™ Real-Time System Garbage Collector

Qian Zhu and David Vengerov

> *Sun Microsystems Laboratories*

Adaptive Optimization of the Sun Java™ Real-Time System Garbage Collector

Qian Zhu and David Vengerov

SMLI TR-2009-186 September 2009

Abstract:

Garbage collection (GC) is one of the largest sources of unpredictability in Java™ applications, and a real-time virtual machine must use garbage collection algorithms that minimize delays to real-time threads and at the same time maximize the overall application's throughput. In order to achieve the optimal tradeoff between these conflicting objectives, the GC cycle (which needs to take place periodically in order to free the memory no longer used by the application) needs to be triggered at the optimal time: if it is triggered too soon then the application's throughput will decrease unnecessarily, while if it is triggered too late then the application can run out of free memory and block real-time threads unnecessarily. Starting with Sun Java Real-Time System 2.0 (Java RTS), a new real-time garbage collector (RTGC) is available. One of the key RTGC parameters is the *StartupMemoryThreshold,* which determines how low the free memory in the system can fall before a garbage collection is triggered. This paper presents a framework for dynamically adapting the *StartupMemoryThreshold* for achieving the optimal balance between the application's throughput and pause time, which was integrated into the beta release of Java RTS 2.2. An experimental evaluation of this framework using the SPECjbb2005 benchmark confirmed its effectiveness. This framework can be used in conjunction with any concurrent or a time-based incremental garbage collector.

Sun Labs
16 Network Circle
Menlo Park, CA 94025

email address:
david.vengerov@sun.com

© 2009 Sun Microsystems, Inc. All rights reserved. Sun, Sun Microsystems, the Sun logo, Java, Solaris and JDK are trademarks or registered trademarks of Sun Microsystems, Inc. or its subsidiaries in the U.S. and other countries. Information subject to change without notice.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

For information regarding the SML Technical Report Series, contact Mary Holzer or Nancy Snyder, Editors-in-Chief <Sun-Labs-techrep-request@sun.com>. All technical reports are available online on our website, http://research.sun.com/techrep/.

Adaptive Optimization of the Java RTS Garbage Collector

Qian Zhu and David Vengerov
Sun Microsystems Laboratories
UMPK16-160
16 Network Circle
Menlo Park, CA 94025
contact: david.vengerov@sun.com

1 Abstract

Garbage collection (GC) is one of the largest sources of unpredictability in Java^{TM} applications, and a real-time virtual machine must use garbage collection algorithms that minimize delays to real-time threads and at the same time maximize the overall application's throughput. In order to achieve the optimal tradeoff between these conflicting objectives, the GC cycle (which needs to take place periodically in order to free the memory no longer used by the application) needs to be triggered at the optimal time: if it is triggered too soon then the application's throughput will decrease unnecessarily, while if it is triggered too late then the application can run out of free memory and block real-time threads unnecessarily. Starting with Sun Java RTS 2.0, a new real-time garbage collector (RTGC) is available. One of the key RTGC parameters is the *StartupMemoryThreshold*, which determines how low the free memory in the system can fall before a garbage collection is triggered. This paper presents a framework for dynamically adapting the *StartupMemoryThreshold* for achieving the optimal balance between the application's throughput and pause time, which was integrated into the beta release of Java RTS 2.2. An experimental evaluation of this framework using the SPECjbb2005 benchmark confirmed its effectiveness. This framework can be used in conjunction with any concurrent or a time-based incremental garbage collector.

2 Introduction

There is a considerable interest in Java as a software development platform for real-time and embedded applications. The Java Real-Time System (Java RTS) [15] has emerged, which offers tools for specification, scheduling, and management of real-time applications. There is a general agreement that an efficient and predictable execution of such real-time applications is necessary for the Java RTS to claim the real-time performance.

Essentially, a garbage collector (GC) suitable for real-time applications must be able to recycle memory without denying CPU resources to any real-time threads. Furthermore, a real-time GC must be fast enough so that real-time threads will not be blocked by memory exhaustion when they attempt to allocate more memory. Therefore, the GC should be granted enough processor cycles to complete its work on time. On the other hand, granting too many processor cycles to garbage collection work reduces the application's overall throughput.

Starting from Java RTS 2.0, a Real-Time Garbage Collector (RTGC) is available. One of the key parameters in RTGC is the *StartupMemoryThreshold*, which specifies how low the free memory can fall before RTGC is invoked. A very low setting of this parameter would increase the application's throughput (since more time would be devoted to application's work) but would lead to the risk of the free memory falling below *CriticalReservedBytes*, at which point all non-critical threads would be blocked. On the other hand, a very high setting of the *StartupMemoryThreshold* will hurt the overall application's throughput since RTGC will be invoked too frequently and will thus consume too many CPU resources. Therefore, an automated algorithm should ideally be used for selecting the appropriate *StartupMemoryThreshold* for each application, and its value should be dynamically adjusted as the application's behavior changes. Such an algorithm is presented in this paper, which takes into account both the variation in the amount of free memory consumed during each GC cycle for a constant memory allocation rate and the nonlinear relationship between the observed memory allocation rate and the *StartupMemoryThreshold*.

