8-31-2008

NMFLUX: Improving Degradation Behavior of Server Applications through Dynamic Nursery Resizing

Witawas Srisaan
University of Nebraska-Lincoln, witty@cse.unl.edu

Cheng Huan Jia
University of Nebraska-Lincoln, cjia@cse.unl.edu

University of Nebraska–Lincoln, Computer Science and Engineering
Technical Report TR-UNL-CSE-2008-0008
Issued August 31, 2008

NMFLUX: Improving Degradation Behavior of Server Applications through Dynamic Nursery Resizing

Witawas Srisa-an and ChengHuan Jia
Department of Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115
{witty,cjia}@cse.unl.edu

ABSTRACT

Currently, most generational collectors are tuned to either deliver peak performance when the heap is plentiful, but yield unacceptable performance when the heap is tight or maintain good degradation behavior when the heap is tight, but deliver sub-optimal performance when the heap is plentiful. In this paper, we present NM-FLUX (*continuously varying* the *Nursery/Mature* ratio), a framework that switches between using a fixed-nursery generational collector and a variable-nursery collector to achieve the best of both worlds; i.e. our framework delivers optimal performance under normal workload, and graceful performance degradation under heavy workload. We use this framework to create two generational garbage collectors and evaluate their performances in both desktop and server settings. The experimental results show that our proposed collectors can significantly improve the throughput degradation behavior of large servers while maintaining similar peak performance to the optimally configured fixed-ratio collector.

1. INTRODUCTION

Garbage collection (GC) is a process to automatically reclaim dynamically allocated memory. It has been adopted as a language feature in many modern object-oriented languages including Java, C#, and Visual Basic .NET. With garbage collection, programmers are relieved from the burden of explicitly managing memory, a task that has proved to be tedious and error prone. As of now, the most adopted GC strategy is generational garbage collection.

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 [24]. Because the nursery is usually configured to be much smaller than the mature space (an area to host surviving objects from the nursery), generational collectors often yield shorter GC pauses than most other GC strategies. The two common ways to set the size of the nursery are to use fixed nursery/mature ratio throughout execution (e.g. HotSpot generational collector [5]) or varying the nursery size based on the amount available memory after each collection (e.g. the Appel generational collector [1] in Jikes RVM [9]).

In applications that demand a large volume of dynamic memory, it is attractive to use languages with garbage collection as the development platform. One example of such memory intensive applications is application servers. An application server is a software system that delivers "applications to client computers". It also handles most business logics and data accesses for the applications it manages[1]. The leading technology used to develop application servers is Java Platform Enterprise Edition or JEE (formerly known as J2EE) from Sun Microsystems. Many commercial and open-source implementations of the JEE platform include IBM WebSphere [10], JOnAS [17], and JBoss [11].

Two common characteristics of application servers are that they are long-running, and their service demands can vary significantly. Interestingly, the periods of higher demands often "coincide with the times when the service has the most value" [27]. Thus, it is crucial for these servers to be able to face unexpected heavy demands without failing or yielding unacceptable performances. However, a study by Xian *et al.* [29] has shown that the throughput performances of these application servers degrade ungracefully and the root cause for such poor degradation behavior is *garbage collection.*

This Work. To date, most investigations of GC performances in application servers have been done using fixed-ratio generational collectors [8, 21, 29, 28]. As the first step, we implemented an Appel collector into HotSpot, our experimental Java virtual machine (JVM) platform from Sun Microsystems. We then investigated its performance in an application server setting using SPECjbb2000. We found the Appel collector to deliver more graceful degradation behavior, but much lower average throughput performance than the fixed-ratio collector. We then investigated the main reasons that cause the throughput performance of the fixed ratio collector to degrade so poorly. Our investigation yielded the following conclusions:

1. *Longer-living objects.* A study by Xian *et al.* has shown that as the demands become heavier, objects also tend to live longer due to higher degree of concurrency [28]. In large servers, higher demands often mean that a larger number of threads compete for the CPUs, and therefore, each thread makes less execution progress in a given amount of time. Thus, objects are not used in a timely fashion and remain reachable in the heap for a longer period of time.

