Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 70 (2008) 89–110
Science of Computer Programming
www.elsevier.com/locate/scico

Garbage collection: Java application servers' Achilles heel

Feng Xian*, Witawas Srisa-an, Hong Jiang

Computer Science and Engineering, University of Nebraska-Lincoln, 256 Avery Hall, Lincoln, NE, 68588-0115, USA

Received 1 December 2006; received in revised form 1 February 2007; accepted 11 July 2007
Available online 24 October 2007

Abstract

Java application servers are gaining popularity as a way for businesses to conduct day-to-day operations. While strong emphasis has been placed on how to obtain peak performance, only a few research efforts have focused on these servers' ability to sustain top performance in spite of the ever-changing demands from users. As a preliminary study, we conducted an experiment to observe the throughput degradation behavior of a widely-used Java application server running a standardized benchmark and found that throughput performance degrades ungracefully. Thus, the goal of this work is three-fold: (i) to identify the primary factors that cause poor throughput degradation, (ii) to investigate how these factors affect throughput degradation, and (iii) to observe how changes in algorithms and policies governing these factors affect throughput degradation.
Published by Elsevier B.V.

Keywords: Application servers; Throughput degradation; Garbage collection; Performance analysis

1. Introduction

Web applications have recently become a method of choice for businesses to provide services and gain visibility in the global marketplace. For example, eBay has over 180 million users worldwide. In addition, the advent of a suite of Web applications from Google[1] also propels the sophistication of Web services and applications to a new height. The enabling software that allows applications to be served through the Web is referred to as application servers. Industry observers expect applications servers to generate revenue of about $6 billion by the year 2010 [1].

Two of the most adopted application server technologies are based on Java and .NET, which occupy about 70% of application servers market share (40% for Java and 30% for .NET) [2]. The major reason for such popularity is due to the rich set of libraries and features provided by these technologies, which promotes quick development and short time-to-market. On average, such technologies often reduce the code size and development cycle by 50% when compared to older technologies such as CGI written in C/C++ [3].

Application servers often face significant variations in service demands—the higher demands often coincide with "the times when the service has the most value" [4]. Thus, these servers are expected to maintain responsiveness, robustness, and availability regardless of the changing demands. However, the current generation of application servers

* Corresponding author.
E-mail addresses: fxian@cse.unl.edu (F. Xian), witty@cse.unl.edu (W. Srisa-an), jiang@cse.unl.edu (H. Jiang).

[1] These applications include office productivity, graphics, and maps (see www.google.com).

0167-6423/$ - see front matter Published by Elsevier B.V.
doi:10.1016/j.scico.2007.07.008

is not well-equipped to meet such expectations, as they often fail under heavy workload. For example, on the day that Apple announced the release of its Video Ipod, the Apple store site was down for over one hour due to heavy traffic [5]. In addition, these servers are susceptible to Distributed Denial of Service (DDoS) attacks. One notable example is when a group of Korean high school students launched a DDoS attack on a university's website to prevent other students from applying [6]. While application server technologies continue to be widely adopted, the knowledge of why these servers fail and how to prevent them from failing is still elusive.

1.1. This work

To date, very little research has been conducted on the throughput degradation behavior of Java application servers [7,8]. Specifically, very little information is known about the system's behavior under stress. We have conducted experiments and found that a relatively small change in client's requests (20% increase) can cause throughput to drop by as much as 75%. This paper reports the results of our extensive study to investigate the reasons behind such a poor throughput degradation behavior of Java application servers. There are three major contributions resulting from our work.

(1) Identify the primary factors that cause poor throughput degradation.
(2) Investigate the effects of these factors on throughput.
(3) Observe how changes of algorithms and policies utilized in these factors affect throughput degradation.

The remainder of this paper is organized as follows. Section 2 briefly describes pertinent background concepts related to this work. Section 3 details our experiments to identify opportunities for improvement. Section 4 details our experimentation plan. Sections 5 and 6 report the results and discuss the possible improvements. Section 7 highlights related work. Section 8 concludes the paper.

2. Background

2.1. Garbage collection overview

One of the most useful language features of modern object-oriented programming languages is garbage collection (GC). GC improves programming productivity by reducing errors resulting from explicit memory management. Moreover, GC underpins sound software engineering principles of abstraction and modularity. GC leads to cleaner code since memory management concerns are no longer cluttered with the programming logic [9,10]. For our purpose, we summarize three garbage collection schemes that are related to this paper: mark-sweep, generational, and reference counting. Refer to [10,11] for a comprehensive summary of each garbage collection algorithm.

Mark and sweep collection [12] consists of two phases: *marking* and *sweeping*. In the marking phase, the collector traverses the heap and marks each of the reachable objects as live. The traversal usually starts from a set of *roots* (e.g. program stacks, statically allocated memory, and registers) and results in a transitive closure over the set of live objects. In the sweeping phase, the memory is exhaustively examined to find all the unmarked (garbage) objects and the collector "sweeps" their space by linking them into a free list. After sweeping, the heap can be compacted to reduce fragmentation.

Generational garbage collection [10,13] segregates objects into "generations" using age as the criterion. The generational collection exploits the fact that objects have different lifetime characteristics; some objects have a short lifespan while others live for a long time. As far as distribution, studies have shown that "most objects die young" (referred to as the weak generational hypothesis [13,10]). Thus, the main motivation is to frequently collect the youngest generation, which is only a small portion of the heap. Collection in the young generation is referred to as *minor* and collection of the entire heap is referred to as *major* or *full*. Since most of the generational collectors are copying-based (refer to [10] for more information), small volumes of surviving objects translate to short garbage collection pauses because there are fewer number of objects to traverse and copy.

Reference counting (*RC*) [14] records the number of references to an object in its reference count field (often resides in the object's header [15,10]). The counter is initialized to zero when the object is allocated. Each time a pointer to that object is copied, the reference count is incremented, and each time a pointer to that object is removed, the reference count is decremented. When the reference count reaches zero, the object is reclaimed. This approach suffers

from the inability to reclaim cyclic structures. Each structure is purely self-referential, which represents memory space that cannot be reclaimed during program execution. Because of this limitation, reference counting is generally accompanied by a back-up tracing collector [10] or a complex algorithm to break up and detect the cyclic structures [16,17].

2.2. Merlin algorithm

To explore the design and evaluation of GC algorithms quickly, researchers often use trace-driven simulation. The most accurate way is the brute force method, which generates "perfect traces" by invoking whole-heap GC at every potential GC point in the program (e.g. after each allocation request). But this process is prohibitively expensive, and thus, granulated traces resulting from invoking the garbage collector periodically are used instead. Unfortunately, a study by Hertz et al. [18] reports that different granularities in GC invocations can produce significantly different results.

To address the efficiency problems of the brute force method and the accuracy problems of the granulated traces, Hertz et al. [18] propose the *Merlin trace generation algorithm*. The Merlin algorithm records the timestamp of each live object and later uses the timestamps to reconstruct the time at which the object died. Because it uses timestamps rather than collections to identify time of death, the new algorithm does not require frequent collections. Rather, it makes use of normal collections to identify which objects have died and then uses timestamps to identify when they died. Ordering the dead objects from the latest timestamp to the earliest, the algorithm works from the current time backwards. Therefore, it can determine when each object was last known to be alive, saving further analysis of the object. By avoiding frequent collections, the Merlin algorithm can make perfect tracing efficient and alleviate the need for granulated tracing. The Merlin algorithm is used to generate lifetime information in our experiments.

2.3. Vertical profiling

An investigation by Hauswirth et al. [19] discovers that it has become more difficult to understand the performance of modern object-oriented systems. Thus, they propose *virtical profiling*, a performance analysis technique that examines multiple execution layers and identifies which layer is the major factor affecting the overall performance. Their technique profiles the following layers:

- *Architecture*: performance monitoring and cache management components
- *Operating system*: virtual memory management component
- *Java virtual machine (JVM)*: dynamic compilation and dynamic memory management components
- *Application*: throughput performance

Our work adopts their proposed technique to understand the performance of Java application servers. While we do not capture any information from the architecture level, we utilize information from the operating system, Java virtual machine, and application layers generated by our experimental platform to analyze the overall performance.

3. Motivation

A study by Welsh et al. [4] reports three important trends that magnify the challenges facing Web-based applications. First, services are becoming more complex with widespread adoption of dynamic contents in place of static contents. Second, the service logics "tend to change rapidly". Thus, the complexity of development and deployment increases. Third, these services are deployed on general-purpose systems and thus are not "carefully engineered systems for a particular service" [4]. Such trends are now a common practice. Complex services including entire suites of business applications are now deployed using Web application servers running commodity processors and open-source software. With this in mind, we conducted an experiment to observe the degradation behavior of Java application servers on an experimental platform similar to the current common practice (i.e. using Linux on X86 system with MySQL database and JBoss application server). For detailed information on the experimental setup, refer to Section 4.2.

Characteristic	jvm98	jAppServer2004 (Transaction Rate = 100)
# of Threads	2	2600+
# of Objects	17 million	209 million
Allocated Space	500 MB	15 GB
Execution Time	seconds	hours

Fig. 1. Comparing basic characteristic of jvm98 to that of jAppServer2004.

3.1. Benchmarks comparison

Over the past ten years, numerous studies have been conducted to evaluate the performance of Java runtime environment. The evaluations are done at key runtime systems including interpretation [20], dynamic compilation [21–23], memory management [23–27,15,28–30], and synchronization [23,8,31]. However, such research efforts are often conducted using benchmark programs that are not representative of server workloads. To illustrate this claim, we compare the basic characteristics of *jvm98*, a commonly used benchmark suite from SPEC [32], with *jAppServer2004*, a standardized application server benchmark from SPEC [33], in Fig. 1.

In most instances, the jvm98 benchmark programs are not multi-threaded (*mtrt* is the only multi-threaded application). They create about 17 million objects at most, and require as little as a few megabytes of heap space to operate. These applications also complete their execution in tens of seconds. On the other hand, jAppServer2004 utilizes hundreds to thousands of threads. Multiple gigabytes of heap space are often needed. The benchmark also takes as long as a few hours to complete, which loosely emulates the behavior of long-running servers. With such drastic differences, it is quite possible that the lessons learned over the past decades may not be fully applicable to server environments.