The rest of the paper is organized as follows. A background of the field and a discussion of the related work is provided in the section 3. The *StartupMemoryThreshold* adaptation problem is defined in Section 4. This section also describes the algorithm used in Java RTS 2.0 for adapting *StartupMemoryThreshold* and discusses its shortcomings. Section 5 describes an improved Threshold Adjustment Algorithm suitable for a constant or a decreasing workload. Section 6 presents the Adaptive Optimization Framework, which dynamically detects increases in the workload and appropriately re-adjusts parameters of the Threshold Adjustment Algorithm. Results from experimental evaluation are reported in Section 7. Section 8 concludes the paper.

3 Related Work

At a high level, any garbage collector works by first distinguishing live objects from dead objects (garbage) and then reclaiming the memory space used by the dead objects, making it available for the future memory allocation needs. The first step can be done either using a *tracing* or a *reference counting* approach [17]. A reference counting collector maintains a count of the number of pointers to each object. If this count drops to 0, then the object is declared to be dead. A tracing collector is invoked occasionally and traverses the pointers between live objects starting from those objects that are assumed to be live at the present moment (also known as roots), which include objects referenced from anywhere in the call stack (that is, all local variables and parameters in the functions currently being invoked), and any global variables. The objects that are not reached by traversing all such pointers are assumed to be dead.

The main advantage of reference counting over tracing garbage collection is that objects are reclaimed as soon as they can no longer be referenced, and in an incremental fashion, without long pauses for collection cycles. The main disadvantage of the reference counting approach is that, in its simple form, it will never consider cyclic chains of objects for deletion, since their reference count is guaranteed to stay nonzero. Methods for dealing with this issue exist but they also increase the overhead and complexity of a reference counting collector. A good discussion contrasting reference counting and tracing approaches is presented in [2], which shows that these algorithms converge towards each other if efficiency of implementation is desired.

Real-time systems require very small pause times for applications so that all critical tasks can be executed in a timely fashion. The reference counting approach can be viewed as an instance of a "work-based" approach to garbage collection in real-time systems. In this approach, the GC work is distributed over that of the application activities by performing a bounded amount of GC work at

each memory allocation request (see [3] and [16] for some examples of this idea). The disadvantage of this approach for real-time systems is that the worst-case amount of work that needs to be performed by every application (mutator) thread is large, and hence the mutator threads need to be given much larger time slices than they require on average in order to make sure that they finish their execution on time to meet the deadlines. Thus, as pointed out in [5], the last decade witnessed a shift of interest from the work-based GC approach to a "time-based" GC approach in real-time systems.

The time-based approach implements the GC process as a separate thread, whose execution can be interleaved with that of the mutator thread on a single CPU (which makes this approach especially suitable for embedded systems). In order to do that, the GC algorithm needs to be made "incremental" so as to allow it to make some progress every time the GC thread is executed for a short amount of time. Some example of the incremental time-based GC algorithms can be found in [1, 9, 10].

When the trend of building multi-core systems emerged in recent years, it became apparent that stopping all application threads running on such systems in order to do GC work is impractical. Therefore, researchers started focusing on designing *concurrent* garbage collectors that can run in parallel (either as a single GC thread or as multiple GC threads) with multiple application threads. The tracing algorithms discussed above were adapted to make them work while the applications are modifying objects in the heap [8, 11, 13]. The real-time garbage collector (RTGC) used by default in Java RTS 2.0 is another example of a concurrent collector.

While a lot of thought has been dedicated to the structure of concurrent collectors, there has been almost no discussion of when the GC cycle should be initiated. In fact, most of the research papers describing concurrent and time-based incremental GC algorithms do not even mention this issue at all. This is not surprising considering that there is no obvious *optimal* solution for this problem. The papers that do describe the triggering process for such collectors all suggest starting a GC cycle when the free memory drops below a certain fixed threshold (e.g., [1, 4, 6, 10, 12]). None of them take into account the possibility of the application's memory allocation rate changing over time, which would in turn change the amount of memory consumed during the GC cycle and hence affect the optimal starting time for the GC cycle that would guarantee that the application does not run out of memory before the cycle ends while at the same time delaying the cycle as much as possible so as to maximize the application's throughput. The Adaptive Optimization Framework proposed in this paper does assume that the memory allocation can vary over time and keeps adjusting the threshold for starting the GC cycle so as to maximize the application's throughput while trying to ensure it does not run out of memory.

4 Problem Definition

When an application is running, it allocates memory space for its objects and data structures. Thus, the available memory decreases over time and should periodically be recycled (by invoking the garbage collector, RTGC) so as to release the space occupied by objects that are no longer used. The goal of Java RTS is to enable good performance for real-time threads, delaying their execution as little as possible. To achieve this goal, Java RTS dynamically determines the priority at which the RTGC worker thread is executed.