2. *Inefficient memory usage.* A study by Hertz and Berger finds that an application using garbage collection often requires three to five times more memory than a similar application

[1]Definition of Application Servers from WikiPedia, http://en.wikipedia.org/wiki/Application_server

using explicit memory management [6]. This additional memory is used to extend GC intervals so that objects have more time to die. As the demands become heavier, the heap is filled up much quicker, resulting in higher frequency of GC invocations. However, because objects are now longer living, these GC invocations are not effective.

3. *Frequent invocations of full collection.* Most generational collectors utilize *copy-reserve* space, a space equaled to the size of the nursery located in the mature generation, to ensure that nursery collection (or minor collection) can complete successfully. If the amount of copy-reserve is smaller than the nursery, there is a chance that minor collection will fail due to a larger volume of surviving objects than the size of copy-reserve. When such condition occurs, full heap collection is invoked instead of minor collection. We discovered that under heavy workload, this condition occurs repeatedly, leading to many consecutive full collection invocations.

We leveraged these insights to construct *NMFLUX*, a framework that dynamically selects when to use fixed-ratio collector and when to use variable-ratio collector. NMFLUX monitors the GC behavior to detect instances when the JVM invokes full collection consecutively due to the copy-reserve space being too small. When such an instance is detected, the nursery/remote ratio is reduced.

We utilized this framework to create two variations of generational collector: the *dynamic-ratio* collector and the *hybrid* collector. Our dynamic-ratio collector initially sets the nursery/remote ratio to be the value that yields the optimal performance. Once an instance of two consecutive full collection invocations is encountered, our dynamic-ratio collector reduces the nursery/mature ratio from *1:m* to *1:m+1*, where *m* indicates the number of times the mature space is larger than the nursery. In this technique, the collector has full control of the nursery size, as it remains fixed until the next instance of back-to-back full collection.

Our hybrid collector is also similar to the dynamic-ratio collector except that it switches to an Appel collector when full collection is invoked consecutively. Thus, the collector does not have the full control of the nursery size because the Appel collector automatically adjusts the size based on the available amount of heap memory. In effect, the goal of these two techniques is to facilitate more minor collection invocations when the demand is high, and thus, improving the throughput performance at this critical execution region. It is worth noting that both collectors can switch back to the optimal fixed-ratio once the demands become lighter.

The remainder of this paper is organized as follows. Section 2 provides information pertinent to this work. Section 3 reports the results of our investigation of the throughput degradation behavior between fixed-ratio collectors and our Appel collector. Section 4 details the design of NMFLUX and the dynamic-ratio and the hybrid collectors. Section 5 evaluates the effectiveness of our proposed schemes. Section 6 briefly discusses some of the existing related research efforts. Section 7 discusses future work, and the last section concludes this paper.

2. BACKGROUND

One of our proposed algorithms is a combination of the fixed-ratio generational collector and our implementation of an Appel collector in HotSpot. This section outlines the HotSpot collector [5] and the basics of an Appel collector [1].

2.1 Generational Collector in HotSpot

The HotSpot VM partitions the heap into three major generations: nursery, mature, and permanent. The nursery is further partitioned into three areas: *eden* and two survivor spaces, *from* and *to*, which collectively account for a minimum of 20% of the nursery (i.e. the ratio of the eden to the survivor spaces is 4:1). There is also a requirement that each of the two survivor spaces be larger than 64 KB. Users can set the size of the nursery using a command-line argument that specifies the ratio between the nursery and the mature space (e.g. the ratio of 1/3 nursery and 2/3 mature or 1:2 is used as the default ratio for systems using AMD 64 processors). Once set, the ratio stays fixed throughout an execution. 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.