3.2. Experimental methodology

Initially, our experiments were conducted using the smallest amount of workload allowed by jAppServer2004. We set the maximum heap size to be twice as large as the physical memory—4 GB heap with 2 GB of physical memory in this case. We chose this setting to emulate application servers facing unexpected heavy demands, which can drive up the memory requirement. Also note that our adopted Java virtual machine, HotSpot from Sun Microsystems [34], only commits a small amount of memory at the beginning and gradually commits more memory as the demand increases. We monitored the throughput delivered by the system. We then gradually increased the workload until the system refused to service any requests.

For comparison, we also conducted another experiment to observe the degradation behavior of the Apache Web server (we used the same computer system and *web2005*, a benchmark from SPEC [35] to create requests). Since the two benchmarks report different throughput metrics (jobs per second for jAppServer2004 and connections per second for web2005), we normalized the throughput and the workload to percentage. That is, we considered the maximum throughput delivered by a system during an execution as 100% (referred to as t) and the maximum workload when a system completely refuses requests as 100% (referred to as w). The degradation rate (referred to as d) is $d = \frac{\Delta t}{\Delta w}$.

3.3. Results and analysis

According to Fig. 2, the result shows that JBoss is able to deliver good throughput performance for about 60% of the given workload. However, when the workload surpasses 60%, the throughput reduces drastically. This system begins to refuse connection at 80% of the maximum workload. A drastic degradation in throughput (nearly 75%) occurs when the workload increases by only 20%. Thus, the degradation rate, d, is $\frac{0.75}{0.20} = 3.40$. Also notice that the value of d for the Apache is 1.69 (see Fig. 2). A smaller value of d means that the application is more failure-resistant to increasing workload. We also investigated the effect of larger memory on throughput performance. Again, larger memory improves the maximum throughput (see Fig. 3) but has very little effect on degradation behavior.

Analysis. According to Hibino et al. [8], the degradation behavior exhibited by the application server is considered ungraceful because such a behavior can lead to nonlinear responses and unpredictable systems. Moreover, it gives very little time to administer recovery procedures. Hibino et al. investigate the factors that affect the throughput degradation behavior of Java Servlets by examining the operating system behaviors. They find that thread synchronization at the

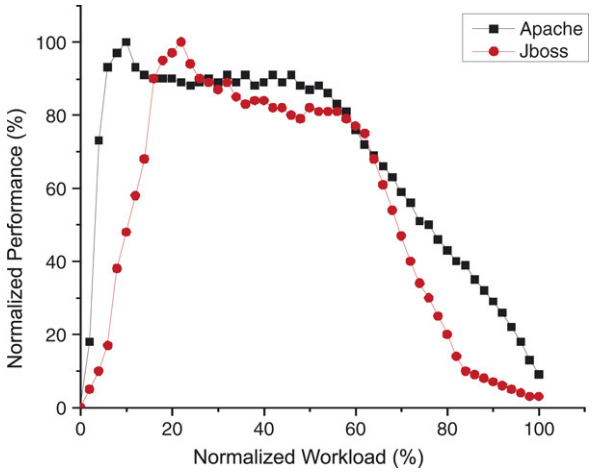

Fig. 2. Throughput degradation behaviors of JBoss and Apache.

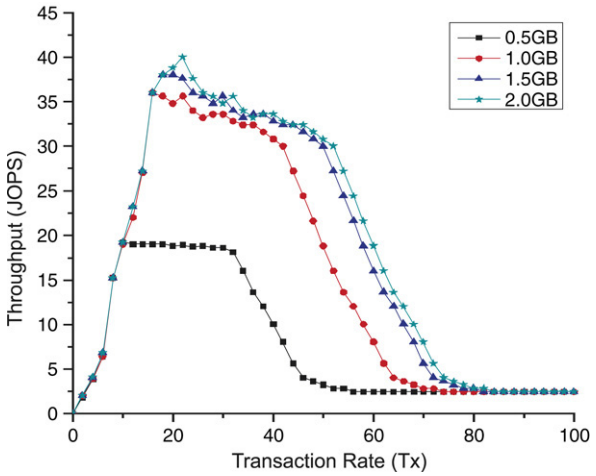

Fig. 3. Throughput comparison with respect to heap sizes.

OS level is the most prominent factor causing poor degradation. We wish to point out that their work does not examine any factors within Java virtual machines. On the other hand, our investigation is focused specifically at the Java Virtual Machine level. Since Java Virtual Machines (*JVMs*) provide execution environments for these application servers, we hypothesized that the major factors causing throughput performance to degrade ungracefully reside in the Virtual Machines.

4. Experiments

In this study, our main objectives are as follows:

Research Objective 1 (RO1): Identify the major factors responsible for the rapidly declining throughput of Java application servers due to small workload increase.

Research Objective 2 (RO2): Investigate how these factors affect the throughput of Java application servers.

Research Objective 3 (RO3): Observe how changes in algorithms and policies controlling these factors affect the throughput of Java application servers.

4.1. Benchmarks

There are two major components: application servers and workload drivers. The selected application servers must meet the following criteria. First, they must be representative of real-world/widely-used application servers. Second,

we must have accessibility to the source code to control and manipulate their execution context. Our effort began with the identification of server applications that fit the two criteria. We investigated several possibilities and selected the two open-source application servers described below.

JBoss [36] is by far the most popular open-source Java application server (34% of market share and over fifteen million downloads to date). It fully supports J2EE 1.4 with advanced optimization including object cache to reduce the overhead of object creation.

Java Open Application Server (JOnAS) [37] is another open-source application server. It is built as part of the ObjectWeb initiative. Its collaborators include the France Telecom, INRIA, and Bull (a software development company).

For the workload driver, we chose jAppServer2004 [33], a standardized benchmark from SPEC for testing the performance of Java application servers. It emulates an automobile manufacturer and its associated dealerships. Dealers interact with the system using web browsers (simulated by a driver program) while the actual manufacturing process is accomplished via RMI (also driven by the driver). This workload stresses the ability of Web and EJB containers to handle the complexities of memory management, connection pooling, passivation/activation, caching, etc.

The workload can be configured by varying the transaction rate, which specifies the number of Dealer and Manufacturing threads. Throughput of the benchmark is measured by JOPS (job operations per second). The jAppServer2004's design document [33] includes a complete description of the workload and the application environment in which it is executed.

Note that in this paper, we chose not to include other industrial standard server-side benchmarks such as SPECjbb2000 [38] or a more recent release SPECjbb2005 [39]. This is because these benchmarks do not provide realistic server environments. For example, both versions of SPECjbb simply simulate application servers, and all the database requests are not processed by real database engines. Additionally, they do not simulate network I/O and request time-out mechanism, which are common in real-word Java application servers.

4.2. Experimental platforms

To deploy jAppServer2004, we used four machines to construct two three-tier architectures. Since our experiments utilized both the Uniprocessor system and the Multi-processor system, our configuration can be described as follows.

Uniprocessor application server (System A): The client machine is a dual-processor Apple PowerMac with 2x2GHz PowerPC G5 processors and 2 GB of memory. The server is a single-processor 1.6 GHz Athlon with 1 GB of memory. The MySQL[2] database server is a Sun Blade with dual 2GHz AMD Opteron processors as the client machine running Fedora Core 2 and 2 GB of memory.

Multi-processor application server (System B): The client machine is the same as the system above. However, we swapped the application server machine and the database server machine. Thus, the dual-processor Sun Blade is used as the application server, and the single-processor Athlon is used as the database server.

In all experiments, we used Sun HotSpot virtual machine shipped as part of the J2SE 1.5.0 on the server side. The young generation area is set to $1/9$ of the entire heap, which has been shown to minimize the number of the expensive mature collections. We ran all experiments in stand-alone mode with all non-essential daemons and services shut down.

The virtual machine was instrumented to generate trace information pertaining to the runtime behavior, e.g. object allocation information, reference assignments, execution thread information, on-stack references, and garbage collection (GC) information. It is not uncommon for these traces to be as large as several gigabytes. These trace files were then used as inputs to our analysis tool that performs lifetime analysis similar to the Merlin algorithm proposed by Hertz et al. [18]. The major difference between our approach and theirs is that we employed off-line analysis instead of on-line analysis.

4.3. Variables and measures

We utilized several workload configurations to vary the level of stress on the application server. In all experiments, we increased the workload from the minimum value available to the maximum value that still allows the application

[2] MySQL available from http://www.mysql.com.

to operate. For example, we began our experiment by setting the transaction rate to 1. In each subsequent experiment, we increased the transaction rate until JBoss encounters failure. The failure point is considered to be the maximum workload that the system (combination of application server, Java virtual machine, OS, etc.) can handle. As shown in Section 3, throughput dramatically degrades as the workload increases. This degradation is likely caused by the runtime overhead. To address our RO1, we monitored the overall execution time (T), which is defined as:

$$T = T_{\text{app}} + T_{\text{gc}} + T_{\text{jit}} + T_{\text{sync}}.$$

It is worth noticing that T_{app} is the time spent on executing the application itself. T_{gc} is the time spent on garbage collection. T_{jit} is the time spent on runtime compilation. Many modern virtual machines use Just-In-Time (JIT) compilers to translate byte-code into native instructions when a method is first executed. This time does not include the execution of compiled methods; instead, it is the time spent on the actual methods compilation and code cache management. Finally, T_{sync} is the time spent on synchronization. We monitored synchronization operations such as lock/unlock, notify/wait, the number of threads yield due to lock contentions. We chose these time components because they have historically been used to measure the performance of Java Virtual Machines [20].

By measuring the execution time of each runtime function, we can identify the function that is most sensitive to the increasing workload. The result of this research objective is used as the focal point in RO2. To address RO2, we further investigated the runtime behaviors of these factors. Once again, we varied the workload but this time, we also measured other performance parameters such as the number of page faults in addition to throughput performance. These parameters gave us deeper insight into the effects of these factors on the throughput performance. To address RO3, we conducted experiments that adjust both the fundamental algorithms and the policies used by the runtime factors and observed their effects on the throughput performance. By making these changes, we expected to identify alternative algorithms and policies more suitable for Java application servers.

