AS-GC: An Efficient Generational Garbage Collector for Java Application Servers

Feng Xian, Witawas Srisa-an, ChengHuan Jia, and Hong Jiang

Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115
{fxian,witty,cjia,jiang}@cse.unl.edu

Abstract. A generational collection strategy utilizing a single nursery cannot efficiently manage objects in application servers due to variance in their lifespans. In this paper, we introduce an optimization technique designed for application servers that exploits an observation that *remotable* objects are commonly used as gateways for client requests. Objects instantiated as part of these requests (*remote objects*) often live longer than objects *not created* to serve these remote requests (*local objects*). Thus, our scheme creates remote and local objects in two separate nurseries; each is properly sized to match the lifetime characteristic of the residing objects. We extended the generational collector in HotSpot to support the proposed optimization and found that given the same heap size, the proposed scheme can improve the maximum throughput of an application server by 14% over the default collector. It also allows the application server to handle 10% higher workload prior to memory exhaustion.

1 Introduction

Garbage collection (GC) is one of many features that make Java so attractive for the development of complex software systems, especially but not limited to, application servers. GC improves programmer productivity by reducing errors caused by explicit memory management. Moreover, it promotes good software engineering practice that can lead to cleaner code since memory management functions are no longer interleaved with the program logic [1, 2]. As of now, one of the most adopted GC strategies is *generational garbage collection* [3, 4].

Generational GC is based on the hypothesis that "most objects die young", and thus, concentrates its collection effort in the *nursery*, a memory area used for object creation [4]. Currently, generational collectors are configured to have only one nursery because such a configuration has proven to work well in desktop environments. However, recent studies have found the configuration to be inefficient in large server applications [5, 6] because they frequently create objects that cannot be classified as either short-lived or long-lived. Such a variance in lifespans can result in two major performance issues in any single-nursery generational collectors.

1. *A large volume of promoted objects.* If the nursery size is too small, objects with longer lifespans are promoted and then die soon after promotion. In this situation,

E. Ernst (Ed.): ECOOP 2007, LNAI 4609, pp. 126–150, 2007.

the time spent in promoting objects and higher frequency of full heap collection invocations can result in longer collection pauses and more time spent in GC.
2. *Delayed collection of dead objects.* If the nursery size is large enough to allow longer living objects more time to die, short-lived objects are not collected in a timely fashion. This scenario can result in larger heap requirement, poor heap utilization, and higher paging efforts [7, 8].

1.1 This Work

We introduce the notion of *remote* and *local* objects as a framework for identifying objects with similar lifespans in application servers. The proposed framework exploits the *key objects* notion [9], which leverages temporal locality to cluster objects with similar lifespans. In Java application servers, *remotable* objects are commonly used as gateways for client requests. Once a request arrives, many more objects are created, forming a cluster, to perform the requested service. Once the request is satisfied, most of these objects die. Studies have shown that objects connected to remotable objects tend to have longer lifespans than other short-lived objects in an application [5, 6]. Thus, our technique considers these *remotable* objects as the *key* objects and any objects connected to these remotable objects as *remote objects*. We then refer to the remaining objects as *local objects*.

We then present a new generational collector based on the notion of *remote* and *local* objects. Our garbage collector is optimized based on the hypothesis that remote and local objects have different lifespan characteristics. Therefore, managing them in two separate nurseries (i.e. local nursery and remote nursery) will result in better garbage collection efficiency, as each nursery can be optimally sized based on the allocation volume and lifespan characteristic of the residing objects. Garbage collection in each nursery can be done independently of the other nursery, and the surviving objects from both nurseries are promoted to a shared mature generation. A low-overhead run-time component is used to dynamically identify and segregate remote and local objects. We have extended the generational collector in the HotSpot virtual machine (we refer to the HotSpot's collector as the *default* collector) to support the proposed optimization technique (we refer to the optimized version as the *collector for application server* or *AS-GC*). We then compared the performance of AS-GC with that of the highly tuned default collector. The results of our experiments indicate that our proposed scheme yields the following three benefits.

1. *Timely object reclamation.* The results show that the minor collectors of the local and remote nurseries are called more frequently, and each time, the percentage of surviving objects is lower than that of the default collector. Higher frequency of minor collection invocations means that our approach attempts to recycle objects quickly. Higher efficiency means that fewer objects are promoted, leading to shorter pauses, fewer major collection invocations, and less time spent in garbage collection.
2. *Higher throughput.* Given the same heap space, our collector yields 14% higher maximum throughput than that of the default collector. This improvement is achieved with negligible runtime overhead.

3. *Higher workload.* With the default collector, the throughput performance degrades significantly due to memory exhaustion when the workload reaches a certain level. Because our scheme is more memory efficient, it can operate with less heap space. Therefore, it can handle 10% higher workload before the same exhaustion is encountered.

Even though our proposed solution is domain-specific, it should have great potentials for a wider adoption by language designers and practitioners as the application server market is one of the biggest adoptors of Java [10]. It is worth noting that our approach is significantly different from the existing techniques to improve the efficiency of garbage collection (e.g. pretenuring, older-first, and Beltway [1, 11, 12, 13]). However, our approach can also be integrated with these techniques to achieve even higher GC efficiencies.

The remainder of this paper is organized as follows. Section 2 describes the preliminary studies and discusses the results that motivate this work. Section 3 provides an overview of the proposed technique and implementation details. Section 4 details the experimental environment. Section 5 describes each experiment and reports the results. Section 6 further discusses the results of our work. Section 7 provides an applicability study of this work. Section 8 highlights some of the related work, and the last section concludes this paper.

2 Why Design a Garbage Collector for Application Servers?

> "It has been proven that for any possible allocation algorithm, there will always be the possibility that some application program will allocate and deallocate blocks in some fashion that defeats the allocator's strategy."
>
> Paul R. Wilson *et al.* [14]

The same argument can be made about garbage collection. Most garbage collectors, shipped as part of any commercial Java Virtual Machines (JVMs), are based on the generational approach utilizing a single nursery. While such a collection strategy has worked well for Java over the past decade, studies have shown that objects in Java application servers may not always be short-lived [5, 6], leading to an inefficiency of any single-nursery generational collector.

Longer living objects in these server applications can degrade the efficiency of these collectors. When this happens, the throughput performance of these server applications can seriously suffer. Such inefficiency can also result in poor memory utilization [7], leading to a large number of page faults under heavy workload, ungraceful degradation of throughputs and failures [6].

In the remainder of this section, we highlight some of the differences in run-time characteristics between *desktop applications* and *application servers*. We then report the result of our experiments to investigate the lifespan characteristics in these applications and the differences in the performance of generational collection in these two types of applications.

Table 1. Comparing the basic characteristics of SPECjvm98, SPECjbb2000, and SPECjAppServer2004

Characteristic	SPECjvm98	SPECjbb2000 (8 warehouses)	SPECjAppServer2004 (Trans. rate = 40)
# of simultaneous threads	5 (in MTRT)	11	331
# of allocated objects	8 million (in Jess)	33 million	80 million
Amount of allocated space	231 (in db) MB	900 MB	5.1 GB
Total execution time	seconds	minutes	hours

2.1 Basic Characteristics of Application Servers

Application servers often face significant variations in service demands, the higher demands often coincide with "the times when the service has the most value" [15]. Thus, these servers are expected to maintain responsiveness, robustness, and availability regardless of the changing demands. Past studies have shown that under the heaviest workload, the resource usage can be so intense that, often times, these servers would suddenly fail with little or no warning [6, 16, 17, 18].

To better understand the differences in resource usage between desktop applications and application servers, we conducted an experiment to compare their basic runtime characteristics (see Table 1). We used SPECjvm98, SPECjbb2000, and SPECjAppServer2004 in our study. SPECjvm98 [19] is a commonly used benchmark suite in the research community. All applications in the suite are designed to run well in general purpose workstations. SPECjbb2000 [20] is a server benchmark designed to emulate the application server tier. It does not make any connections to external services (e.g. database connections). On the other hand, SPECjAppServer2004 [21] is a benchmark for real-world application servers designed to run on high-performance computer systems (more information about this benchmark is available in Section 4).