The relative priority of different threads is illustrated in Figure 1. When the free memory in the Java heap drops below the *StartupMemoryThreshold*, RTGC is invoked at the *Normal Priority*, where non real-time threads running on a CPU are preempted by the RTGC thread running on that CPU (if the number of RTGC threads is smaller than the number of CPUs, then non real-time threads can keep running in parallel with RTGC threads on different CPUs). This temporarily decreases the

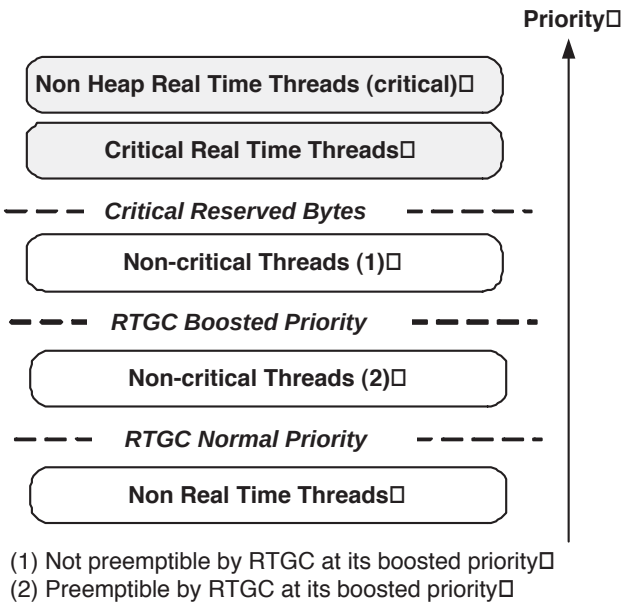

Figure 1. RTGC Priorities

application's throughput. The first phase of RTGC involves marking as "dead" the objects in the heap that are no longer used by the application. The second phase of RTGC involves adding heap regions with only the "dead" objects to the list of available heap regions where new objects can be allocated. Therefore, the available heap memory keeps falling during the first phase of RTGC. If it falls below the *BoostedMemoryThreshold*, then the RTGC priority is increased to *Boosted Priority*, where it takes precedence over some non-critical real-time threads (i.e., over threads with soft real-time deadlines). If the available memory further falls below the *CriticalReservedBytes*, then all non-critical real-time threads are blocked to protect the critical real-time threads with hard deadlines, which really hurts some real-time applications that depend on quick execution of some tasks.

In order to have a high-level of determinism (i.e., avoid blocking real-time threads), the *StartupMemoryThreshold* can be set to a high value so as to ensure that the free memory will not fall below the *BoostedMemoryThreshold*. However, this will lead to RTGC being invoked more frequently, which will reduce the average fraction of time that the non real-time threads are working, thereby decreasing the overall application's throughput. For soft real-time scheduling, some customers care more about the overall application's throughput and some care more about the determinism (the degree to which they can be sure that important tasks can be executed without delay).

The algorithm used in Java RTS 2.0 for adjusting the *StartupMemoryThreshold* works as follows [7]:

$$Expected_mem_drop_{new} = MAX(Mem_drop, (Expected_mem_drop_{previous} \cdot (100 - SlideFactor)/100)) \quad (1)$$

$$StartupMemoryThreshold = MAX(MinFreeBytes, (Expected_mem_drop_{new} \cdot (100 + SafetyMargin)/100)) \quad (2)$$

This calculation is performed at the end of each RTGC cycle. In the above formulas, the variables have the following meanings:

- *Mem_drop* represents the amount by which the free memory dropped during the latest GC cycle (measured relative to the free memory in the system at the beginning of the cycle).
- *Expected_mem_drop* represents a prediction of the *Mem_drop* during the next GC cycle.
- *SlideFactor* is an integer converted to a percentage value by which *Expected_mem_drop* is reduced after each GC cycle.
- *SafetyMargin* is an integer converted to a percentage value by which *Expected_mem_drop* is increased in order to arrive at the final value of the *StartupMemoryThreshold* that would guarantee (with a high probability) that the free memory will not be exhausted before the end of the GC cycle despite the possible variations in the memory allocation rate.
- *MinFreeBytes* is the user-estimated amount of free memory below which the RTGC should always be invoked, which is usually set to *BoostedMemoryThreshold* plus some user-chosen safety margin.

The parameters *SlideFactor*, *SafetyMargin*, and *MinFreeBytes* have some default values but can also be configured by an expert user.