4.4. Hypotheses

We hypothesized that increasing workload can affect two major runtime components of a JVM: threading and garbage collection. Our hypothesis was based on two observations. First, increasing workload is due to more simultaneous clients. This can, in turn, result in larger synchronization overhead, which affects performance. Second, a larger number of clients also result in more object creations. Therefore, the heap is filled up quicker, and garbage collection is called more frequently.

We conducted experiments to investigate the validity of these conjectures based on the following hypotheses.

H1: Thread synchronization and garbage collection are the two runtime functions most sensitive to workload. Our second research question attempts to identify the causes that affect the performance of the identified runtime functions, and in turn, affects the throughput of the applications. We conjectured that runtime algorithms (e.g. generational garbage collection) and policies (e.g. when to call garbage collection) can greatly affect the performance of runtime functions. Therefore, our experiments are designed to also validate the following hypothesis.

H2: Runtime algorithms and management policies can affect the performance of runtime functions and overall throughput. Therefore, changes in the algorithms and/or policies can affect throughput degradation behavior. We conducted experiments to validate this hypothesis and reported the results.

5. Results

5.1. RO1: Factors that affect throughput

We conducted experiments to identify factors that can affect the throughput performance of Java application servers. We measured the execution time of each major runtime function in the virtual machine when the system is facing (i) the lightest workload and (ii) the heaviest workload. Fig. 4 reports the accumulated execution time (T). Notice that when the workload is light, only a small portion of time is spent on common VM functions. That is, T_{gc}, T_{sync}, and T_{jit} only account for 5%, 2% and 5% of the execution time, respectively. The remaining 88% is spent on application execution (T_{app}). Within this period, the maximum throughput is also achieved. Also notice that T_{jit} is very small and does not increase with workload. Because most commercial virtual machines do not discard compiled methods, they can be reused throughout the program execution [40,22].

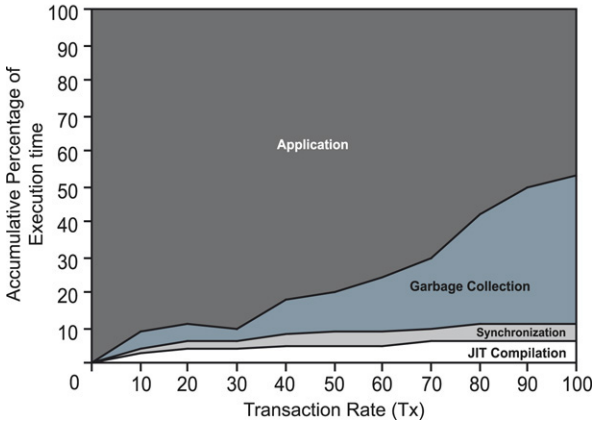

Fig. 4. Accumulative time spent on major runtime functions.

Workload	Minor GC		Full GC	
	# of invocations	Avg. Pause (min:max) (seconds)	# of invocations	Avg. Pause (min:max) (seconds)
10	2037	0.021 (0.015:0.026)	48	0.78 (0.412:1.340)
20	2219	0.020 (0.011:0.033)	72	1.02 (0.232:2.021)
30	2901	0.022 (0.014:0.031)	115	1.13 (0.512:2.372)
40	3213	0.024 (0.011:0.039)	140	1.20 (0.412:3.721)
50	3907	0.021 (0.015:0.029)	192	1.45 (0.670:5.142)
60	4506	0.023 (0.012:0.026)	250	2.91 (1.010:7.020)
70	5102	0.027 (0.014:0.036)	370	3.31 (1.012:12.012)
80	5678	0.023 (0.015:0.037)	422	4.98 (2.102:34.014)
90	6150	0.025 (0.013:0.039)	512	6.12 (2.456:100.040)
100	7008	0.028 (0.015:0.039)	709	10.10 (3.124:300.024)

Fig. 5. Garbage collection activities and pause times.

Synchronization as a factor. A study by Hibino et al. [8] reports that thread synchronization is the major factor that causes the throughput of the dynamic content generation tier to degrade differently among various operating systems; in most cases, the throughput performance degrades ungracefully. Because their observation is made at the operating system level, it is not surprising for them to draw such a conclusion. Most runtime functions in JVMs may not utilize system calls. For example, a memory allocator invokes system calls only when it needs to enlarge the heap. Thus, their methodology would regard runtime functions in the VM as application execution. We expect that deeper insight can be gained by observing the application, virtual machine, and operating system performances in a similar fashion to the *vertical profiling* technique introduced by Hauswirth et al. [19].

As stated earlier, we monitored the accumulated execution time of all major runtime functions in HotSpot. Our result is reported in Fig. 4. Notice that the experimental result confirms our hypothesis that synchronization is workload-sensitive, as the time spent in synchronization becomes larger with higher workload due to more resource contentions. However, the increase is only a small percentage and should not affect the throughput degradation behavior.

Garbage collection as a factor. Fig. 4 shows that the time spent in garbage collection increases dramatically with heavier workload. Just prior to the complete failure of JBoss, the accumulative garbage collection time is more than 50% of the overall execution time. We also found that garbage collection pauses can be as much as 300 s during the heaviest workload (see Fig. 5). As more time is spent on garbage collection, less time is spent on executing the application; thus, the throughput performance degrades drastically. As a result, we conclude that garbage collection is a major factor that can affect the throughput and degradation behavior of the Java application server.

5.2. *RO2: Effects of GC on throughput performance*

Currently, many commercial virtual machines including Sun J2SE 1.5 and Microsoft .NET CLR rely on generational garbage collection as the algorithm of choice for object management in server systems. Thus, our first

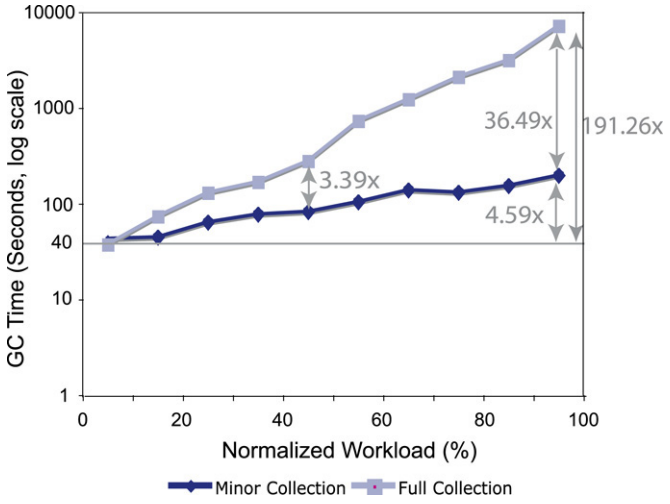

Fig. 6. Time spent on minor GC and full GC.

focus is on the effects of the generational algorithm on throughput performance. For more information on generational garbage collection, refer to Section 2.

Since the heap size can also affect garbage collection performance (i.e. a bigger heap translates to more time for objects to die), heap resizing policy can also play an important role. Thus, it is the second focus of our study. In addition, we also study other factors such as programmer's intervention and garbage collection triggering policy that can affect the performance and efficiency of garbage collection.

5.2.1. Effects of generational garbage collection

Generational collectors are designed to work well when the majority of objects die young. As reported earlier, the generational collector used in the JDK 1.5 performs extremely well when the workload is light. However, its performance degrades significantly as the workload becomes much heavier. To understand the major causes of such a drastic degradation, we investigated the garbage collection frequency when the workload is heavy.

Notice that more time is spent on full collection as the workload is getting heavier (see Fig. 6). At the heaviest workload, the system spent over 7000 s on full collection (about 36 times longer than that of minor collection and over 190 times longer than the time spent on full GC at the lightest workload). Pause time can also be as long as 300 s. We further investigated this behavior and found that frequent garbage collection prevents the application from making any significant progress. In effect, the garbage collector simply thrashes. Thus, the mark-sweep collector used to perform full collection touches objects again and again, resulting in high garbage collection processing cost.

Investigating lifespan. One possible reason for frequent mature collection invocations is that many objects survive minor collection, and thus, quickly occupy space in the mature collection. To investigate whether such phenomenon exists in jAppServer2004, we conducted a set of experiments to compare the lifespans of objects in desktop applications and server applications.

We measured lifespan by the amount of memory allocated between birth and death (in bytes). (We only accounted for objects allocated in the garbage-collected heap and ignored any objects created in the permanent space.) We measured the execution progress by the accumulated amount of allocated memory (also in bytes) [27]. We chose allocated memory instead of time because it is completely independent of other execution factors such as paging overhead. Fig. 7 depicts our findings. The x-axis represents the normalized lifespans, and the y-axis represents the normalized death times with respect to the normalized total allocated memory.

The vast majority of objects in *jess*, a benchmark in jvm98, are short-lived; that is, most objects have lifespans of less than 10% of the maximum lifespan. Moreover, the number of dead objects is uniformly distributed throughout the execution. Note that we also conducted similar studies using other applications in the jvm98 benchmark suite and found their results to be very similar to *Jess*. For brevity, we do not include the results of those studies. The results of our study nicely conform to the "weak generational hypothesis" (most objects die young), which is the cornerstone of generational garbage collection [41,13].

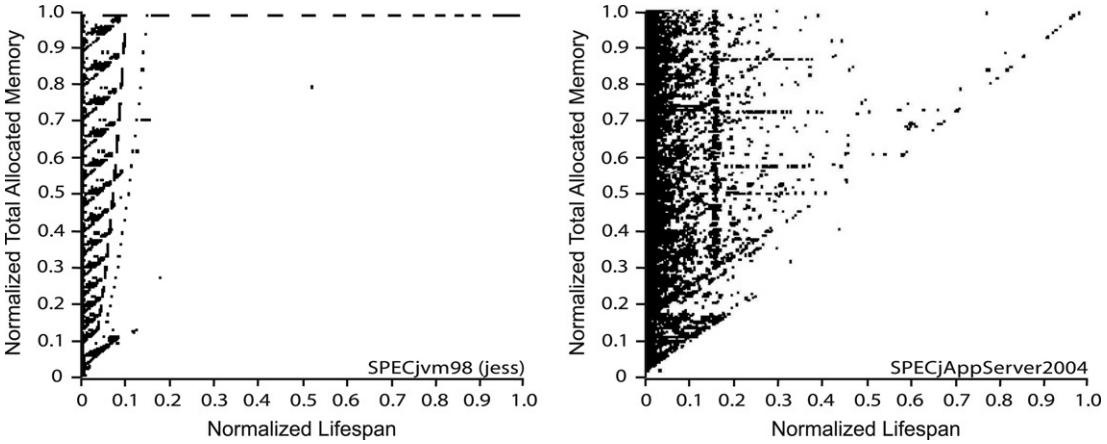

Fig. 7. Comparing lifespans of objects in *jess* with lifespans of objects in jAppServer2004.