From Table 1, the differences in memory requirement and degree of concurrency can translate to much higher resource usage in server applications. However, they do not yield any insights into the differences in lifespan of objects between these two types of applications. Therefore, we conducted further experiments to compare their lifespan characteristics.

2.2 An Experiment to Evaluate Lifespans of Objects in Server Applications

We measured lifespan by the amount of memory allocated between birth and death (in bytes)[1]. We measured the execution progress by the accumulated amount of allocated memory (also in bytes) [11]. Figure 1 depicts our findings.

The vast majority of objects in *Jess*, a benchmark program in the SPECjvm98 suite, are short-lived; that is, most objects have lifespans of less than 10% of the maximum lifespan (as illustrated in 1a). Note that we also conducted similar studies using other

[1] We only accounted for objects allocated in the garbage-collected heap and ignored any objects created in the permanent space.

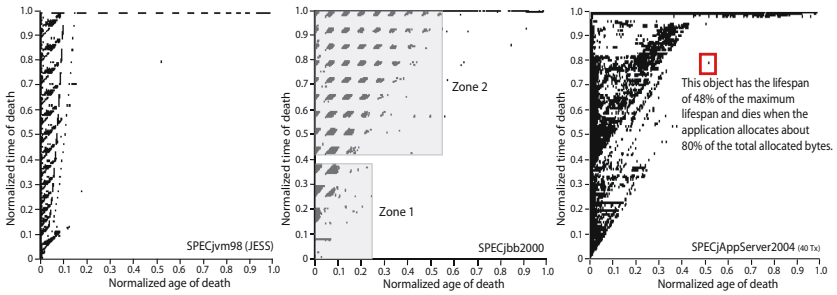


Fig. 1. Each dot in these graphs represents an object. The x-axis represents the normalized age of death, and the y-axis represents the normalized time of death. Thus, the position of each dot provides us with the age of that particular object and the time that it dies. For example, the squared object in the SPECjAppServer2004 graph (c) has a lifespan of 48% of the maximum lifespan and dies when the application allocates about 80% of the total allocated bytes.

applications in the SPECjvm98 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 [3, 4].

On the other hand, large numbers of objects in the SPECjbb2000 with 8 warehouses (Figure 1b) and SPECjAppServer2004 with 40 Tx (Figure 1c) have lifespans of up to 30% to 50% of the maximum lifespans. It is worth noting that there are more objects with longer lifespans as these programs approach termination (as indicated by the triangular patterns). This is to be expected as the amount of work in each benchmark becomes heavier as the program continues to run. For example, SPECjbb2000 starts with a single warehouse and creates one more warehouse each time it finishes making the queries. In our experiment, this process continues until 8 warehouses are created. It is worth noting that the clusters of dead objects (appeared in Figure 1b as groups of dark spots) correspond to the number of warehouses created and worked on by the application.

Our past research effort on .NET server applications also indicates a similar lifespan behavior to the Java server benchmarks [5]. We hypothesize that such behavior is a result of a high degree of concurrency in these server applications (see more discussion about this issue in Section 6). If concurrency is indeed the main factor for such a lifespan behavior, it is also possible for multithreaded desktop applications to exhibit a similar behavior. Since most of the available desktop benchmarks are not heavily multithreaded, we have yet to conduct further experiments to validate our hypothesis. Such experiments will be left for future work.

Next, we conducted an experiment to investigate the efficacy of the generational collector in the SPECjbb2000 benchmark. Our investigation focused on two execution areas: the first 40% of execution (*zone 1* of Figure 1b) where most objects are still short-lived and the last 60% of execution (*zone 2* of Figure 1b) where most objects are long-lived. We observed the following results.

1. *Generational collector performs efficiently in zone 1.* Figure 1b clearly shows that objects in this zone can be easily segregated into short-lived and long-lived. While executing in this zone, the generational scheme performs very efficiently.
2. *Generational collector is not efficient in zone 2.* Figure 1b shows that the lifespans cannot be easily classified into the short-lived and long-lived taxonomy. Therefore, the generational collector begins to lose its efficiency upon entering this zone. We also noticed that the heap size is increased dramatically even though the number of objects created in this zone is only twice as much as that of zone 1.

The lifespan behavior as depicted in zone 2 poses two important challenges to generational collectors. First, *if the nursery size is set too small, minor collection may promote a significant number of objects*. A large volume of promoted objects can cause the pause times to be long. Moreover, these promoted objects can result in more frequent collection of the older generation. This observation is reported by Xian et al. [6].

Second, the nursery may need to be set to a much larger size to allow objects with diverse lifespans sufficient time to die. Our study shows that the performance differences due to larger nursery sizes without increasing the overall heap size, are not noticeable. To yield a better performance, the entire heap space must be enlarged to provide a sufficient GC headroom. With a larger nursery, *the truly short-lived objects are not collected in a timely fashion and continue to occupy the heap space*, resulting in a much larger heap requirement, as noted in zone 2 and reported by Hertz and Berger [7].

In the next section, we provide the detailed information about the proposed generational collector designed to address these two challenges.

3 A Generational Collector for Application Servers (AS-GC)

In this section, we discuss a notion called *key objects* that is used to optimize the proposed generational strategy. We also discuss three major runtime components, dynamic objects segregation mechanism, nurseries management, and inter-type reference tracking mechanism that we implemented in HotSpot.

3.1 Defining Key Objects

Our work leverages the previous research on *Key Objects* to dynamically identify clusters of similar-lifespan objects [9]. Hayes defines *key objects* as "clusters of objects that are allocated at roughly the same time, and live for roughly the same length of time" [9]. In other words, the idea is to segregate objects into groups based on temporal locality and lifespan similarity. Our technique considers *remotable* objects as the key objects. Any objects connected to these remotable objects become part of their clusters and are assumed to have similar lifespan [9, 22, 23]. As stated earlier, these objects are referred to as remote objects, and any objects that are not part of these clusters are referred to as local objects. These two types of objects, once identified, will be managed in two separate nurseries.

3.2 Dynamic Objects Segregation

Our next step is to efficiently segregate local and remote objects. While the segregation process can be done statically [24], we chose a dynamic scheme because the distinction between remote and local objects can be easily done at run-time. Our scheme detects when remote methods are invoked. While these remote methods are still in scope, any newly allocated objects are considered remote.

In HotSpot [25], methods, classes and threads are implemented by *methodOop*, *KlassOop* and *Thread* objects, respectively. To segregate remote and local objects, we added a new flag bit, *is_remote* to *methodOop* to indicate that the corresponding method is remote. If a method belongs to any interfaces that extend *java.rmi.Remote* (e.g., some enterprise Beans, EJBHome or EJBObject interface), we set this flag. Otherwise, the flag remains unset.

For each thread, we also added a simple attribute *CallTreeDepth* to record the depth of the current call tree on the thread. At every method entry and exit, the *CallTreeDepth* is incremented or decremented accordingly. Particularly, when a thread first makes a remote method call, the method's information and the depth of the call tree are recorded. When a remote method call exits, the corresponding recorded information is also deleted. If a thread still maintains information about a remote method call, it means that the remote method call is still in scope, so all objects created during this time are categorized as remote objects.

There are two major sources of overhead in the type segregation process: bookkeeping of remote method calls and remote/local-type checking. In our implementation, three comparison operations are performed at every method entry and exit. In type-checking, only two comparison operations are needed to determine if an object is remote. Through experiments, we found the overhead of the type segregation process to be roughly 1% of the total execution time.

3.3 Local and Remote Nurseries

Organization: Once the type of an object is identified, the next step is to create the local and remote nurseries to host local and remote objects, respectively. Since we are extending the heap organization of HotSpot to support our proposed scheme, we first outline the heap layout adopted by HotSpot (as shown in Figure 2a).