HotSpot uses copying collector to collect the nursery (*minor collection*) and mark-compact to collect the entire heap (*full collection*). 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 [22], and it is used as an aging area for longer living objects to die within the nursery. It is worth noting that the aging area is only effective when the number of copied objects from the eden and the *from* spaces are small. If the number of surviving objects become too large (such as in application servers), most of these objects are promoted directly to the mature generation, leading to more frequent full collection invocations.

Similar to most copying-based collector, HotSpot uses *copy-reserve* space to ensure that the amount of available memory in the mature generation is large enough to accommodate surviving objects from minor collection. It is possible that all objects in the nursery survive minor collection and thus, the size of the copy-reserve space is usually set to be the same as the size of the nursery. When the amount of the copy-reserve space is less than the nursery, full collection based on mark-compact algorithm is invoked. We refer to the default collector in HotSpot as *fixed-ratio* throughout the paper.

The full collector in HotSpot performs garbage collection in four phases: marking, precompaction, adjusting pointers, and compaction. The marking phase goes through the root sets and marks all reachable objects. To avoid deep recursion, a marking stack is used [12]. The precompaction phase calculates a new target address for each object after compaction and encodes the address into the object. The next phase updates any references to an object to the new target address. This is done by simply reading the value encoded in the object as part of the precompaction phase [13]. The last phase slides objects toward the lowest address of the mature space.

2.2 Appel Collector

Similar to the collector in HotSpot, Appel collector partitions the heap into nursery and mature generations. However, the nursery size is variable depending on the object occupancy in the mature space. If copying is used to collect the nursery, a copy-reserve space is also used to ensure a successful completion of minor collection. An Appel collector adjusts the nursery size after each minor collection. Initially, the nursery, n, occupies half of the heap and copy-reserve space, cr, occupies the other half ($n = cr = \frac{heap}{2}$). When the nursery is full, the surviving objects, m, are copied to the copy-reserve space. Once done, the nursery occupies half of the available space in the heap, and the copy-reserve occupies the other half ($n = cr = \frac{heap-m}{2}$). This nursery resizing process repeats until a certain size threshold is reached. At that point, full collection is invoked. To date, semi-space copying [1, 9], mark-sweep [9], and mark-compact [14] have been used to perform full collection in Appel collectors.

Because both Appel and fixed-ratio collectors are widely used, it is worth to point out some advantages and disadvantages of each approach. One argument for using the fixed-ratio collectors is that the nursery size can be tuned to match the available memory in a system and a lifespan characteristic of an application. For example, the size of the nursery can be tuned to make sure that by the time the nursery is exhausted, most of the objects allocated up to that point have already died. Thus, in desktop-like applications where lifespan characteristics are more predictable, the fixed-ratio collectors can perform very well [5, 15].

On the other hand, the nursery size varies in Appel collectors. Once the nursery size becomes too small to allow enough time for newly created objects to die, these objects are promoted, causing higher minor collection overhead. Thus, given an application with a uniform lifespan characteristic, an Appel collector may not perform as well as a fixed-ratio collector.

In applications with variable and unpredictable lifespan characteristics such as application servers, a fixed ratio collector may not perform efficiently when the lifespan characteristic is different than the expected characteristic. As will be shown in the next section, these collectors may forgo minor collection entirely when facing heavy demands. On the other hand, Appel collectors are more tolerant to changes in characteristic as the nursery is dynamically sized based on the available memory. Thus, minor collection continues to be invoked as long as there is available memory in the heap.

3. MOTIVATION

A study of SPECjAppServer2004 by Xian *et al.* has shown that a 20% increase in workload can result in a 75% decrease in the throughput performance [29]. Such degradation behavior is considered ungraceful as it can lead to non-uniform responses and unstable system performances. The study further identifies garbage collection as the root cause of the problem. The experimental results indicate that at the heaviest workload, full collection can spend over five minutes to complete its task, preventing the application from making any execution progress.