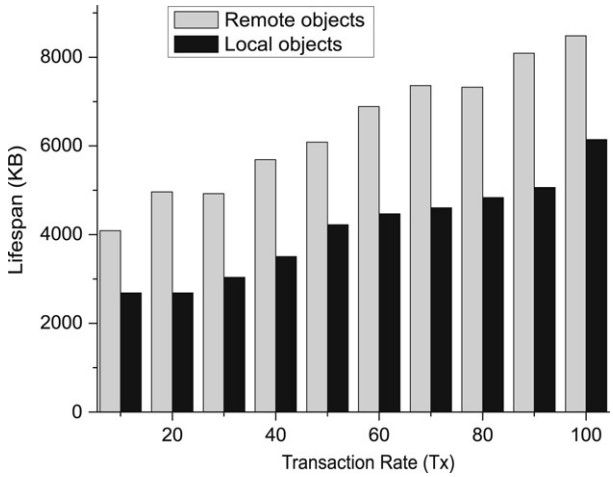

Fig. 8. Comparing lifespans of local and remote objects.

On the other hand, the lifespan characteristic of objects in jAppServer2004 is significantly different than that of objects in Jess. First, in addition to the short-lived objects, a large number of objects in the server application have lifespans of up to 30% to 50% of the maximum lifespan. Second, the number of dead objects is not distributed uniformly; that is, there are more dead objects as these applications move toward the end of execution. Thus, we conclude that objects in jAppServer2004 become longer living as the workload becomes heavier.

Remote versus local objects. A study by [42] has shown that there are commonly two types of objects in .NET server applications: *local* and *remote*. Remote objects are defined as objects created to serve remote requests, and local objects are created to serve local requests. Note that we define remote objects as remotable objects (a remotable object is the object, which implements interface *java.rmi.Remote*) or any objects directly or indirectly reachable from a remotable object. In Java Application Servers (J2EE, JBoss, etc.), all *Enterprise Java Beans* (EJBs) are remotable objects. They implement two interfaces: *EJBHome* and *EJBObject*, which extend the superinterface *java.rmi.Remote*. All objects other than remote objects are local objects. The study showed that objects of these two corresponding types have very distinctive lifespans; remote objects tend to live much longer. To investigate whether such observation applies to our experiments, we investigated the lifespans of remote objects and local objects when JBoss faces the heaviest workload.

Fig. 8 indicates that the average lifespan of remote objects is longer than that of local objects. Thus, a generational garbage collector is likely to spend additional time and resources to promote these long-lived remote objects.

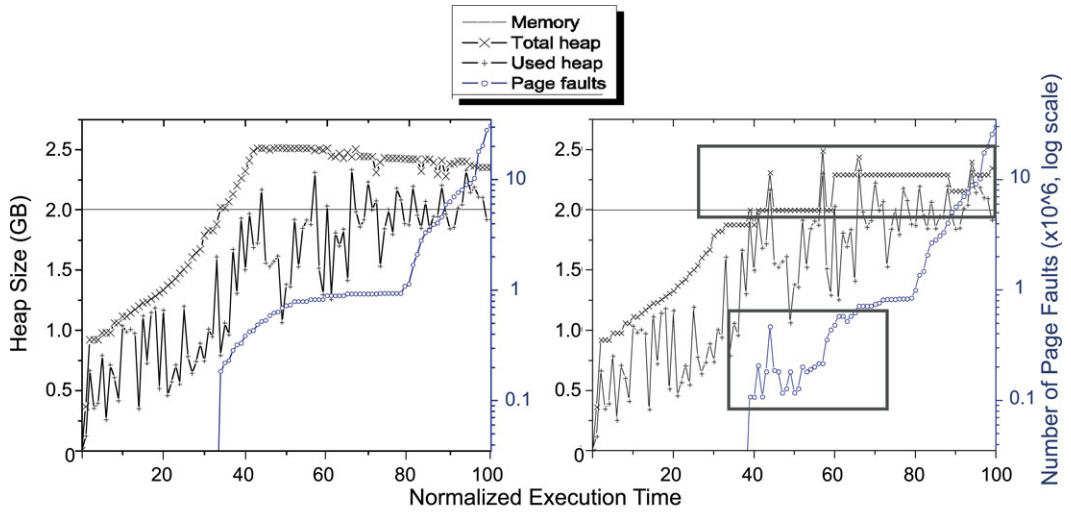

Fig. 9. Comparing memory and paging activities of Jboss with (left) and without (right) the adaptive sizing mechanism (transaction rate = 100).

5.2.2. *Effects of heap enlargement policy*

We also investigated heap enlargement policy adopted in HotSpot as a possible cause of poor garbage collection performance. Typically, there are two major considerations in performing heap resizing: *when to resize* and *by how much*.

In HotSpot, the pause time of each GC is evaluated by applying a linear curve fitting process to the most recent GC pause times. If the slope of the resultant linear function is positive (i.e. the pause times are increasing), HotSpot would expand the heap by 20%. If the slope is negative, HotSpot would reduce the heap by 4%. This approach has two significant drawbacks based on our experimental observation. First, the VM increases the heap size aggressively but reduces the heap size too conservatively. When the footprint in the heap is smaller than the memory size, but the total heap is larger than the memory size, it takes a long time to reduce the heap.

Second, the heap enlargement mechanism does not take into account the physical memory available and often lets the heap grow to be larger than the physical memory capacity very quickly. For example, Fig. 9(a) shows the heap sizing activity at the heaviest workload (transaction 100). Note that the physical memory size is 2GB. The solid line is the new heap size after each sizing point. The dotted line is the actual heap usage (i.e., the number of live objects) after each GC invocation. The star line is the number of page faults during the lifetime measured using the scale shown on the y-axis on the right of both figures. The figure shows that the heap is increased to be larger than the available physical memory at about 33% of the execution time. At this point, the actual heap usage is still smaller than the physical memory size. This enlargement induces a large number of page faults for the remainder of the execution. As stated earlier (see Fig. 5), the pause time can be as long as 300 s as a significant amount of page faults occur during a full collection invocation.

Summary. We found that generational garbage collection may not be suitable for application servers under stress. This is due to a large number of objects in applications servers tend to be longer living; thus, less objects are collected in each minor collection and more frequent full collection is invoked.

In addition, the current policy adopted by HotSpot also enlarges the heap very frequently to yield optimal garbage collection performance early on. In this strategy, the heap can become so large that the heap working set can no longer fit within the physical memory capacity. If this point is reached too soon, the system would spend a large amount of time servicing page faults. This is especially true during the mature collection, as mark-sweep has been known to yield poor paging locality [10,30]. Two possible solutions to address this issue are: (i) to use garbage collection techniques that are more suitable for long-lived objects, and (ii) to adaptively resize the heap based on the amount of physical memory and transaction time requirement. We will preliminarily evaluate these two options in the next subsection.

5.2.3. Intervention from programmers

Often times, programmers creates customize data structure in hope of improving the overall performance. Such an example exists in JBoss where programmers create a special region in the heap to cache objects to reduce the allocation efforts. However, such a cache can also lead to a larger number of long-lived objects that must be promoted during garbage collection, as these objects continue to be reachable from the cache. In this subsection, we investigate the effect of object cache on garbage collection performance.

Objects cache. JBoss uses a large cache pool to keep hot beans. As more beans are kept in the cache, fewer calls are made to the database [43] . Usually the optimal-cache size is set to be proportional to the workload. Keeping objects in the cache makes these objects long-lived, reducing the efficiency of generational garbage collection.

In our study, we found that the maximum number of beans in the JBoss cache pool can reach 1 million, which translates to at least 8 MB of heap space. We also discovered that the average tracing rate of the mark-sweep garbage collector is 59.2 MB/sec. This means that the garbage collector spends about 130 ms traversing objects in the cache pool. Since these objects are kept alive by the cache, they are traversed again and again. During the heaviest workload, it will take up as much as 10% of the overall garbage collection time. As a reminder, the GC time during this period can be as much as 50% of execution time. Therefore, the time spent by the collector in the cache pool is very noticeable.

5.2.4. Untimely garbage collection invocations

The collection triggering mechanism can also affect the throughput of application servers. Currently, the most adopted approach is space-based, i.e., the virtual machine invokes garbage collection when the heap space is full. Therefore, this technique may not invoke garbage collection at the point that yields the highest garbage collection efficiency. Poor efficiency results in longer pauses.

We first describe our definition of garbage collection efficiency. Note that the description can be applied to any stop-the-world garbage collectors. In long-running server applications, the path of execution consists of several GC cycles and mutation cycles. Each GC cycle corresponds to a period when GC is working. Each mutation cycle corresponds to a period when the application is running. Stop-the-world garbage collectors performed all GC work at the end of the mutation cycles when the heap memory is exhausted, in effect halting the application until the end of the GC cycle.

When a GC cycle begins, the collector traverses all reachable objects and reclaims unreachable objects. Here we define $\sum \text{live}(t)$ and $\sum \text{dead}(t)$ as the number of live objects and the number of dead objects at time t, respectively. The former reflects the cost of detecting and maintaining all reachable objects (e.g. copying, traversing, and marking), and the latter reflects the amount of garbage that can be reclaimed (e.g. sweeping). The best time to invoke garbage collection is when the number of dead objects is high while the number of live objects is low. Here we define the GC efficiency of time t as:

$$\text{Efficiency}(t) = \frac{\sum \text{dead}(t)}{\sum \text{dead}(t) + \sum \text{live}(t)}.$$

This formula also represents the garbage and live objects ratio at time t. It is proportional to $\sum \text{dead}(t)$ while inversely proportional to $\sum \text{live}(t)$. We then computed the exact object reachability and lifetime information. Such information can be used to calculate $\sum \text{dead}(t)$, $\sum \text{live}(t)$, and $\text{Efficiency}(t)$ at any time t.