While the above algorithm does adjust the *StartupMemoryThreshold* dynamically based on the amount of free memory consumed during the previous GC cycle, it requires using a large *SafetyMargin* in order to guarantee (with a high probability) that the free memory will not be exhausted before the end of the GC cycle, since the actual variance in *Mem_drop* is not computed. More importantly, this algorithm does not consider the changes that should be made to the *StartupMemoryThreshold* if a change in the memory allocation rate is detected. Both of these shortcomings are addressed by the new algorithms proposed in Sections 5 and 6 below.

5 Threshold Adjustment Algorithm

This section describes the new algorithm for adjusting the *StartupMemoryThreshold* proposed in this paper. A strategy for dynamically adapting parameters of this algorithm based on the observed memory allocation rates will be described in Section 6.

The pseudo code for the Threshold Adjustment Algorithm is given in Figure 2. This algorithm is executed after each RTGC invocation and estimates the *StartupMemoryThreshold* for the next RTGC cycle. A detailed description of the algorithm's steps is given below.

Step 1: The algorithm starts with parameter initialization. While $threshold_1$ is set by default to 1/2 of the heap size, any other initial setting can be used.

Step 2: Incrementally update the mean and the standard deviation of the *Mem_drop* values observed so far, which are denoted as *Mem_drop_avg* and *Mem_drop_stdev*.

Step 3: Reduce $threshold_1$ by a factor of 2 (or by any other factor) so as to have a quick convergence to a stable value. In order to make sure that the free memory does not drop below the *BoostedMemoryThreshold*, $threshold_2$ is set to *BoostedMemoryThreshold* plus a safety margin that depends on the previously observed volatility of the *Mem_drop* variable. The variable *margin* in Step 3 was set to a constant value of 0.1.

Step 4: Set *StartupMemoryThreshold* to the larger of $threshold_1$ and $threshold_2$. Go to **Step 2.**

Algorithm 5.1: THRESHOLDADJUSTMENT(ss)

```
// Step 1: Initialize algorithm
threshold_1 = HeapSize/2;
StartupMemoryThreshold = threshold_1;
Mem_drop_avg = Mem_drop_stdev = count = 0;
Sum_mem_drop = Sum_mem_drop_squared = 0;
// Step 2: Measure memory drop and calculate its average and
// standard deviation over the last "count" instances
count = count + 1;
Mem_drop = mem_at_RTGC_start − min_mem_during_RTGC;
Sum_mem_drop = Sum_mem_drop + Mem_drop;
Sum_mem_drop_squared = Sum_mem_drop_squared + Mem_drop^2;
Mem_drop_avg = Sum_mem_drop/count;
Mem_drop_stdev = sqrt(Sum_mem_drop_squared/count − Mem_drop_avg^2);
// Step 3: Calculate two temporary thresholds
threshold_1 = threshold_1 × 0.5;
threshold_2 = BoostedMemoryThreshold + (Mem_drop_avg+
          Mem_drop_stdev) × (1 + margin);
// Step 4: Set the value for the StartupMemoryThreshold
if threshold_1 > threshold_2
    then
        startup_threshold = threshold_1;
    else{
        startup_threshold = threshold_2;
        threshold_1 = startup_threshold;
    }
```

Figure 2. Pseudo-code for Threshold Adjustment Algorithm

The proposed Threshold Adjustment Algorithm is adaptive and can quickly reduce the RTGC overhead (by reducing the value of $threshold_1$ by 50% at each RTGC) while at the same time avoid reaching the *BoostedMemoryThreshold* and thus minimize the delay for all real-time threads.

6 Adaptive Optimization Framework

The Threshold Adjustment Algorithm presented in the previous section is executed until a sufficiently large increase in the application's memory allocation rate is detected (if the memory allocation rate is decreased, then the Threshold Adjustment Algorithm will automatically reduce the *StartupMemoryThreshold*). When that happens, the algorithm in Figure 2 is moved to Step 1 (during which the variables Mem_drop_avg, Mem_drop_stdev and $count$ are initialized to 0) and then a new value of $threshold_1$ is computed to be used for the next RTGC invocation. More details about this process are given below.

6.1 Detecting Workload Increase

In order to detect an increase in the application's memory allocation rate, we measure *MemoryAllocationRate* at 100-millisecond time intervals from the beginning of the application's execution by

recording the amount of free memory consumed during each such interval. These values are then stored in an array of size 20 (any other size can be used), and once the array fills up, its values get overwritten starting from the first one. This allows storing 20 most recent values of the *MemoryAllocationRate*. Once the array storing *MemoryAllocationRate* values is filled up, the mean and the standard deviation of the stored values is computed after a new observation is made and recorded in the array. If two samples of *MemoryAllocationRate* in a row are unusually large (i.e., greater than $mean + 2 \cdot stdev$ of the array values), then an increase in the memory allocation rate is assumed.

When such an increase is detected, the startup memory threshold *might* need to be increased, since more memory will be consumed between the time points when RTGC is invoked and when it actually starts releasing unused memory, thereby increasing the risk of violating the *BoostedMemoryThreshold*. Unfortunately, it is difficult to estimate the maximum memory drop during RTGC invocation based on the observed memory allocation rate alone. Our experiments showed that there is no obvious correlation between these two variables: when the memory allocation rate was doubled, the memory drop sometimes stayed almost the same and sometimes increased by a factor of 100.