The HotSpot VM partitions the heap into three major generations: nursery, mature, and permanent, which is not shown in Figure 2. The nursery is further partitioned into three areas: *eden* and two survivor spaces, *from* and *to*, which account for 20% of the nursery (i.e. the ratio of the eden to the survivor spaces is 4:1). Object allocations initially take place in the *eden* space. If the *eden* space is full, and there is available space in the *from* space, the *from* space is used to service subsequent allocation requests.

Figure 2b illustrates our heap organization. Our technique simply extends the existing heap organization to create two nurseries instead of just one. Within each nursery, the heap layout is similar to that of HotSpot (an eden space and two survivor spaces). The local and remote nurseries can be individually and optimally sized to match the lifespan characteristics of the local and remote objects, respectively.

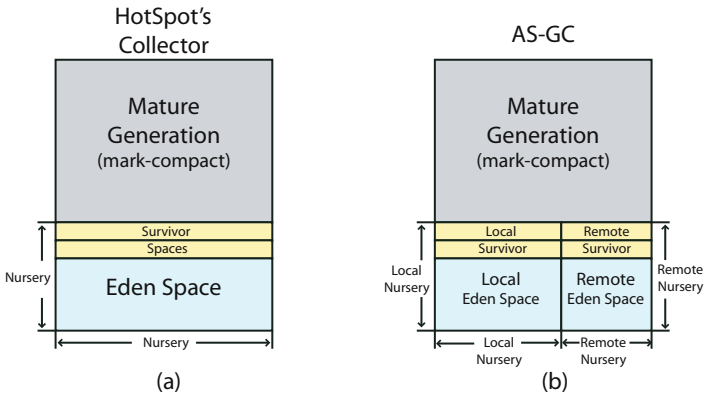


Fig. 2. Comparing the heap organizations of HotSpot and the proposed AS-GC

Garbage collection in HotSpot. We refer to the collection scheme in HotSpot as *GenMS*. In this technique, *minor collection* is invoked when both the *eden* and *from* spaces are full. The collection process consists mainly of copying any surviving objects into the *to* space and then reversing the names of the two survivor spaces (i.e. *from* space becomes *to* space, and vice versa). Thus, the *to* space is always empty prior to a minor collection invocation [25].

The *to* space provides an aging area for longer living objects to die within the nursery, assuming that the volume of surviving objects is not larger than the size of the *to* space. If this assumption does not hold, some surviving objects are then copied directly to the mature generation. When the space in the mature generation is exhausted, *full* or *mature* collection based on mark-compact algorithm is used to collect the entire heap. It is worth noting that the aging area is only effective when the number of copied objects from the eden and the *from* spaces is small. If the number of surviving objects becomes too large (such as in application servers), most of these objects are promoted directly to the mature generation, leading to more frequent mature collection invocations.

Sizing of each nursery: The process to identify the optimal nursery sizes consists of two steps. First, we conducted a set of experiments to identify the optimal ratio between the nursery and the mature space in GenMS. We found that the nursery to mature ratio (*nursery/mature ratio*) of 1:2 (i.e. 33% nursery and 67% mature) yields the optimal throughput performance for our benchmark. This ratio is then used to further configure the local and remote nurseries; that is, the sum of the local and remote nurseries is equal to the nursery size of GenMS. As a reminder, our research objective is to show that our technique is more efficient than GenMS, given the same heap space, and thus, the same nursery size is used.

The second step involves conducting another set of experiments to identify the *local/remote ratio* (size of local nursery / size of remote nursery). We initially anticipated

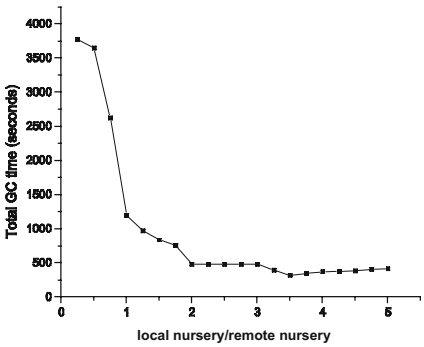


Fig. 3. Identifying optimal local/remote ratio

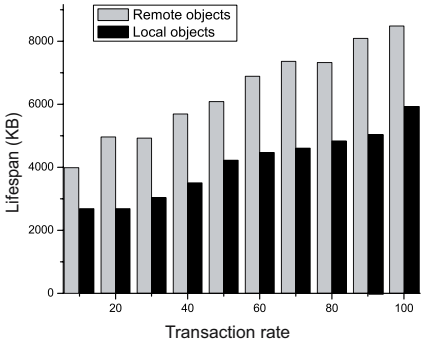


Fig. 4. Lifespans of remote and local objects in GenMS

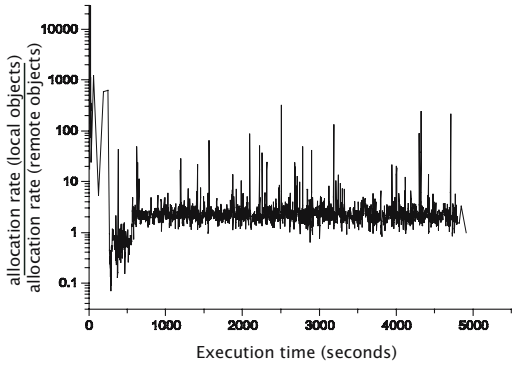


Fig. 5. Allocation rate is defined as the volume of allocated bytes over a period of time. In this experiment, we considered the allocation rates of remote and local objects separately. We then calculated the ratio of local allocation rates to remote allocation rates throughout the execution.

the remote nursery to be larger than the local nursery due to the results of previous studies indicating that the remote objects are longer living. To our surprise, the result of our experiment (depicted in Figure 3) indicates that the local nursery should be at least 3 times larger than the remote nursery.

To better understand why our result is counter-intuitive, we conducted an experiment to validate the previous claims that remote objects are longer living [5, 6]. Our result clearly shows that the claim is valid; remote objects indeed live significantly longer than local objects (see Figure 4). We then investigated the allocation behavior and discovered a valuable insight. The median allocation rate (volume of allocated objects over time) of local objects is three times higher than that of remote objects (see Figure 5). Periodically, the allocation rates of the local objects can be several hundred times higher than those of remote objects. We also noticed that during the initial phase of execution, there are no allocations of remote objects at all. This is expected as all services must be initialized locally prior to taking remote requests.

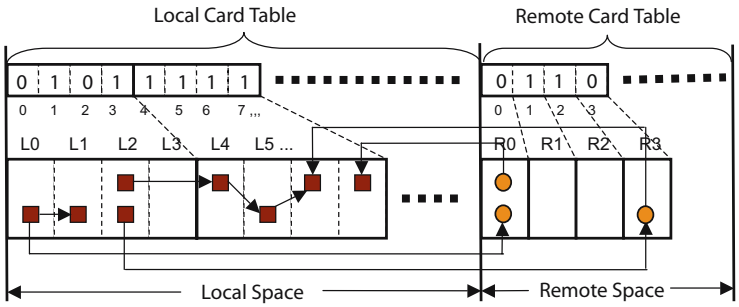


Fig. 6. Each round object represents a remote object, and each square object represents a local object. The nurseries are divided into 128-byte blocks; each block is represented by one byte card allocated in a separated card table area. Initially, each byte is set to the value of 1. When an inter-nursery reference is made (write-barrier is used to detect such a reference), the card representing the memory block that contains the inter-nursery reference becomes dirtied and is assigned the value of 0. In the example shown here, there are four inter-nursery references originated from memory blocks L0, L2, R0 and R3; thus, those cards have the value of 0.

This finding suggests that a possible dominating factor in determining the local/remote ratio is the allocation rate. It is also very likely that the allocation rate can influence the lifespans of objects in GenMS (will be discussed in Section 6).