Based on the calculation, we can obtain the exact invocation points that would yield the most efficient garbage collection. We then examine the accuracy of the space-based approach in invoking garbage collection at these efficient points. Our method divides the mutation cycles into three regions: *first-half* (within the first-half of a mutation cycle), *second-half* (within the second-half of a mutation cycle), and *at-the-end* (when the heap is full). Fig. 10 depicts our finding.

It is very interesting that with lighter workload, the efficient invocation points tend to be toward the end of mutation cycles. This means that the current space-based approach would have worked efficiently most of the time. However, as the workload becomes heavier, the most efficient points tend to fall in the second-half of the mutation cycles *but not at the end*. This implies that most garbage collection invocations are not triggered at the most optimal places. The inefficient triggering mechanism prolongs pause time and, in turn, affects the server performance and throughput.

5.3. RO3: Effects of changes in algorithms and policies on throughput

In this section, we report the results of our experiments to investigate the effect of changes in garbage collection algorithms and the governing policies adopted in HotSpot. We experimented with different garbage collection

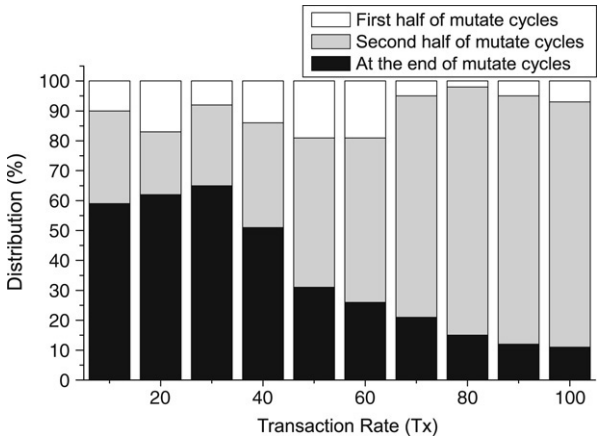

Fig. 10. Execution regions that contain optimal GC invocation points.

algorithms that also include parallel and concurrent techniques. Our experiments also varied parameters and policies that control the following mechanisms: heap sizing, object-cache sizing, and GC invocation.

5.3.1. Adaptive heap sizing mechanism

From the previous section, we discovered that the adopted policy in HotSpot increases the heap size so quickly that the heap exceeds the physical memory capacity very early in the execution. As a result, the system suffers a large number of page faults. We experimented with a new adaptive heap sizing policy that has been implemented into HotSpot. Our new policy attempted to maintain the lowest possible heap size especially when the physical memory resource is scarce. As stated earlier, there are two considerations to perform heap resizing: when to expand or reduce the heap and by how much.

Our approach does not change the decision of when to resize the heap. However, we changed the adjustment quantity. Based on our study of the current heap sizing policy in HotSpot, we noticed that page faults begin to occur when the heap is larger than 75% of the physical memory (e.g. 1500 MB heap in a system with 2 GB physical memory). Thus, we used this insight to set a threshold to adjust our sizing policy. When the current heap size is smaller than 75% of the physical memory, the heap is increased by α percent during an expansion. Once the heap size exceeds the 75% threshold, the percentage of enlargement is reduced to β ($\beta \leq \alpha$). We investigated the throughput and its degradation behavior under four different configurations of α and β: $\alpha = 20/\beta = 20$, $\alpha = 20/\beta = 10$, $\alpha = 10/\beta = 5$, and $\alpha = 5/\beta = 2.5$. Notice that $\alpha = 20$ and $\beta = 20$ represent the original policy. In the $\alpha = \beta = 20$ approach, the heap is always increased by 20% of the current size, no matter if its size exceeds the physical memory capacity or not. In our adaptive approach, the JVM adjusts the increasing percentage according to the available memory space. For example, in the $\alpha = 10/\beta = 5$ approach, the heap is enlarged by 10% prior to 1500 MB heap size; afterward, the heap is increased by only 5%.

Fig. 11 reports our finding. It is worth noticing that the changes in the heap sizing policy have only minor effects on the throughput degradation behavior. However; a more conservative enlargement policy can significantly degrade the throughput as shown with $\alpha = 5/\beta = 2.5$ configuration. Also notice that the current policy used by HotSpot ($\alpha = 20/\beta = 20$) does not yield the best throughput performance; instead, $\alpha = 10/\beta = 5$ yields the best throughput throughout the execution. Even though the proposed adaptive heap sizing policy has very little effect on the throughput degradation behavior, it can yield two additional benefits: lower heap usage and smaller number of page faults.

Reduction in heap usage. Fig. 12 compares the amount of heap space needed by the application server with the actual heap size allocated by the JVM using two policies: $\alpha = 20/\beta = 20$ and $\alpha = 10/\beta = 5$. As a reminder, $\alpha = 20/\beta = 20$ is the approach currently used in the HotSpot VM and $\alpha = 10/\beta = 5$ has shown to yield higher throughput. Notice that the proposed adaptive heap sizing policy utilizes the heap space more efficiently by committing the heap memory only slightly higher than the actual heap usage (125 MB). On the other hand, the approach currently used by HotSpot committed a much larger amount of additional memory (about 500 MB) once the memory usage exceeds the physical memory.

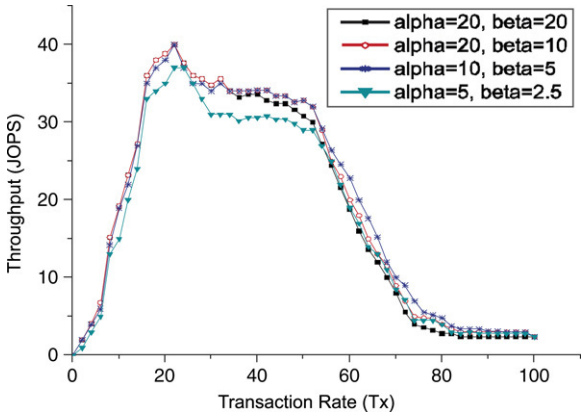

Fig. 11. Throughput performance after applying the adaptive sizing mechanism.

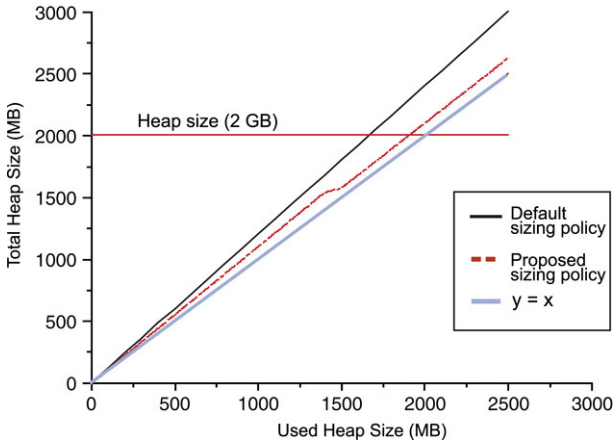

Fig. 12. Heap usage with the adaptive sizing mechanism.

Reduction in page faults. We compared the number of page faults between the two policies: $\alpha = 20/\beta = 20$ and $\alpha = 10/\beta = 5$. Fig. 9(b) shows that our decision to slow the growth percentage at the beginning ($\alpha = 10$ instead of $\alpha = 20$) results in a reduction in the number of page faults early on (highlighted by the lower rectangular box). The reduction is about 10%. However, the proposed adaptive sizing policy has very little effect on the number of page faults once the threshold is reached. Based on our results, we conclude that:

- Moderately conservative heap sizing policy has only a slight effect on maximum throughput. This is illustrated when we can achieve the best throughput with $\alpha = 10/\beta = 5$ approach.
- Moderately conservative heap sizing policy can significantly reduce the number of page faults. However, the technique is more effective before the threshold is reached.
- Conservative heap sizing policy can reduce the amount of memory usage (highlighted by the top rectangular box in Fig. 9(b)) and slightly improves the throughput throughout the execution. However, it has very little effect on the throughput degradation behavior.

5.3.2. Improving garbage collection parallelism

Starting in J2SE 1.4.x, Sun also provides two additional GC techniques, parallel garbage collection (ParGC) and concurrent garbage collection (CMS), in addition to the default generational mark-sweep [7]. The parallel collector is similar to the generational mark-sweep approach except that it utilizes parallel threads to perform minor and major collection. Thus, it is a stop-the-world approach designed to minimize pause time.

In Concurrent Mark-Sweep (CMS), a separate garbage collector thread performs parts of the major collection concurrently with the applications threads. For each major collection invocation, the concurrent collector pauses all the application threads for a brief period at the beginning of the collection and toward the middle of the collection.

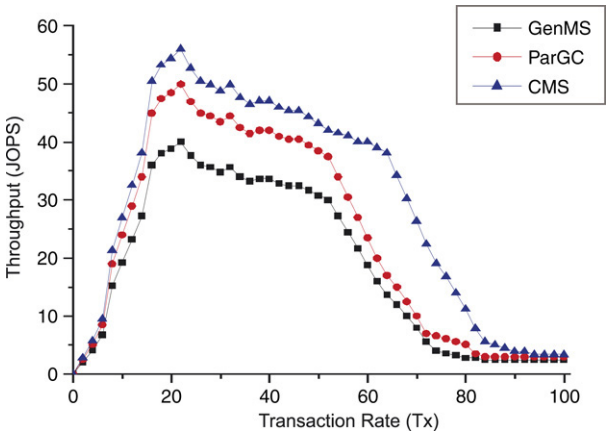

Fig. 13. Effects of CMS and ParGC on throughput performance.

The remainder of the collection is done concurrently with the application. This collector performs parallel scavenging in the minor collections and concurrent mark-and-sweep in the major collections.

According to Sun, the concurrent collector is ideal for server applications running on multi-processor systems (it cannot be used in single-processor systems). A study by Sun has shown that the concurrent collector can deliver higher throughput performance than the other approaches [34]. However, the effect of the concurrent garbage collector on the throughput degradation behavior is not known. Therefore, the goal of this experiment is to investigate the effect of CMS on throughput degradation behavior. Note that we used *system B*, the multi-processor system for this experiment.