In order to address this issue, we implemented a separate algorithm that gives a conservative estimate of the memory drop based on the previously observed memory drops and the corresponding memory allocation rates. This algorithm is presented below.

6.2 Estimating Memory Drop for a New Workload

In the proposed algorithm, the range of possible *mem_drop* sizes (which runs from 0 to the size of the whole heap) is divided into N parts, or "buckets", and an array of size N denoted as *mem_alloc_rates*[] is created, where each entry corresponds to one of these subranges. Each index of the *mem_alloc_rates*[] array represents a memory drop subrange and the content of the entry stores the memory allocation rate for which the memory drop is expected to fall in that range. Figure 3 shows an example where the heap space (2048MB) is divided into 100 parts. A *mem_alloc_rates*[] array of size 100 was created, with *mem_alloc_rates*[1] giving the minimum memory allocation rate for which the memory drop can be as high as 20MB, *mem_alloc_rates*[2] giving the minimum memory allocation rate for which the memory drop can be as high as $2 \cdot 20 = 40$MB, etc. The entry *mem_alloc_rates*[11] = 120 indicates that for the memory allocation rate of 120MB/s, the memory drop can be as high as $11 \cdot 20 = 220$MB.

When a particular memory drop (corresponding to some empty entry in the *mem_alloc_rates*[] array) is observed for the first time during some RTGC collection, the array entry is filled with the average of the 10 most recent memory allocation rate observations (made over a 1 second interval) that are recorded just before this collection began. If the location in the *mem_alloc_rates*[] array already had some value, this entry and some of its neighboring *mem_alloc_rates*[] entries are *possibly* updated using the following procedure. Let's denote the newly observed allocation rate by *rate_new* and the previously stored allocation rate by *rate_prev*. If $rate_prev < rate_new$, then we haven't gained any new information regarding the *worst case* memory drop for a particular allocation rate, and so no updating is performed. If $rate_prev > rate_new$, then the neighbors with smaller indexes are identified and the search stops when an entry in the *mem_alloc_rates*[] array that is smaller than *rate_new* is located. The memory allocation rates stored in the *mem_alloc_rates*[] array between these two indexes are then updated.

In the example of Figure 3, a new observation is made of the memory drop equal to 235MB when the memory allocation rate is 120MB/s. The observed memory drop is rounded off to the upper boundary of the subrange into which it fell (240MB in this case) and the *mem_alloc_rates*[] index corresponding to the 240MB memory drop is found (which is 12). After that, *mem_alloc_rates*[12] is updated to be 120 and the algorithm moves to the next smaller index 11. Since *mem_alloc_rates*[11] is equal to the

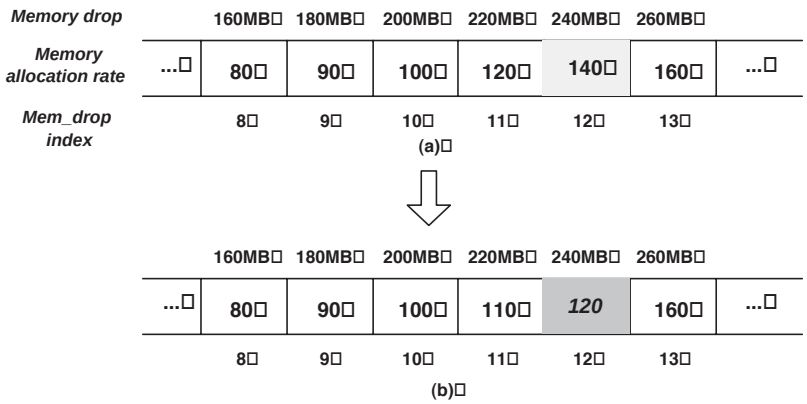

Figure 3. An example of how the $mem_alloc_rates[]$ array is updated when a memory drop of 235MB is observed for the memory allocation rate of 120MB/s

recently measured value of 120MB/s (but is not smaller!), $mem_alloc_rates[11]$ is also updated to a new value of 110, which is a half way point between its previous value of 120 and the value of 100 recorded in its nearest neighboring entry to the left. The logic behind this update is that since we have determined that the allocation rate of 120MB/s can cause a memory drop of up to 240MB, then we *could* observe a memory drop of up 220MB for a *smaller* allocation rate than 120MB/s. Thus, the entry in $mem_alloc_rates[11]$, which corresponds to the memory drop of 220MB, is decreased to be smaller than 120. After that we move one more entry to the left to $mem_alloc_rates[10]$ and find that its associated allocation rate is 100MB/s, which is smaller than the latest measured allocation rate of 120MB/s, and so the updating process stops.