3.4 Tracking Inter-nursery References

It is possible for objects to make inter-nursery references (i.e. a reference originated from the remote nursery to an object in the local nursery, and vice versa). Thus, we need a mechanism to track these inter-nursery references. Through a preliminary experiment, we discovered that it is common for the number of references from the remote nursery to the local nursery to be as many as 20 times higher than those from the local nursery to the remote nursery. This is likely because many of the services in these servers are done by worker threads created during the initialization. This observation led us to design a card table mechanism that uses two different scanning granularities for the local and remote nurseries to reduce scanning time. Figure 6 illustrates the organization of our card tables.

When a minor collection is invoked in the local nursery, the *remote card table* is scanned to locate any inter-nursery references coming from the remote nursery in a *fine-grained way* (byte by byte). This is because the volume of the inter-nursery references coming form the remote space tends to be very high. For each encountered dirty card, the memory block is further scanned to locate inter-nursery references. Note that the mechanism to record inter-generational references (references from the mature space to the nurseries) [4, 2] is already provided by HotSpot. Thus, we do not need to implement such a mechanism.

On the contrary, when a minor collection is invoked in the remote nursery, the *local card table* is scanned in a *coarse-grained way* (word by word[2]). This is because there

[2] In our experimental system, one word is corresponding to four bytes.

are fewer inter-nursery references originated from the local space. For every dirty word, the collector then identifies each dirty card within the word before proceeding to scan the corresponding memory block to locate any potential inter-nursery references. So for the local card table, each card is marked in the fine-grained way but scanned in the coarse-grained way, and thereby, reducing the cost of scanning.

4 Experimental Environment

In this section, we describe our experimental environment consisting of an application server and a benchmark program. We also provide the detailed information about the computing platforms and the operating environments in which the experiments were conducted.

4.1 Application Server and Workload Driver

There are two major software components in our experiment, the Application Servers and the workload drivers. We investigated several server benchmarks and selected JBoss [26] as our application server. JBoss is by far the most popular open-source Java Application Server (with 25% of market share and over fifteen million downloads to date). It fully supports J2EE 1.4 with advanced optimizations including object cache to reduce the overhead of object creation. Note that MySQL[3] is used as the database server in our experimental environment.

In addition to identifying the application server, we need to identify workload drivers that create realistic client/server environments. We chose an application server benchmark, jAppServer2004 from SPEC [21], which is a standardized benchmark for testing the performance of Java Application Servers. It emulates an automobile manufacturing company and its associated dealerships. The level of workload can be configured by *transaction rate* (Tx). This workload stresses the ability of the Web and EJB containers to handle the complexities of memory management, connection pooling, passivation/activation, caching, etc. The throughput of the benchmark is measured in JOPS (job operations per second).

4.2 Experimental Platforms

To deploy SPECjAppServer2004, we used three machines to construct the three-tier architecture. The client machine is an Apple PowerMac with 2x2GHz PowerPC G5 processors with 2 GB of memory and runs Mac OS-X. The application server is a single-processor 1.6 GHz Athlon with 1GB of memory. The database server is a Sun Blade with dual 2GHz AMD Opteron processors with 2GB of memory. The database machine and the application server run Fedora Core 2 Linux.

In all experiments, we used the HotSpot VM shipped as part of the Sun J2SE 1.4.2 [25] to run the JBoss application server. Unless specified differently in the next section, the heap space was limited to 2 GB (twice the amount of physical memory). The

[3] Visit www.mysql.com for more information.

nursery/mature ratio was set to the optimal value of 1:2, and the local/remote ratio was selected to be 3:1. We conducted all experiments in a standalone mode with all non-essential daemons and services shut down.

5 Results and Analysis

In this section, we report the experimental results focusing on the following performance metrics: garbage collection time, garbage collection efficiency and frequency, maximum throughput, memory requirement, and workload capacity.

5.1 Garbage Collection Behaviors

We first measured the GC frequency. As shown in Table 2, our collector invokes the minor collection more frequently than the GenMS approach. This is not necessarily a bad thing. Higher frequency of minor collection invocations can translate to reduced heap requirement if each of these invocations is effective in collecting dead objects. As reported in the table, the average survival rate[4] of the proposed scheme is consistently lower than that of GenMS when the same transaction rate is applied. Because the local and remote nurseries are also smaller than the nursery in GenMS, the volume of the promoted objects in our scheme is also lower.

Table 2. Comparing survival rates

Normalized workload (%)	GenMS		AS-GC			
	Minor collections	Survival rate	Minor collections		Survival rate	
			Local	Remote	Local	Remote
10	695	3.7%	976	26	3.6%	3.7%
20	1019	5.8%	1230	81	4.8%	4.9%
30	1981	6%	2204	401	5.1%	5.2%
40	2913	6.8%	3201	1098	5.6%	5.1%
50	3707	7.1%	3520	1233	6.8%	6.1%
60	4506	8.2%	4501	1622	6.9%	7.0%
70	5102	8.9%	5020	1903	7.0%	7.1%
80	6278	9.7%	6409	2411	8.2%	7.9%
90	7150	10.9%	7533	2702	9.0%	9.0%
100	8008	12.9%	8904	3202	10.1%	10.2%

More efficient minor collection translates to fewer full collection invocations (see Figure 7). At the maximum workload (Tx = 100), the reduction can be as much as 20%. Fewer full collection invocations also result in less time spent in GC; the reduction in GC time ranges from 25% to 32% when the workload is above 30 Tx (see Figure 8).

In terms of GC pauses, we report our results based on the concept of *Bounded Minimum Mutator Utilization* (BMU) [27]. Figure 9 shows BMU of GenMS and AS-GC at the initial decline of throughput (50 Tx). The x-intercept indicates the maximum pause

[4] The survival rate is the percentage of objects that survives each minor collection.

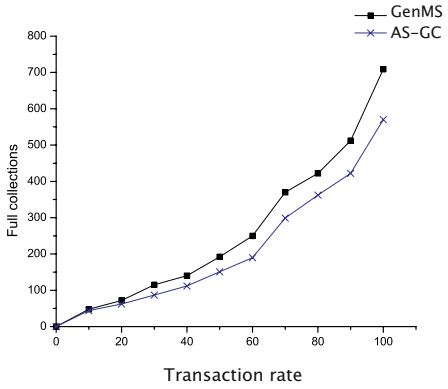


Fig. 7. Comparing major collection frequency

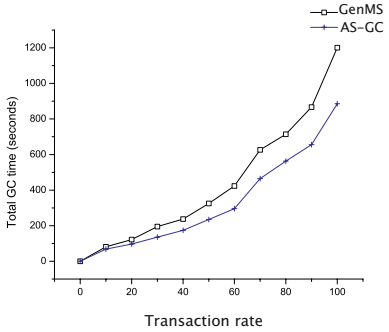


Fig. 8. Comparing overall collection times

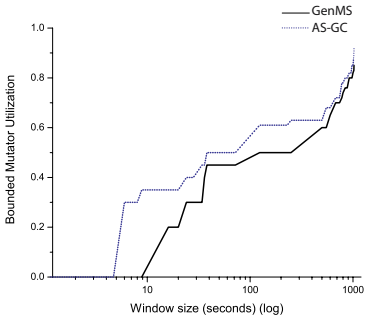


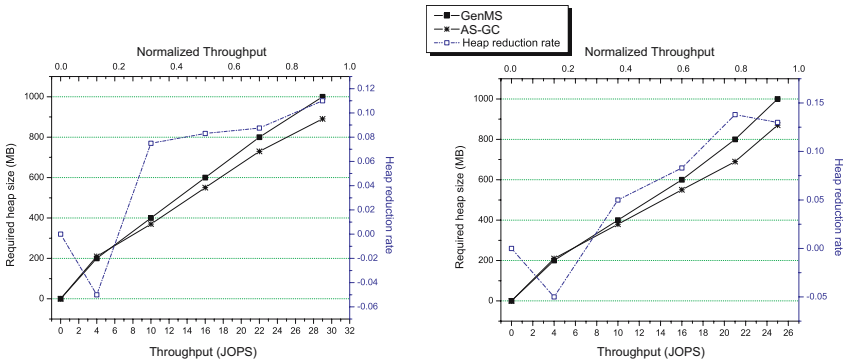
Fig. 9. Comparing BMUs

time, and the asymptotic y-value indicates the fraction of the total time used for the mutator execution (average utilization). Note that we considered the additional time to track the inter-nursery references as a component of the GC time.