Fig. 13 indicates that CMS can greatly improve the maximum throughput of the system. The differences in throughput performances between the concurrent collector and the single threaded generational mark-sweep (GenMS) can be as high as 40%. However, comparing the degradation rates of the three GC techniques, d_{CMS}, d_{ParGC}, and d_{genMS}, shows that both CMS and ParGC have very little effects on the degradation rates. Based on this finding, we concluded that *the concurrent collector running on a more powerful computer system improves the maximum throughput due to better parallelism, but does not make the throughput degrade more graceful*.

5.3.3. Different garbage collection algorithms

We conducted our experiments on the Jikes RVM due to its flexibility in choosing different garbage collection algorithms. Since JBoss is not supported on the RVM, we also used a different application server. JOnAS is another open-source application server that is supported by the RVM. Once again, we used jAppServer2004 as the workload driver.

To make certain that our substitution still provides a sound experimental platform, we conducted an experiment to compare the throughput degradation behaviors of the two systems, $\text{system}_{\text{HotSpot}}$ (jAppServer running on JBoss and J2SE 1.5) and $\text{system}_{\text{RVM}}$ (jAppServer running on JOnAS and RVM using generational collection (GenMS)). If the two systems show similar throughput patterns (based on normalized information), we assumed that any improvement resulting from modifications of $\text{system}_{\text{RVM}}$ would also translate to similar improvements in $\text{system}_{\text{HotSpot}}$ if the same modifications were also applied. Fig. 14 depicts the results of our comparison. Notice that the patterns are nearly identical.

Next, we conducted a set of experiments using different garbage collection techniques. The goal of these experiments is to compare the differences in the throughput behavior of each technique from the reference configuration ($\text{system}_{\text{RVM}}$). The description of each technique is given below.

GenMS: This hybrid generational collector uses a copying nursery and the MarkSweep policy for the mature generation. It is very similar to the generational mark-and-sweep collector in HotSpot. Thus, it is used as the reference configuration.

SemiSpace: The semispace algorithm uses two equal sized copy spaces. It contiguously allocates into one, and reserves the other space for copying into since in the worst case all objects could survive. When full, it traces and copies live objects into the other space, and then swaps them.

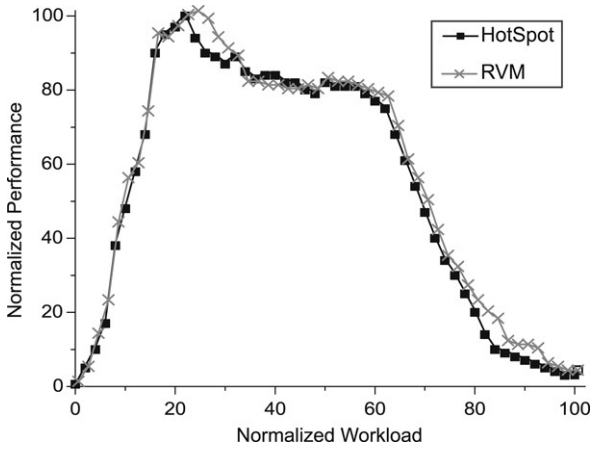

Fig. 14. Comparing throughputs of $\text{system}_{\text{RVM}}$ to $\text{system}_{\text{HotSpot}}$.

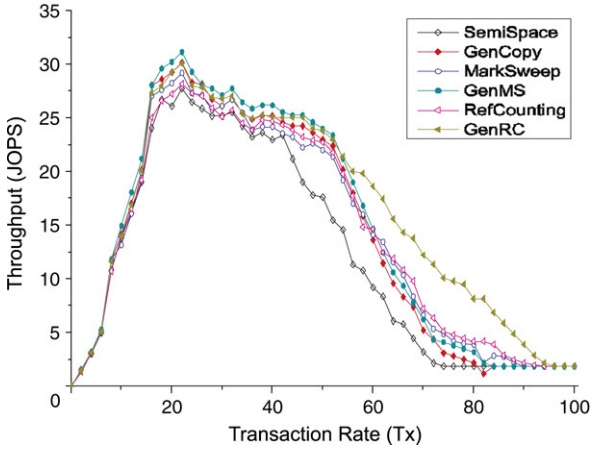

Fig. 15. Comparing the throughputs of different GC techniques.

GenCopy: The classic copying generational collector [44] allocates into a young (nursery) space. The write barrier records pointers from mature to nursery objects. It collects when the nursery is full, and promotes survivors into a mature semispace. When the mature space is exhausted, it collects the entire heap.

MarkSweep: It is a tracing and nongenerational collector. When the heap is full, it triggers a collection. The collection traces and marks the live objects using bit maps, and lazily finds free slots during allocation. Tracing is thus proportional to the number of live objects, and reclamation is incremental and proportional to allocation.

RefCount: The deferred reference-counting collector uses a freelist allocator. During mutation, the write barrier ignores stores to roots and logs mutated objects. It periodically updates reference counts for root referents and generates reference count increments and decrements using the logged objects. It then deletes objects with a zero reference count and recursively applies decrements. It uses trial deletion to detect cycles [16,45].

GenRC: This hybrid generational collector uses a copying nursery and RefCount for mature generation [15]. It ignores mutations to nursery objects by marking them as logged, and logs the addresses of all mutated mature objects. When the nursery fills, it promotes nursery survivors into the reference counting space. As part of the promotion of nursery objects, it generates reference counts for them and their referents. At the end of the nursery collection, GenRC computes reference counts and deletes dead objects, as in RefCount.

Fig. 15 reports our finding. It is worth noticing that most techniques yield very similar throughput degradation behaviors. The two exceptions are SemiSpace and GenRC. For SemiSpace, the collection time is proportional to the number of live objects in the heap. Its throughput suffers because it reserves one-half of heap space for copying. It also

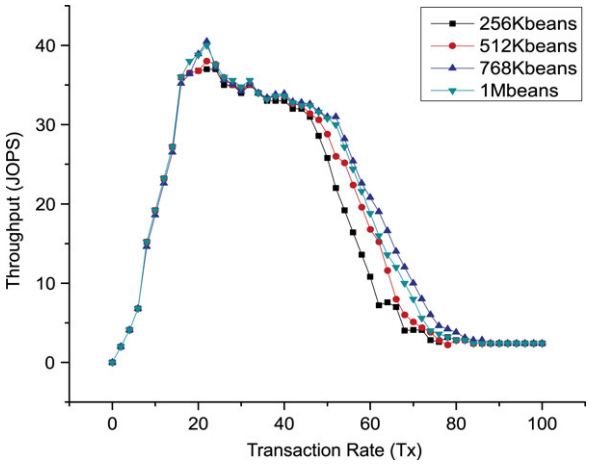

Fig. 16. Effects of varying sizes of object cache.

repeatedly copies objects that survive for a long time. Therefore, it has the lowest throughput at all workload levels compared to the other 5 collectors.

GenRC, on the other hand, allows the throughput of the application server to degrade much more gracefully. Unlike GenMS in which mature collection is frequently invoked during the heaviest workload, GenRC allows mature collection to be performed incrementally; thus, long pauses are eliminated and the memory space is recycled more efficiently. In addition, GenRC also ignores mutations of the young objects; thus, the bookkeeping overhead due to reference manipulations is avoided.

5.3.4. Object-cache management

As stated earlier, the more beans in cache, the longer it will take to scan the cache pool during garbage collection. Therefore, varying the cache size may affect the throughput performance and the degradation behavior of application servers. In our investigation, we varied the JBoss cache pool size and observed the effects of the cache size on the throughput performance. Fig. 16 represents performances of four different cache sizes: 256K, 512K, 768K and 1M beans.

The figure shows that the 256K cache size has the worst peak throughput and the poorest degradation behavior. This is because the cache size is too small, and results in much more frequent objects creation and possible communication overhead between the application server and the database server. In contrast, the 768K cache size performs the best and shows nearly the same peak throughput as the 1 MB cache size. It also performs best under heavy workload. This indicates that a cache pool can benefit server performance if its size is set appropriately. However, if this value is set too large, GC overhead incurred by cache pool may outweigh its benefit.

5.3.5. Optimal garbage collection triggering locations

To investigate the benefits of invoking garbage collection at the most optimal locations, we created a mark-sweep collection simulator that takes allocation information and optimal triggering locations as the input. Our simulator can utilize the space-based criterion as well as the optimal locations to trigger garbage collection. Since our investigation was simulation-based, we could not measure the execution time or throughput, but we could measure the efficiency of each garbage collection invocation.

We used 100-transaction rate in jAppServer2004 because it is the rate that causes the system to refuse connection. Fig. 17 depicts our finding. The x-axis represents GC invocation points during the benchmark execution. The y-axis represents the GC efficiency. The solid line represents the actual GC efficiency values of the space-based approach after each GC invocation. The dotted line represents the simulated optimal GC efficiency values between two adjacent GC points.

The space-based approach performs very well when the workload is light. However, after the first 13% of all garbage collection invocations, the optimal approach yields consistently higher efficiency (as much as 30%). While the result showed great promise, it is nontrivial to identify the optimal garbage collection points. Moreover, it is

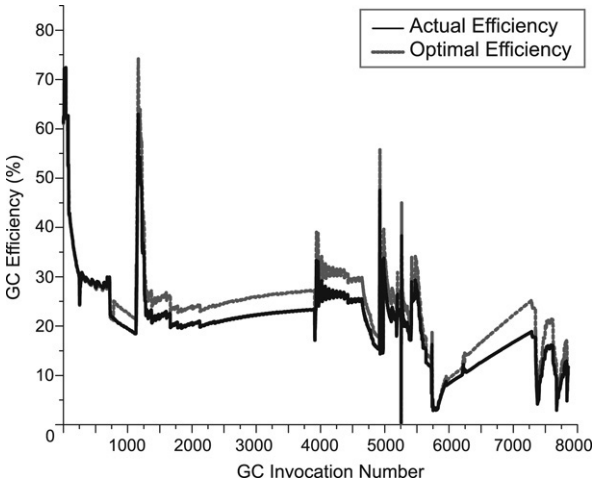

Fig. 17. Identifying efficient garbage collection triggering points.

unclear how the improved GC efficiency affects the throughput degradation. We will leave more experimentation and analysis of this approach for future work.