6.3 Re-initializing the Threshold Adjustment Algorithm

When an increase in the memory allocation rate is detected using the procedure described in Section 6.1, the Threshold Adjustment Algorithm described in Figure 2 is moved to Step 1, during which its parameters are re-initialized. For example, if we observe that the memory allocation rate has increased from 80MB/s to 120MB/s, we locate the entry with the value of 120 (say it is entry number 12 in the updated $mem_alloc_rates[]$ array in Figure 3) and see that it corresponds to the memory drop of 240MB (which is the maximum memory drop that we are likely to observe for the allocation rate of 120MB/s). In general, let's denote the new expected memory drop as mem_drop_new. Then, both $threshold_1$ and $StartupMemoryThreshold$ are set to $BoostedMemoryThreshold + mem_drop_new$. After that, if the free memory in the system is smaller than the new value of the $StartupMemoryThreshold$ computed above, then a garbage collection should be triggered immediately. Otherwise, the Threshold Adjustment Algorithm proceeds in a normal fashion and triggers the next garbage collection when the free memory reaches $StartupMemoryThreshold$.

7 Experimental Evaluation

This section presents results from several experiments we conducted to evaluate the proposed Threshold Adjustment Algorithm and the Adaptive Optimization Framework surrounding this algorithm (which detects increases in application's workload and re-initializes the algorithm).

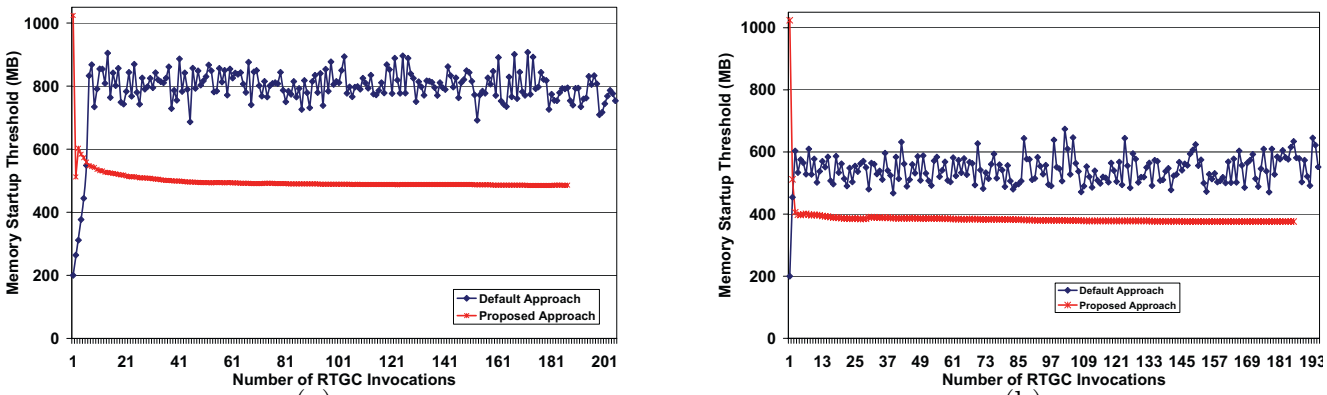

Figure 4. Evolution of the StartupMemoryThreshold for the Default RTGC Policy and for the Adaptive Optimization Framework for two constant workloads: (a) 4 warehouses (b) 8 warehouses

7.1 Experimental Setup and Goals

The experiments were carried out on a server with 4 CPU cores running at 2.20 GHz each and a 4 GB of memory. The server had the Solaris^{TM} 10 Operating System and JDK^{TM} version 6 installed on it. We used the SPECjbb2005 benchmark as the application, which represents a large and important class of customer workloads. 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. 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.

The *BoostedMemoryThreshold* parameter was set to 200MB for all experiments (creating a goal for our algorithm to keep the free memory above 200MB) and the total heap space was 2048MB. The proposed Adaptive Optimization Framework was compared with the current policy implementation in RTGC 2.0, which we referred to as the "Default RTGC Policy". In that policy, the startup memory threshold is estimated using the memory usage information from the previous RTGC cycle. More details can be found in [7].

7.2 Constant Workload

The experiment in this section was designed to evaluate the Threshold Adjustment Algorithm described in Figure 2 without requiring the Adaptive Optimization Framework to "kick in" and re-initialize this algorithm. This was achieved by using a constant SPECjbb2005 workload, so that no workload increases would be detected (and in fact none were detected in the experiments described in this section). Two SPECjbb2005 thread sequences were used: $\{4, 4, 4, 4, 4\}$ and $\{8, 8, 8, 8, 8\}$. One RTGC worker thread was used for the first sequence and two RTGC worker threads were used for the second sequence so as to speed up the garbage collection process in RTGC. The plots of the *StartupMemoryThreshold* for both of the thread sequences are shown in Figure 4 (a) and (b). These plots show that the proposed Threshold Adjustment Algorithm reduces both the mean and the standard deviation of the *StartupMemoryThreshold* while still keeping it above the *BoostedMemoryThreshold* of 200MB. A smaller *StartupMemoryThreshold* means that the start of RTGC is delayed, which leads to an increase in the application's throughput (since a larger fraction of time is spent on application's work rather than garbage collection). In general, when the workload is more intensive, RTGC should