Full collection invocations dominate the GC pause times. On average, the pause times are usually more than 1 second. As shown in the graph, GenMS has the largest x-intercept value of around 8.83 seconds, and its utilization is about 10% lower than that of AS-GC (occurs around 200 seconds). The x-intercept of AS-GC is significantly smaller, because of less copying overhead. The overall utilization (asymptotic y-value) of AS-GC is about 5% higher than that of GenMS.

5.2 Heap Utilization

We define heap utilization as the amount of heap space needed to yield a certain throughput performance. Thus, if two systems yield the same throughput but one requires a smaller heap space, that system has better heap utilization. To evaluate this performance metric, we measured the heap space required by the two collectors to yield the same



(a) Workload = 20 Tx (b) Workload = 40 Tx

Fig. 10. Comparing heap usage between two workloads

throughput given the same workload. We chose two workload levels, 20 Tx and 40 Tx. At 20 Tx, both collectors achieve their corresponding maximum throughputs. At 40 Tx, the heap becomes tight but the application still maintains acceptable throughput performance (see Section 5.3).

For the experimental methodology, we measured the throughputs of GenMS under different heap sizes ranging from 200MB to 1GB. (We chose 1GB to minimize the effect of paging.) We then varied the heap size of AS-GC until we achieved the same throughputs as delivered by GenMS. The ratio between mature and nursery spaces is maintained at two to one.

Figure 10a reports our findings when the workload is set to 20 Tx. The solid lines in the graph illustrate the required heap sizes (left-side y-axis) of the two GC techniques to achieve the throughput specified in the lower x-axis. The dotted line is used to show the heap reduction percentage of AS-GC (right-side y-axis) over GenMS, based on the normalized throughput (top x-axis).

As shown in Figure 10a, once the heap size is large enough to handle the specified workload level (over 200 MB), AS-GC requires smaller heap space to achieve the same throughput as GenMS. When 20 Tx is used, we see the heap size reduction of 11%. When the workload is 40 Tx (see Figure 10b), AS-GC uses 13.4% smaller heap to deliver the same throughput. Since paging is not a major factor in this experiment, the main reason for better heap utilization is our collector's ability to collect dead objects more quickly and more efficiently.

5.3 Throughput

We conducted a set of experiments to measure the throughput of each collector. Each measurement was done using the same workload and the same heap size. This time, we allowed the size to be as large as 2GB so that we can evaluate the effect of AS-GC on paging. Figure 11 illustrates the throughput behavior of SPECjAppServer2004 utilizing GenMS and AS-GC. Figure 12 reports the percentage of improvement in throughput performance when AS-GC is used.

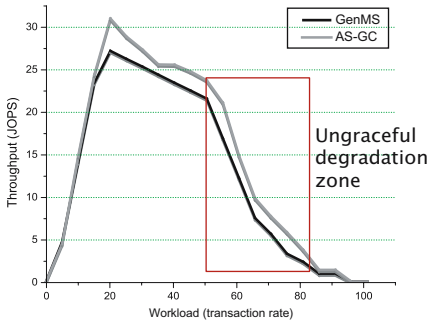


Fig. 11. Comparing throughputs

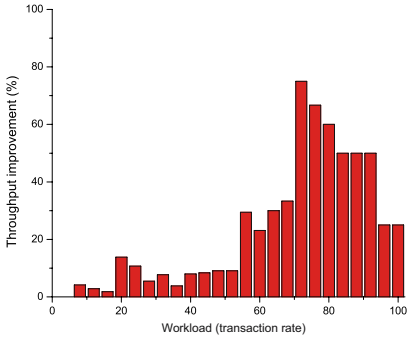


Fig. 12. Illustration of throughput performance improvement

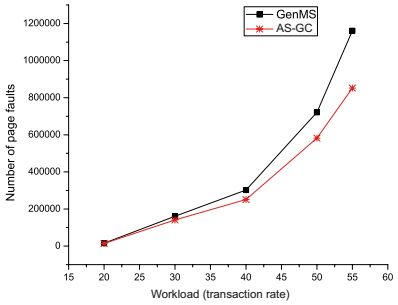


Fig. 13. Comparing paging behavior under heavy workloads

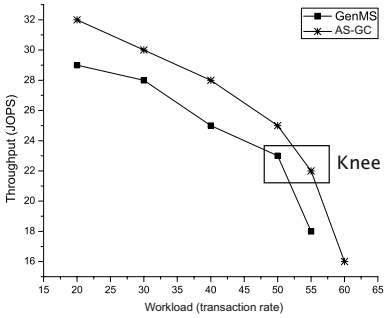


Fig. 14. Comparing throughput degradation locations

Notice that we can achieve about 14% throughput improvement when the workload is 20 Tx. Once the workload is around 50 to 55 Tx, the amount of heap space needed to execute the program exceeds the available physical memory (1GB). At this point, the system rapidly loses its ability to respond to user's requests. As the heap become tighter and tighter, the throughput improvement can range from 30% (55 Tx) to 78% (70 Tx). However, the system, when facing such high demands, is suffering from excessive paging (see Figure 13). While the percentage of improvement is large, the actual throughput delivered by the system is very small. It is worth noting that the main reason for a 30% improvement in the throughput performance when the transaction rate ranges from 55 to 65 is due to a significant reduction in the paging effort.

5.4 Ability to Handle Heavier Workload

To evaluate our collector's ability to handle varying workload, we set the initial workload to 20 Tx and the heap size to 1GB to minimize the effect of paging. We executed

SPECjAppServer2004 using this initial configuration. We then gradually increased the workload until we could precisely identify a period of execution where the throughput performance degraded sharply (Figure 11).

From Figure 14, the throughput of AS-GC degrades drastically at 55 Tx while the throughput of GenMS degrades at 50 Tx. This difference translates to 10% higher workload capacity before failure. By utilizing the heap space more efficiently, AS-GC should be able to respond to an unanticipated workload-increase better than GenMS.

6 Discussion

In this section, we provide a discussion about a runtime phenomenon called *lifespan interference* that occurs when multiple threads share the same nursery. We also discuss the feasibility of applying region-based memory management as an alternative to our approach to improve the performance of application servers.

6.1 Lifespan Interference

When a heap is shared by mulitple threads, thread scheduling performed by the underlying operating system can significantly affect lifespans of objects belonging to a thread. We refer to such an effect on lifespans due to scheduling as *lifespan interference*, which is illustrated in Figure 15.

In Figure 15a, *Thread 1* ($T1$) allocates *object a*, *object b*, and *object c* before making an I/O access. At this point, the operating system would suspend the execution of $T1$. Since there are no other threads allocating objects from the same heap as $T1$ in this scenario, the lifespan of every object in *T1* can be easily calculated based on the object allocation pattern of $T1$. Thread scheduling by the operating system has no effect on lifespan in a single-threaded environment. Thus, the lifespan of *object a* is 3 because *objects b*, *c*, and *d* are created during the lifetime of object *a*.