6. Future work

Our experiments show that inappropriate triggering mechanism can affect server performance. However, identifying the most efficient invocation points dynamically is not trivial. We have developed a predictive model called Fortune Teller [46] to estimate the amount of dead objects in the heap. Our preliminary experiments already showed that it is sufficiently accurate on several jvm98 benchmark applications. The model needs to be validated on more applications, particularly on server applications. By using this model, we can use its information to invoke garbage collection at instances yielding highest efficiency so that the unused memory is recycled more efficiently.

It is also possible that there is a correlation between allocation behavior and lifespan. We have conducted preliminary experiments and found that in many applications, there are allocation pauses that coincide with the highest mortality rates (see Fig. 18). The upper graph shows the volume of live objects in the heap and the total allocated bytes of an application server thread throughout its lifetime. The bottom graph shows an enlarged portion of the upper graph, which clearly shows that a large number of objects die after each pause. This insight may allow us to predict the best time to invoke garbage collection. We are currently working on a phase-based triggering garbage collector.

In Java application servers, objects can be classified into local objects and remote objects, depending on the type of services for which they were created. We have demonstrated that remote objects tend to be long-lived. We are currently working on a Service-Oriented garbage collection that segregates objects based on service types. The simulation results have shown that the scheme can significantly reduce the number of mature collection invocations [47].

We also investigated the garbage collection overhead by looking at several GC components such as heap size, collection algorithm, triggering mechanism and sizing policy. We did not consider other non-GC components in the VM that may influence GC and throughput performance. One possible component is thread scheduling. In a multi-threaded server environment, a large number of threads are created and operate simultaneously. We plan to experiment with thread scheduling algorithms that are GC cognizant. For example, threads that are expected to relinquish a large amount of memory may be given a higher priority so that memory is timely reclaimed. We expect that such algorithms can further improve throughput and affect the degradation behavior.

7. Related work

A study by Blackburn et al. [48] compares the cost of different GC techniques using different heap sizes and architectures. There are also several research efforts that recognize the effect of garbage collection on throughput performance. Ulterior reference counting [15] attempts to improve the overall throughput of Java applications by using reference counting for the mature generation space. These efforts do not study the influence of GC techniques on throughput degradation of application servers.

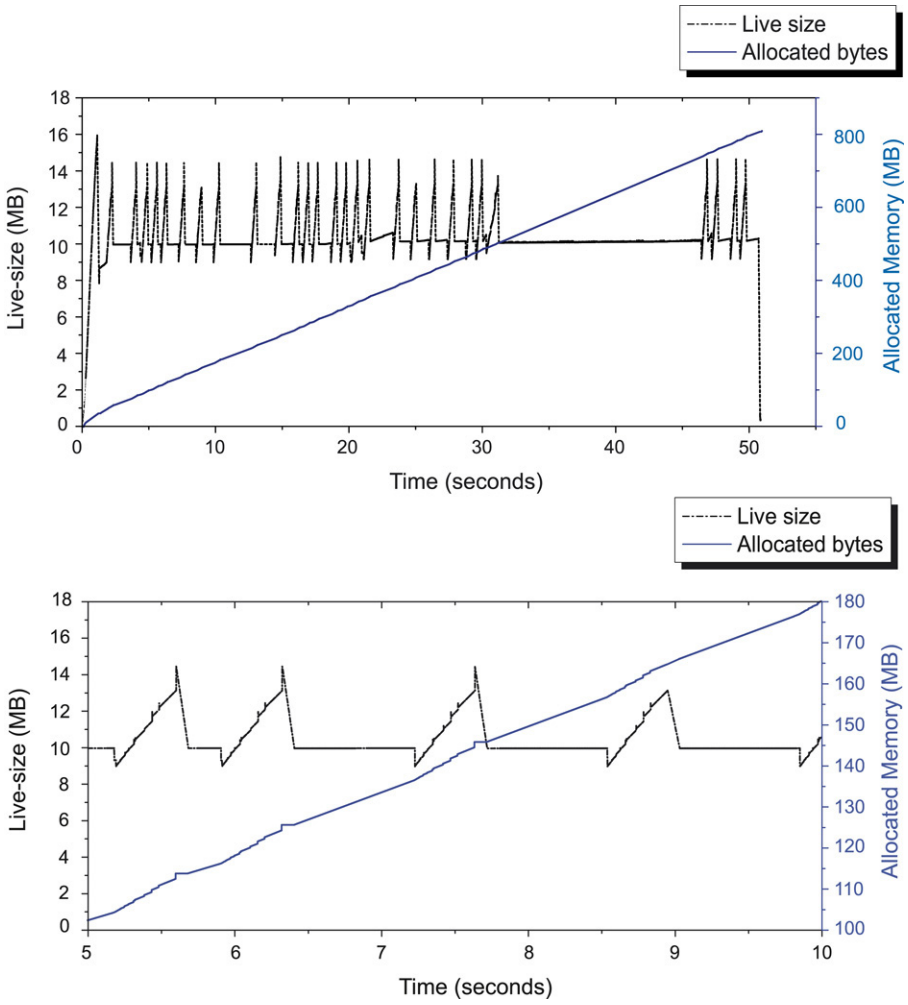

Fig. 18. Correlation between allocation pauses and the volume of lived objects.

Recent studies have shown that once the heap size is larger than the physical memory, paging overhead dominates the execution time, and may even result in thrashing [27,49,29,30]. Recent efforts have concentrated on dynamic sizing of the heap to maximize the performance of the existing GC techniques while minimizing paging [50,29,30,49,51]. While these solutions have shown to work well, they all accept the fact that generational GC is memory inefficient, and therefore assume there is enough physical memory for the needed headroom. In large server applications, this assumption does not always hold. Workload variation can reduce the amount of available headroom and cause the heap size to be larger than the available physical memory.

Currently, there are two general techniques to improve the efficiency of generational GC. The first technique is *pretenuring*. The basic notion is to identify long-lived objects and create them directly in the mature generation. The goal of this technique is to reduce promotion cost, thus reducing the GC time and improving the overall performance. Blackburn et al. [27] use a profile-based approach to select objects for pretenuring. They report the reduction in GC time of up to 32% and an improvement in execution time by 7%. They also report a slight increase in heap usage with pretenuring. Harris [52] uses dynamic sampling based on overflow and size to predict long-lived objects. Subsequent work to further optimize pretenuring includes dynamic object sampling [53] and class-based lifespan prediction [54].

The second technique is to avoid performing garbage collection on newly created objects because they may not have sufficient time to die; instead, the collection effort is mostly spent on older objects. Stefanović et al. [28] propose an older-first garbage collector that prioritizes collection of older objects to give young objects more time. This technique

evolves to become a major part of the *Beltway* framework, introduced by Blackburn et al. [9]. In this framework, the heap is divided into several belts, and each belt groups one or more increments (a unit of collection) in a FIFO fashion. All objects are allocated into belt 0 (similar as the young generation). Beltway uses an older-first approach to collect each belt and all survivors are promoted to the last increment of the higher belt. Beltway always collects the oldest increment of a belt first, which gives youngest objects more time to die. The results of their experiment show an average of 5% to 10% improvement in execution times and 35% improvement under tight heaps.

It is unclear how pretenuring and the beltway framework would handle applications with a large number of longer living objects. If the decision is to pretenure these objects, then the major collection frequency would be high. On the other hand, if the heap size is enlarged to allow more time for objects to die in the nursery, then very short-lived objects are not reclaimed promptly. Similarly, each belt in the Beltway framework can be viewed as a generation. While the use of increments can avoid collection of the newly created objects, the framework still must make the decision on how to deal with longer-living objects. If belt 0 is small, these objects would be promoted to the subsequent belt, resulting in more frequent collection of the older belts. If belt 0 is large, short-lived objects are still not collected promptly.

An effort by Hibino et al. [8] investigates the degradation behavior of Web application servers running on different operating systems, including Linux, Solaris 9, FreeBSD, and Windows 2003 servers. They report that Solaris 9 has the most graceful degradation behavior. They also identify the factor that has the greatest effect on the degradation behavior as thread synchronization (waiting time to acquire locks). They report that Linux threads issue a larger number of system calls during the operation, and the thread scheduling policy is inappropriate.

8. Conclusion

This paper explored the throughput degradation behavior of a standardized Java application server benchmark. We found that throughout performance degrades ungracefully. During a period of heavy workload, a 22% increase in the workload can degrade throughout by as much as 75%. This result motivated us to investigate the major factors affecting throughput performance and how they affect degradation behavior.

We monitored execution time of three major components in the Virtual Machine: runtime compilation, garbage collection, and synchronization. Our results show that garbage collection is the major factor. GC can consume as much as 50% of the overall execution time at the heaviest workload. Further studies led us to the following conclusions:

(1) The assumption that most objects die young may not hold true in application servers. Thus, the Generation Mark-Sweep technique used in the HotSpot VM does not perform well.
(2) Garbage collection techniques that increase parallelism while greatly improving maximum throughput have very little effect on degradation behavior.
(3) Ulterior Reference Counting, an incremental generational technique, can positively impact degradation behavior.
(4) More conservative heap sizing policy only minutely affects the degradation behavior. However, it can reduce the heap usage by 20% and reduce the number of page faults by 10%.
(5) Space-based criteria to invoke garbage collection may not be the most efficient approach at heavy workload. Our study showed that the most efficient invocation points occur before the heap space is fully exhausted.

Acknowledgments

This work was sponsored in part by the National Science Foundation through awards CNS-0411043 and CNS-0720757 and by the Army Research Office through DURIP award W911NF-04-1-0104. We thank the anonymous reviewers for providing insightful comments for the final version of this paper.