	4 warehouses		8 warehouses	
	Default	Our Approach	Default	Our Approach
Throughput	11733.54	**14163.53**	10666.81	**12111.47**
CPU Overhead	0.75	**0.76**	1.08	**1.05**
# RTGC Invocations	205	**187**	195	**186**
# *BoostedMemoryThreshold* Violations	4	**0**	3	**0**

Table 1. Comparison between Default RTGC Policy and Our Approach for two constant workloads

be invoked earlier so as to prevent reaching the *BoostedMemoryThreshold*. However, the time average value *StartupMemoryThreshold* for the 8 warehouse sequence is smaller for both policies than for the 4 warehouse sequence. This is due to the fact that two RTGC worker threads were used in the 8 warehouse experiment, and a larger number of threads allows RTGC to recycle memory faster.

Another comparison between the proposed Threshold Adjustment Algorithm and the Default RTGC Policy is presented in Table 1, which compares the application's throughput, the CPU overhead of RTGC, the number of RTGC invocations and the number of times the free memory falls below the *BoostedMemoryThreshold*. As the table shows, the proposed Threshold Adjustment Algorithm can achieve 20.7% and 13.5% increase in application's throughput relative to the Default RTGC Policy in the 4 warehouse and 8 warehouse cases, respectively. While achieving a better throughput, the CPU overhead caused by RTGC under our framework is comparable to the CPU overhead of the Default RTGC Policy for both workloads. The number of RTGC invocations is smaller for the proposed Threshold Adjustment Algorithm because it converged to a lower *StartupMemoryThreshold* and thus delayed the invocations of the garbage collector. Furthermore, the *BoostedMemoryThreshold* was never violated for the Threshold Adjustment Algorithm, while the Default RTGC Policy had some violations for both the 4 warehouse and the 8 warehouse cases. This demonstrates that the proposed Threshold Adjustment Algorithm not only improves the application's throughput but also increases its determinism by avoiding delays to real-time threads that would arise once RTGC is boosted.

7.3 Dynamic Workload

The experiment in this section was designed to test how well the Adaptive Optimization Framework can detect increases in the application's workload and re-adjust the parameters of the Threshold Adjustment Algorithm. This was achieved by using an increasing SPECjbb2005 warehouse sequence $\{1, 2, 3, ..., 8\}$. A comparison between the Default RTGC Policy and the Adaptive Optimization Framework is shown in Table 2. The 8 values of throughput in the table are those obtained for the 8 successive values of the number of SPECjbb2005 warehouses implied by the test sequence. Table 2 shows that the Adaptive Optimization Framework has a higher throughput than the Default RTGC Policy for each warehouse case. Furthermore, the CPU overhead of the proposed framework is only 83% of the one observed for the Default RTGC Policy and about 20% fewer RTGC invocations were made under the Adaptive Optimization Framework. Most importantly, the *BoostedMemoryThreshold* is never violated for the proposed framework while being violated 39 times for the Default RTGC Policy. Therefore, just like the Threshold Adjustment Algorithm evaluated in Section 7.2, the complete Adaptive Optimization Framework also increases both the application's throughput and its determinism.

	warehouse sequence { 1 2 3 4 5 6 7 8 }	
	Default	Our Approach
Throughput	5030.8	**5219.21**
	8421.17	**9657.16**
	11984.27	**13353.46**
	13086.52	**14075.18**
	12358.68	**13528.59**
	12460.42	**12724.67**
	11481.37	**11916.57**
	10355.9	**11208.01**
CPU Overhead	1.14	**0.95**
# RTGC Invocations	249	**199**
# *BoostedMemoryThreshold* Violations	39	**0**

Table 2. Comparison between Default RTGC Policy and the Adaptive Optimization Framework: increasing workload

8 Conclusion

This paper proposed a Threshold Adjustment Algorithm that automatically adjusts the *StartupMemoryThreshold* in the Java RTS garbage collector (RTGC). Experimental evaluation of this algorithm using a constant workload showed that it both increases the application's throughput and eliminates the delay to real-time threads relative to the threshold adjustment policy currently used in Java RTS 2.0. If the workload decreases over time, then the Threshold Adjustment Algorithm keeps reducing the *StartupMemoryThreshold* by any specified fraction at every RTGC invocation. In order to handle cases when the application's workload increases or oscillates over time, an Adaptive Optimization Framework was developed that detects increases in the application's memory allocation rate and appropriately re-adjusts parameters of the Threshold Adjustment Algorithm. Experimental evaluation of this framework was conducted using an increasing SPECjbb2005 workload, and the results showed a noticeable improvement in the SPECjbb2005 throughput and an increased determinism, as no delay was caused by RTGC to the real-time threads. The Adaptive Optimization Framework presented in this paper can be used in conjunction with any concurrent or a time-based incremental garbage collector.