In Figure 15b, $T1$, $T2$, and $T3$ share the same heap. Again, $T1$ is suspended by the operating system during the I/O access. Let's further assume that the scheduler picks

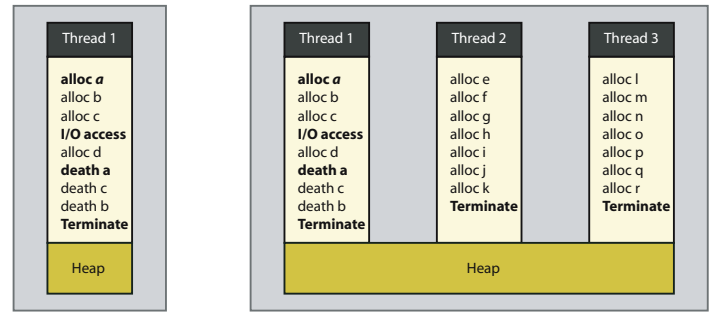


Fig. 15. What is the lifespan of object a?

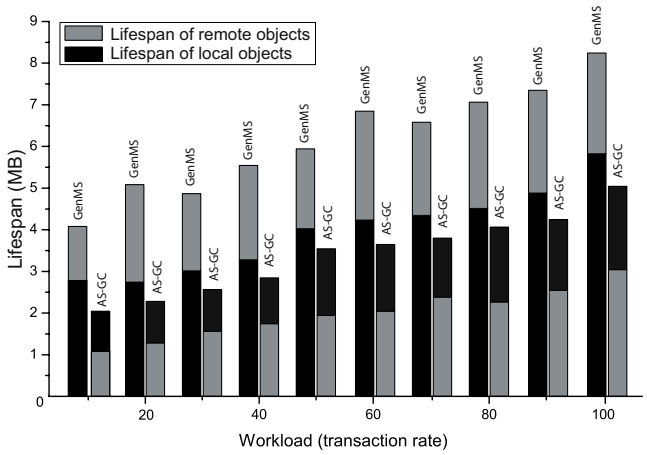


Fig. 16. Comparing the lifespans of remote and local objects when GenMS and AS-GC are used

$T2$ to run next. At this point, the lifespan of object a must include objects created by $T2$. Notice that in the example, the execution of $T1$ does not depend on any objects created by $T2$, but these objects can greatly affect the lifespans of objects created by $T1$. Depending on how long $T1$ is suspended, the lifespan of object a can be any values ranging from 3 to 17 (when both $T2$ and $T3$ complete their execution before $T1$ is resumed).

Lifespan interference is one reason why GenMS is not very efficient in large multi-threaded server applications. Objects that should be short-lived (according to the per-thread-allocation pattern) can appear to be much longer living due to scheduling. By segregating remote and local objects into two separate nurseries, the lifespans of remote and local objects are determined by the number of object allocations in remote and local nurseries, respectively. Figure 16 depicts the lifespans of objects in SPECjAppServer2004 when the GenMS and AS-GC approaches are used. In all workload levels, remote objects are longer living than local objects when GenMS is used. However when AS-GC is used, the lifespans of remote objects are reduced by as much as 75% (transaction rate = 20). In fact, the lifespans of the local objects are now much longer than those of the remote objects.

To put this into perspective, we compare the effect of interferences in our approach with existing approaches. It is clear that our technique provides better isolation from interferences than the shared nursery technique. On the other hand, our technique is not as isolated as techniques such as thread local heaps [28] or thread specific heaps [29], which create a subheap for each thread. However, it is unclear how well such approaches would perform given a large number of threads created in these server applications. For example, a study by [30] has shown that when each thread gets its own sub-heap, the memory utilization tends to be poor due to unused memory portion within each sub-heap. In addition, a study by [28] also reports that the run-time overhead to perform dynamic monitoring of threads may offset the improvement in garbage collection performance obtained through the thread local heap approach.

6.2 Region-Based Memory Management

Another approach to improving the performance of application servers from the memory management perspective is to utilize *region-based* memory management [31,32,33,34] instead of or in conjunction with garbage collection. In region-based memory management, each object is created in a program specific region [31]. When a region is no longer needed, the entire area is reclaimed. One notable example of using region-based memory management in Java is scope-memory adopted in the *Real-Time Specification for Java* [35]. In this approach, a region is created for each real-time thread. The lifetime of objects created in this region is strictly bounded to the lifetime of the thread owning the region; as the thread terminates, the region is also destroyed.

As stated earlier, the lifetime of a remote object in an application server tends to be bounded by the time taken to complete a request or a task. Thus, it may be possible to bound the lifetime of a region to a task. However, our investigation of application servers also reveals many runtime factors that can make the deployment of region-based memory management in application servers challenging. First, not all objects created during a request are task-bounded. Techniques such as object caching [26] and HTTP sessions allow objects to outlive the task that creates them. Second, the time taken to complete a task can vary from a few seconds to over twenty minutes. Third, within each task there are tens to hundreds of threads that cooperate to complete a task. Fourth, within each task there can be hundreds of garbage collection attempts that yield efficient result, meaning that there is plenty of memory space to be recycled prior to the task termination. We have partially attempted a few solutions, discussed below, that can potentially address these factors.

Identifying task-bounded objects. Compile-time analysis may be employed to segregate task-bounded objects from non-task-bounded objects [31,32,33,34]. However, this implies the accessibility of source programs, which may not be made readily available by commercial software vendors. Moreover, standardized interfaces need to be established to allow VMs created by multiple vendors (e.g. Sun, BEA, IBM, etc.) to exploit the information generated by the compiler. Currently, we are extending our experimental framework to support this solution.

Reducing heap requirement. In region-based memory management, unreachable objects are not reclaimed until the end of the task. The policy of not reclaiming these objects can severely degrade performance and affect robustness of application servers especially when the memory demand is high, but the unused memory is not timely recycled [6]. One solution to conserve the heap usage is to combine region-based memory management with garbage collection [36,32,33]. It is unclear if this technique will yield higher performance improvement than the proposed AS-GC approach. We are currently experimenting with this proposed technique.

Identifying short-running tasks. Committing a memory region to a task for a long period of time may not be feasible as dead objects are not recycled promptly. However, short-running tasks may benefit from region-based memory management. The selection of the short-running tasks entails identifying threads that participate in each of these short-running tasks. Once these threads are identified, each will be directed to allocate objects in a specific region. While the idea appears to be straight-forward, a practice

of thread pooling may drive up the cost of dynamically identifying these threads. With thread pooling, the analysis may have to be performed constantly as one thread can participate in both short-running and long-running tasks.

7 Applicability Study

In this section, we discuss four important issues that can greatly impact the applicability the proposed approach: generalization, alternative nursery configurations, required tuning efforts, and possible integrations with existing optimization techniques.

Generalization. To demonstrate that our solution can be generalized beyond the benchmark that was used, we conducted a preliminary study to compare the performance of AS-GC and GenMS using a different application server benchmark, SPECjAppServer2002. It is an outdated version of SPEC standardized application server benchmark. It conforms to the older J2EE standard (version 1.2) and also utilizes a different connection mechanism [37]. Our result indicates that we can achieve similar performance gains with AS-GC (14.8% higher maximum throughput). For future work, we will experiment with other commercial application servers as well as workload drivers to further validate the generalizability of our solution.

Nursery Configurations. In our experiments, we configured AS-GC to have the same nursery/mature ratio as GenMS throughout. However, it is possible for the performance of AS-GC to be different if other nursery/mature and local/remote ratios are used. As a preliminary study, we investigated the performance of AS-GC under two other nursery/mature heap configurations (1:3 and 1:1). We found that an additional 2% improvement in the throughput performance can be achieved with a larger nursery (1:1). This finding tells us that better results may be obtainable. As future work, we will conduct more investigation on the effect of heap configuration on the performance of the proposed AS-GC.

Tuning Efforts. Currently, heap tuning is recommended by application server vendors as a way to achieve maximum performance [5]. In our experiments, we used a standard parameter—used by practitioners for the tuning purpose—to set the nursery/mature ratio. To facilitate tuning of the local/remote ratio, we created a new command-line-configurable parameter to allow users to fully utilize our collector. While the tuning process can be tedious, it is a common procedure, and our proposed scheme only requires a small effort in addition to the current tuning practice.