References

[1] Stephen Swoyer, Impressive growth ahead for application server market, On-line article. http://www.adtmag.com/article.aspx?id=19970, January, 2007.
[2] D. Sholler, .NET seen gaining steam in DEV projects, On-line article. http://techupdate.zdnet.com, April, 2002.
[3] Sun Microsystems, Five reasons to move to the J2SE 5 platform. http://java.sun.com/developer/technicalArticles/J2SE/5reasons.html, 2005.
[4] M. Welsh, D.E. Culler, E.A. Brewer, SEDA: An architecture for well-conditioned, scalable internet services, in: Proceedings of the ACM Symposium on Operating Systems Principles, SOSP, Chateau Lake Louise, Banff, Canada, 2001, pp. 230–243.

[5] Netcraft, Video iPod launch slows apple store, On-line article. http://news.netcraft.com/archives/2005/10/12/video_ipod_launch_slows_apple_store.html, 2005.
[6] Chosun Ilbo, Cyber crime behind college application server crash, On-line article. http://english.chosun.com/w21data/html/news/200602/200602100025.html, 2006.
[7] A. Gupta, M. Doyle, Turbo-charging Java HotSpot Virtual Machine, v1.4.x to improve the performance and scalability of application servers, On-line article, http://java.sun.com/developer/technicalArticles/Programming/turbo/.
[8] H. Hibino, K. Kourai, S. Shiba, Difference of degradation schemes among operating systems: Experimental analysis for web application servers, in: Workshop on Dependable Software, Tools and Methods, Yokohama, Japan, 2005. http://www.csg.is.titech.ac.jp/paper/hibino-dsn2005.pdf.
[9] S.M. Blackburn, R.E. Jones, K.S. McKinley, J.E.B. Moss, Beltway: Getting around garbage collection gridlock, in: Proceedings of the ACM SIGPLAN Programming Languages Design and Implementation, PLDI, Berlin, Germany, 2002, pp. 153–164.
[10] R. Jones, R. Lins, Garbage Collection: Algorithms for Automatic Dynamic Memory Management, John Wiley and Sons, 1998.
[11] P.R. Wilson, Uniprocessor garbage collection techniques, in: Proceedings of the International Workshop on Memory Management, IWMM, St. Malo, France, 1992, pp. 1–42.
[12] J.L. McCarthy, Recursive functions of symbolic expressions and their computation by machine, Communications of the ACM 3 (4) (1960) 184–195.
[13] D. Ungar, Generation scavenging: A non-disruptive high performance storage reclamation algorithm, in: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on practical software development environments, 1984, pp. 157–167.
[14] J. Weizenbaum, Symmetric list processor, Communications of the ACM 6 (9) (1963) 524–544.
[15] S.M. Blackburn, K.S. McKinley, Ulterior reference counting: Fast garbage collection without a long wait, in: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, OOPSLA, Anaheim, California, USA, 2003, pp. 344–358.
[16] D.F. Bacon, C.R. Attanasio, H. Lee, V.T. Rajan, S. Smith, Java without the coffee breaks: A nonintrusive multiprocessor garbage collector, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, 2001, pp. 92–103.
[17] T.W. Christopher, Reference count garbage collection, Software Practice and Experience 14 (6) (1984) 503–507.
[18] M. Hertz, S.M. Blackburn, J.E.B. Moss, K.S. McKinley, D. Stefanović, Error-free garbage collection traces: How to cheat and not get caught, in: Proceedings of the 2002 ACM International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS, Marina Del Rey, California, 2002, pp. 140–151.
[19] M. Hauswirth, P.F. Sweeney, A. Diwan, M. Hind, Vertical profiling: Understanding the behavior of object-oriented applications, in: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA, Vancouver, British Columbia, Canada, 2004, pp. 251–269.
[20] E. Armstrong, HotSpot A new breed of virtual machine, JavaWorld.
[21] L. Zhang, C. Krintz, Profile-driven code unloading for resource-constrained JVMs, in: International Conference on the Principles and Practice of Programming in Java, PPPJ, Las Vegas, NV, 2004, pp. 83–90.
[22] L. Zhang, C. Krintz, Adaptive code unloading for resource-constrained JVMs, in: Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES, Washington, DC, USA, 2004, pp. 155–164.
[23] Sun Microsystems, White Paper: The Java HotSpot Virtual Machine. http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_1.html.
[24] S.M. Blackburn, P. Cheng, K.S. McKinley, Oil and water? High performance garbage collection in Java with MMTk, in: Proceedings of the 26th International Conference on Software Engineering, ICSE, Scotland, UK, 2004, pp. 137–146.
[25] D.F. Bacon, P. Cheng, V.T. Rajan, Controlling fragmentation and space consumption in the metronome, a Real-time garbage collector for Java, in: Proceedings of the 2003 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES, San Diego, California, USA, 2003, pp. 81–92.
[26] F. Xian, W. Srisa-an, C. Jia, H. Jiang, AS-GC: An efficient generational garbage collector for Java application servers, in: Proceedings of the 21st European Conference on Object-Oriented Programming, ECOOP, Berlin, Germany, 2007, pp. 126–150.
[27] S.M. Blackburn, S. Singhai, M. Hertz, K.S. McKinely, J.E.B. Moss, Pretenuring for Java, in: Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA, Tampa Bay, FL, 2001, pp. 342–352.
[28] D. Stefanović, K.S. McKinley, J.E.B. Moss, Age-based garbage collection, in: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, Denver, Colorado, United States, 1999, pp. 370–381.
[29] T. Yang, E.D. Berger, S.F. Kaplan, J.E.B. Moss, Cramm: Virtual memory support for garbage-collected applications, in: Proceedings of the USENIX Conference on Operating System Design and Implementation, OSDI, Seattle, WA, 2006, pp. 103–116.
[30] M. Hertz, E. Berger, Quantifying the performance of garbage collection vs. explicit memory management, in: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA, San Diego, CA, USA, 2005, pp. 313–326.
[31] D.F. Bacon, R. Konuru, C. Murthy, M. Serrano, Thin locks: Featherweight synchronization for Java, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, Montreal, Quebec, Canada, 1998, pp. 258–268.
[32] Standard Performance Evaluation Corporation, SPECjvm98 User's Guide, On-Line User's Guide. http://www.spec.org/jvm98, 1998.
[33] Standard Performance Evaluation Corporation, SPECjAppServer2004 User's Guide, On-Line User's Guide. http://www.spec.org/osg/jAppServer2004/docs/UserGuide.html, 2004.
[34] Sun, Tuning Garbage Collection with the 1.4.2 Java[tm] Virtual Machine, On-Line documentation. http://java.sun.com/docs/hotspot/gc1.4.2, (Last Retrieved: June 2007).
[35] Standard Performance Evaluation Corporation, SPECWeb2005, White paper. http://www.spec.org/web2005, 2005.

[36] JBoss, Jboss Application Server, Product Literature. http://www.jboss.org/products/jbossas, (Last Retrieved: June 2007).
[37] ObjectWeb, JOnAS: Java Open Application Server, White paper. http://www.jonas.objectweb.org, (Last Retrieved: June 2007).
[38] Standard Performance Evaluation Corporation, SPECjbb2000, White paper. http://www.spec.org/osg/jbb2000/docs/whitepaper.html, 2000.
[39] Standard Performance Evaluation Corporation, SPECjbb2005, On-Line documentation. http://www.spec.org/jbb2005, 2005.
[40] D. Stutz, T. Neward, G. Shilling, Shared Source CLI Essentials, O'Reilly and Associates, 2003.
[41] H. Lieberman, C. Hewitt, A real-time garbage collector based on the lifetimes of objects, Communications of the ACM 26 (6) (1983) 419–429.
[42] W. Srisa-an, M. Oey, S. Elbaum, Garbage collection in the presence of remote objects: An empirical study, in: Proceedings of the International Symposium on Distributed Objects and Applications, DOA, Agia Napa, Cyprus, 2005, pp. 1065–1082.
[43] JBoss Online Tutorial, Container configuration, On-line article. http://www.huihoo.com/jboss/online_manual/3.0/ch07s16.html.
[44] A.W. Appel, Simple generational garbage collection and fast allocation, Software Practice and Experience 19 (2) (1989) 171–183.
[45] D.F. Bacon, V.T. Rajan, Concurrent cycle collection in reference counted systems, in: European Conference on Object-Oriented Programming, ECOOP, Budapest, Hungary, 2001, pp. 207–235.
[46] F. Xian, W. Srisa-An, H. Jiang, Fortune teller: Improving garbage collection performance in server environment, in: Companion to the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, San Diego, CA, USA, 2005, pp. 246–247.
[47] F. Xian, W. Srisa-an, H. Jiang, Service-oriented garbage collection: Improving performance and robustness of application servers, in: Companion to the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, Portland, Oregon, USA, 2006, pp. 661–662.
[48] S.M. Blackburn, P. Cheng, K.S. McKinley, Myths and realities: The performance impact of garbage collection, in: Proceedings of the joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS, New York, NY, USA, 2004, pp. 25–36.
[49] T. Yang, M. Hertz, E.D. Berger, S.F. Kaplan, J.E.B. Moss, Automatic heap sizing: Taking real memory into account, in: Proceedings of the ACM SIGPLAN International Symposium on Memory Management, ISMM, Vancouver, British Columbia, Canada, 2004, pp. 61–72.
[50] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, M. Ogihara, Program-level adaptive memory management, in: Proceedings of the ACM SIGPLAN International Symposium on Memory Management, ISMM, Ottawa, Canada, 2006, pp. 174–183.
[51] C. Grzegorczyk, S. Soman, C. Krintz, R. Wolski, Isla vista heap sizing: Using feedback to avoid paging, in: Proceedings of the International Symposium on Code Generation and Optimization, CGO, San Jose, CA, USA, 2007, pp. 325–340.
[52] T.L. Harris, Dynamic adaptive pre-tenuring, in: Proceedings of the ACM SIGPLAN International Symposium on Memory Management, ISMM, vol. 36(1), 2000, pp. 127–136.
[53] M. Jump, S.M. Blackburn, K.S. McKinley, Dynamic object sampling for pretenuring, in: Proceedings of the ACM SIGPLAN International Symposium on Memory Management, ISMM, Vancouver, BC, Canada, 2004, pp. 152–162.
[54] W. Huang, W. Srisa-an, J. Chang, Dynamic pretenuring for Java, in: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, Austin, TX, 2004, pp. 133–140.