9 Acknowledgments

The authors would like to thank Bertrand Delsart from the Java RTS team at Sun Microsystems for explaining the dynamics of the Java RTS garbage collector and answering all our questions about it that arose during our work.

References

[1] D.F. Bacon, P. Cheng, and V.T. Rajan, "A Real-Time Garbage Collector with Low Overhead and Consistent Utilization," In Proceedings of the ACM Principles of Programming Languages (POPL 03), pp. 285-298, 2003.

[2] D. F. Bacon, P. Cheng, and V. T. Rajan. "A Unified Theory of Garbage Collection." In Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004), pp. 50 – 68.

[3] H. G. Baker. "The treadmill: real-time garbage collection without motion sickness." *ACM SIGPLAN Notices*, Volume 27, No. 3, pp. 66–70, 1992.

[4] K. Barabash, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, Y. Ossia, A. Owshanko and E. Petrank, "A parallel, incremental, mostly concurrent garbage collector for servers," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 27, Issue 6, pp. 1097–1146, November 2005.

[5] D. Detlefs, "A Hard Look at Hard Real-Time Garbage Collection," In Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04), pp.23–32, 2004.

[6] D. Detlefs, C. Flood, S. Heller and T. Printezis, "Garbage-first garbage collection," In Proceedings of the 4th International Symposium on Memory Management, pp. 37-48, 2004.

[7] Garbage Collection Guide, Sun Java Real-Time System 2.1 http://java.sun.com/javase/technologies/realtime/reference/doc_2.1/release/JavaRTSGarbageCollection.html

[8] T. H. Heil, J. E. Smith. "Concurrent garbage collection using hardware-assisted profiling," In Proceedings of the 2nd International Symposium on Memory Management, pp. 80–93, 2000.

[9] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems.* PhD thesis, Lund Institute of Technology, July 1998.

[10] T. Kim, N. Chang, N. Kim, and H. Shin. "Scheduling Garbage Collector for Embedded Real-Time Systems." In Proceedings of the ACM SIGPLAN 1999 workshop on Languages, Compilers, and Tools for Embedded Systems, pp. 55–64, 1999.

[11] P. McGachey, A.-R. Adl-Tabatabai, R. L. Hudson, V. Menon, B. Saha and T. Shpeisman. "Concurrent GC leveraging transactional memory," In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 217–226, 2008.

[12] H. Paz, D. F. Bacon, E. K. Kolodner, E. Petrank and V. T. Rajan. "An efficient on-the-fly cycle collection," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 29, Issue 4, August 2007.

[13] F. Pizlo, E. Petrank and B. Steensgaard. "A study of concurrent real-time garbage collectors," *ACM SIGPLAN Notices*, Vol. 43 No. 6, June 2008

[14] T. Printezis and D. Detlefs, "A generational mostly-concurrent garbage collector," In Proceedings of the 2nd International Symposium on Memory Management, pp.143–154, 2000.

[15] Sun Java Real-Time System 2.1. http://java.sun.com/javase/technologies/realtime/reference/doc_2.1/release/JavaRTSReadme.html

[16] S. Stanchina and M. Meyer. “Mark-sweep or copying?: a ”best of both worlds” algorithm and a hardware-supported real-time implementation,” In Proceedings of the 6th international symposium on Memory management, pp. 173-182, 2007.

[17] P. Wilson. “Uniprocessor Garbage Collection Techniques.” In Proceedings of the International Workshop on Memory Management, pp. 1–42, 1992.

About the Authors

David Vengerov is a staff engineer at Sun Microsystems Laboratories. He is the principal investigator for the Adaptive Optimization project, which aims at developing and implementing self-managing and self-optimizing capabilities in computer systems. David holds a Ph.D. in Management Science and Engineering from Stanford University, an M.S. in Engineering Economic Systems and Operations Research from Stanford University, an M.S. in Electrical Engineering and Computer Science from MIT and a B.S. in Mathematics from MIT.

Qian Zhu is a Ph.D. candidate at the Computer Science and Engineering department of the Ohio State University. Her research focuses on adaptive middleware for supporting time-critical event processing in a grid/cloud computing environment. In particular, she is interested in applying machine learning algorithms for self-management and self-optimization of distributed systems. Qian holds a M.S. in Computer Science and Engineering from the Ohio State university and a B.S. in Computer Science and Engineering from Beijing Information and Technology Institute, P.R. China.

Sun Microsystems Laboratories
16 Network Circle
Menlo Park, CA 94025

Adaptive Optimization of the Java™ Real-Time System Garbage Collector

Qian Zhu and David Vengerov

SMLI TR-2009-186