Optimization. Fundamentally, our collector is a variation of copying-based generational collection. Thus, any existing techniques (e.g. pretenuring, older-first, Beltway) can be easily integrated into our scheme to further improve the performance. For example, we can have multiple belt 0s to manage clusters of objects with different lifespans. Each of these belts can be properly sized to allow just enough time for objects in the

[5] See http://java.sun.com/docs/performance/appserver/AppServerPerfFaq.html for tuning suggestions from Sun and http://www-03.ibm.com/servers/eserver/iseries/perfmgmt/pdf/tuninggc.pdf for tuning suggestions from IBM.

older increments to die. Pretenuring can also be applied to each sub-nursery to further improve minor collection efficiency. In addition, studies have shown that concurrent and incremental extensions can greatly improve the performance of GenMS. We foresee that if such extensions are applied to our technique, a significant performance improvement can also be expected.

8 Related Work

The main inspiration for our work is based on the concept of *Key objects* [9]. Hayes proposes the key object opportunism approach to manage longer-lived objects in a "keyed-area" [9]. This approach is based on the observation that large clusters of objects are usually allocated at the same time and also tend to die together. The main idea is to select representatives (or key objects) from the cluster and examine the reachability of these key objects more frequently than the rest of the cluster. This approach only applies when key objects exist, and they can be easily detected.

In this work, our key objects are the *remotable* objects, and any objects connected to these remotable objects (i.e. remote objects) are assumed to have similar lifespans. We can make such an assumption because the results from previous work have shown that objects connected together tend to die together [23]. To detect and segregate these remote objects, there are several options. One possible way is to use techniques such as object *colocation optimization* [24] to provide the necessary compile-time analysis to detect remote objects and the runtime component to segregate these remote objects from the local objects. We believe that the colocation technique would have worked well if these remote objects were difficult to be heuristically detected. However, this is not the case as remote objects can be easily detected by monitoring calls to remotable objects. Thus, we choose a dynamic detection technique because it can accomplish our goal at low cost. In terms of object segregation, our technique virtually accomplishes the same goal as their special allocator called *coalloc*.

In addition to the colocation technique, there are at least two additional techniques to improve the efficiency of generational GC. The first technique is *pretenuring*. The basic idea is to identify long-lived objects and create them directly in the mature generation. The goal of this technique is to reduce the promotion cost, thus reducing the GC time and improving the overall performance. Blackburn et al. [11] use a profile-based approach to select objects for pretenuring. They report a reduction in GC time of up to 32% and an improvement in the execution time by 7%. They also report a slight increase in the heap usage with pretenuring. Harris [12] uses dynamic sampling based on overflow and size to predict long-lived objects. Subsequent work to further optimize pretenuring include dynamic object sampling [38] and class based lifespan prediction [39].

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 [40]. Stefanović *et.al.* [13] implements the *older-first* garbage collector that prioritizes collection of older objects to give young objects more time to die. This technique evolves to become a major part of the *Beltway* framework, introduced by Blackburn *et al.* [1]. 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 [1]. All objects are allocated into the belt 0 (can be viewed as similar to the nursery). Beltway framework uses the older-first approach to collect the oldest increment of a belt first. All survivors are promoted to the last increment of the next higher belt. The results of their experiment show an average of 5% to 10% improvement in execution time and 35% improvement under tight heaps.

Compared to our technique, it is unclear how pretenuring and the Beltway framework would handle the lifespan characteristic of objects in application servers. If the decision is to pretenure any longer living 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, 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 the 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, the short-lived objects are still not collected promptly.

On the other hand, our approach invokes minor collection very frequently to quickly reclaim objects; each of our minor invocations also yields good GC efficiency. However, a major short-coming of our technique is that it does not work if objects can be easily segregated into short-lived/long-lived taxonomy. While the argument can be made for a very fine-grained segregation policy (e.g. consider segregating objects with slight differences in lifespans), the dynamic segregation overhead may offset the small benefit that can be gained. However, if clusters of objects with different lifespans can be identified, both pretenuring and the Beltway framework can be applied to further optimize our technique.

The idea of allocating objects exhibiting similar run-time behaviors into their own area is not new. Standard ML of New Jersey has been using up to 14 generations in addition to a *shared* nursery space to achieve good GC performance [41]. Each older generation consists of four arenas; each arena is used to manage a different class of objects (i.e. code objects, arrays, strings, and pairs) with different lifespans and object organizations (containing pointers vs not containing pointers). Our technique differs from this technique in several ways. First, we create two arenas in the nursery. In effect, we attempt to segregate objects at birth to improve minor collection performance. Their scheme segregates objects in the older generation to improve the full collection performance. Second, object segregation in our approach must be determined at allocation time based on the state of allocating threads (serving remote or local requests). Their technique segregates objects at GC time based on object types.

When Steensgaard introduces the thread local heaps approach [29], he also suggests that heap utilization can be improved by grouping threads that share data structures into their own sub-heap [42]. By creating a separate nursery for remote objects, our technique, in effect group threads that access remote objects into their own sub-nursery. This is somewhat similar to the suggestion by Steensgaard except that we do not create a sub-heap that includes both the nursery and the mature space. One reason for such a difference is because our optimization technique is introduced to allow the generational strategy to efficiently manage objects with diverse lifespans while Steensgaard's

technique is designed to improve the allocation and garbage collection parallelism in multithreaded environments.

Recent studies have shown that once the heap size is larger than the physical memory, paging overheads can dominate the execution time and may even result in thrashing [11, 43, 8, 44]. Recent efforts have concentrated on dynamic sizing of the heap to maximize the performance of the existing GC techniques while minimizing paging [43, 8, 44, 45]. While these solutions have shown to work well, they all accept the fact that generational GC is memory inefficient, and thereby, assume that 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 as well as causing the heap size to be larger than the available physical memory. Nevertheless, these techniques can easily support our collector to further improve the GC performance.

9 Conclusion

In this paper, we introduce a new generational collector called AS-GC that takes advantage of an intrinsic behavior of many application servers in which *remotable* objects are commonly used as gateways for client requests. Objects instantiated as part of these requests (*remote objects*) tend to live longer than the remaining objects (*local objects*). This insight is used to create these two types of objects in two optimally sized nurseries. In doing so, the minor collection can be invoked more frequently and efficiently without increasing the heap requirement.

We have implemented the proposed AS-GC and evaluated its performance in an application server setting. We discovered that our proposed scheme can reduce the allocation interferences due to multithreading; a major reason that causes the inefficacy of single-nursery generational collectors. The experimental results show that our collector reduces the frequency of full collection invocations, paging effort, average pause time, and overall garbage collection time. As a result, our collector can yield a 14% increase in the maximum throughput and handle a 10% higher workload.

Acknowledgments

This work was sponsored in part by the National Science Foundation through award CNS-0411043 and by the Army Research Office through DURIP award W911NF-04-1-0104. We thank Mulyadi Oey and Sebastian Elbaum for their contributions in formulating the remote/local objects notion. We are grateful to Matthew Dwyer and Myra Cohen for their valuable feedback on this work. We also thank the anonymous reviewers for providing insightful comments for the final version of this paper.

References

1. Blackburn, S.M., Jones, R.E., McKinley, K.S., Moss, J.E.B.: Beltway: Getting Around Garbage Collection Gridlock. In: Proceedings of the Programming Languages Design and Implementation, Berlin, Germany, pp. 153–164 (2002)

2. Jones, R., Lins, R.: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley and Sons, Chichester (1998)
3. Lieberman, H., Hewitt, C.: A Real-Time Garbage Collector Based on the Lifetimes of Objects. Communications of the ACM 26(6), 419–429 (1983)
4. Ungar, D.: Generational Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. In: ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pp. 157–167. ACM Press, New York (1984)
5. Srisa-an, W., Oey, M., Elbaum, S.: Garbage Collection in the Presence of Remote Objects: An Empirical Study. In: Proceedings of the International Symposium on Distributed Objects and Applications, Agia Napa, Cyprus, pp. 1065–1082 (2005)
6. Xian, F., Srisa-an, W., Jiang, H.: Investigating the Throughput Degradation Behavior of Java Application Servers: A View from Inside the Virtual Machine. In: Proceedings of the 4th International Conference on Principles and Practices of Programming in Java, Mannheim, pp. 40–49 (2006)
7. Hertz, M., Berger, E.: Quantifying the Performance of Garbage Collection vs. Explicit Memory Management. In: OOPSLA '05: 20th annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, San Diego, CA, USA, pp. 313–326 (2005)
8. Yang, T., Berger, E.D., Kaplan, S.F., Moss, J.E.B.: CRAMM: Virtual Memory Support for Garbage-Collected Applications. In: OSDI'06: Proceedings of the USENIX Conference on Operating System Design and Implementation, Seattle, WA (2006)
9. Hayes, B.: Using Key Object Opportunism to Collect Old Objects. In: OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications, Phoenix, AR, pp. 33–46 (1991)
10. IDC: Web services to reach $21 billion by 2007. On-line Article (2003), http://thewhir.com/marketwatch/idc020503.cfm
11. Blackburn, S.M., Singhai, S., Hertz, M., McKinely, K.S., Moss, J.E.B.: Pretenuring for Java. In: Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications, Tampa Bay, FL, pp. 342–352 (2001)
12. Harris, T.L.: Dynamic Adaptive Pretenuring. In: Proceedings of International Symposium on Memory Management, Minneapolis, Minnesota, United States, pp. 127–136 (2000)
13. Stefanović, D., McKinley, K.S., Moss, J.E.B.: Age-Based Garbage Collection. In: OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications, Colorado, United States pp. 370–381(1999)
14. Wilson, P.R., Johnstone, M.S., Neely, M., Boles, D.: Dynamic Storage Allocation: A Survey and Critical Review. In: IWMM '95: Proceedings of the International Workshop on Memory Management, London, UK, pp. 1–116. Springer, Heidelberg (1995)
15. Welsh, M., Culler, D.E., Brewer, E.A.: SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In: Symposium on Operating Systems Principles, pp. 230–243 (2001)
16. Hibino, H., Kourai, K., Shiba, S.: Difference of Degradation Schemes among Operating Systems: Experimental Analysis for Web Application Servers. In: Workshop on Dependable Software, Tools and Methods, Yokohama, Japan (2005)
17. Netcraft: Video iPod Launch Slows Apple Store (2005), http://news.netcraft.com/archives/2005/10/12/video_ipod_launch_slows_apple_store.html
18. Chosun Ilbo: Cyber Crime Behind College Application Server Crash. On-line article (2006), http://english.chosun.com/w21data/html/news/200602/200602100025.html
19. Standard Performance Evaluation Corporation: Spec jvm98 benchmarks (Last Retrieved: June 2005), http://www.spec.org/osg/jvm98
20. Standard Performance Evaluation Corporation: SPECjbb2000, WhitePaper (2000), http://www.spec.org/osg/jbb2000/docs/whitepaper.html

21. Standard Performance Evaluation Corporation: SPECjAppServer2004 User's Guide. On-Line User's Guide (2004), http://www.spec.org/osg/jAppServer2004/docs/UserGuide.html
22. Hirzel, M., Henkel, J., Diwan, A., Hind, M.: Understanding the Connectivity of Heap Objects. In: ISMM '02: Proceedings of the 3rd International Symposium on Memory Management, Berlin, Germany, pp. 36–49 (2002)
23. Hirzel, M., Diwan, A., Hertz, M.: Connectivity-Based Garbage Collection. In: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 359–373. ACM Press, New York (2003)
24. Guyer, S.Z., McKinley, K.S.: Finding your Cronies: Static Analysis for Dynamic Object Colocation. In: OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, BC, Canada, pp. 237–250. ACM Press, New York (2004)
25. Sun: Performance Documentation for the Java HotSpot VM. On-Line Documentation (Last Retrieved: June 2005), http://java.sun.com/docs/hotspot/
26. JBoss:Jboss Application Server. Product Literature (2005), http://www.jboss.org/products/jbossas
27. Sachindran, N., Moss, J.E.B.: Mark-copy: Fast Copying GC with Less Space Overhead. SIGPLAN Notices 38(11), 326–343 (2003)
28. Domani, T., Goldshtein, G., Kolodner, E.K., Lewis, E., Petrank, E., Sheinwald, D.: Thread-Local Heaps for Java. SIGPLAN Not. 38(suppl. 2), 76–87 (2003)
29. Steensgaard, B.: Thread-Specific Heaps for Multi-Threaded Programs. In: ISMM '00: Proceedings of the 2nd International Symposium on Memory Management, Minneapolis, Minnesota, United States, pp. 18–24 (2000)
30. Larson, P., Krishnan, M.: Memory Allocation for Long-Running Server Applications. In: ISMM '98: Proceedings of the 1st International Symposium on Memory Management, Vancouver, British Columbia, Canada, pp. 176–185 (1998)
31. Gay, D., Aiken, A.: Memory Management with Explicit Regions. In: PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, Montreal, Quebec, Canada, pp. 313–323. ACM Press, New York (1998)
32. Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., Cheney, J.: Region-Based Memory Management in Cyclone. In: PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin, Germany, pp. 282–293. ACM Press, New York (2002)
33. Stoutamire, D.P.: Portable, Modular Expression of Locality. PhD thesis, University of California-Berkeley, Chair-Jerome A. Feldman (1997)
34. Tofte, M., Talpin, J.P.: Region-Based Memory Management. Information and Computation 132(2), 109–176 (1997)
35. Bollella, G., Gosling, J.: The Real-Time Specification for Java. Computer 33(6), 47–54 (2000)
36. Elsman, M.: Garbage Collection Safety for Region-Based Memory Management. In: TLDI '03: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, New Orleans, Louisiana, USA, pp. 123–134. ACM Press, New York (2003)
37. Standard Performance Evaluation Corporation: SPECjAppServer2002 User's Guide. On-Line User's Guide (2002), http://www.spec.org/osg/jAppServer2002/docs/UserGuide.html
38. Jump, M., Blackburn, S.M., McKinley, K.S.: Dynamic Object Sampling for Pretenuring. In: ISMM '04: Proceedings of the 4th International Symposium on Memory Management, Vancouver, BC, Canada, pp. 152–162 (2004)

39. Huang, W., Srisa-an, W., Chang, J.: Dynamic Pretenuring for Java. In: International Symposium on Performance Analysis of Systems and Software ISPASS, March 10-13, 2004, Austin, TX, pp. 133–140 (2004)
40. Clinger, W.D., Hansen, L.T.: Generational Garbage Collection and the Radioactive Decay Model. In: PLDI '97: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, Las Vegas, Nevada, United States, pp. 97–108. ACM Press, New York (1997)
41. Reppy, J.H.: A High-Performance Garbage Collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ (1993)
42. Cohen, M., Kooi, S.B., Srisa-an, W.: Clustering the Heap in Multi-Threaded Applications for Improved Garbage Collection. In: GECCO '06: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, Seattle, Washington, USA, pp. 1901–1908 (2006)
43. Hertz, M., Feng, Y., Berger, E.D.: Garbage Collection Without Paging. In: PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, Chicago, IL, USA, pp. 143–153. ACM Press, New York (2005)
44. Yang, T., Hertz, M., Berger, E.D., Kaplan, S.F., Moss, J.E.B.: Automatic Heap Sizing: Taking Real Memory into Account. In: Proceedings of the International Symposium on Memory Management, Vancouver, BC, Canada, pp. 61–72 (2004)
45. Zhang, C., Kelsey, K., Shen, X., Ding, C., Hertz, M., Ogihara, M.: Program-Level Adaptive Memory Management. In: International Symposium on Memory Management, Ottawa, Canada, pp. 174–183 (2006)