The major reason for such a long collection time is because the number of full collection invocations increases disproportionally to the number of minor collection invocations at heavy workload. Figure 1 illustrates this scenario in SPECjbb2000. Notice that at the beginning of the execution, the majority of GC invocations are minor collection (indicated by gray bars). As the execution progresses toward termination, most of the GC invocations become full collection (indicated by black bar). The change becomes drastic at 8-warehouse workload.

Because the study by Xian *et al.* [29] was conducted using only a fixed-ratio collector tuned to yield the best throughput performance, we further investigated if similar throughput degradation behavior is experienced when different nursery/mature ratios are used and when an Appel collector is used. In the following sections, we outline:

1. The steps necessary to make HotSpot support Appel collector.

2. The throughput performances when different ratios are used in the fixed-ratio collector.

3. The throughput performances of two fixed-ratio collectors (one yielding the highest throughput and the other yielding the best degradation behavior) and the Appel collector.

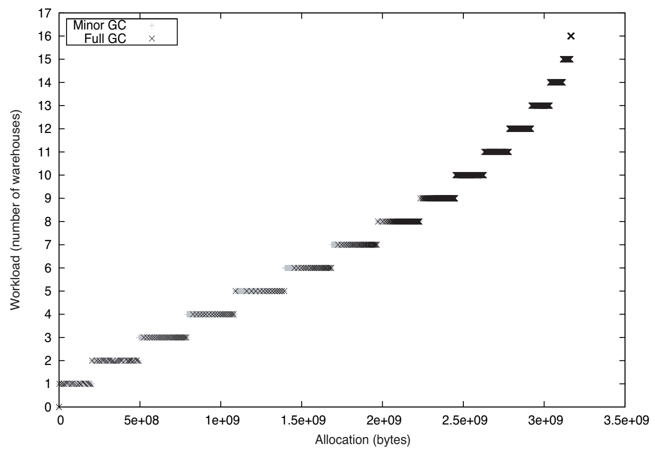


Figure 1: Distribution of minor and full collection invocations over execution time (SPECjbb2000)

3.1 Modifying Heap Layout

The heap layout (detailed in Figure 2a) adopted in HotSpot does not allow dynamic resizing of generations during execution. This is because the starting address of the nursery is fixed, and the compaction process during each full collection slides the objects toward the lowest address of the mature space. Thus, there is no room to adjust the boundary between the nursery and the mature generation. To support our proposed collectors, we reorganized the heap so that the mature space starts at the lowest address of the heap. In this layout, the compaction process slides objects toward the lowest address of the heap, leaving unused memory at the top (higher-address) of the mature space. After each minor collection, the eden space is also empty, allowing straightforward adjustment of the nursery size.

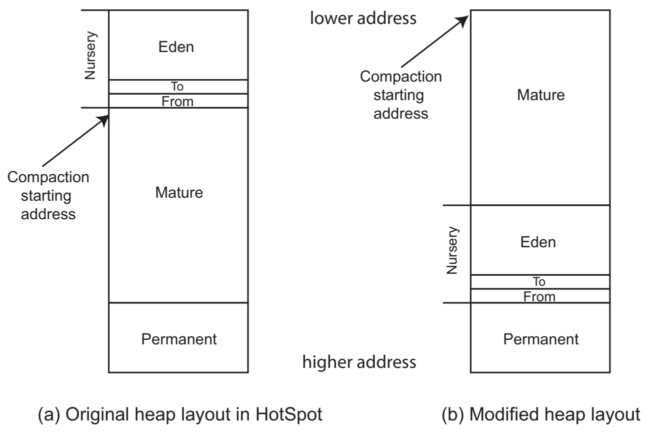


Figure 2: Original vs. modified heap layouts

To confirm that our reorganization has minimal effects on performance, we compared the throughput performances of SPECjbb2000 when the original layout and the modified layout were used. The result is depicted in Figure 3. Notice that the performances were nearly identical throughout the execution. Base on this result, we concluded that our modified heap layout does not affect the overall performance and can be used as the foundation to support the implementation of an Appel collector.

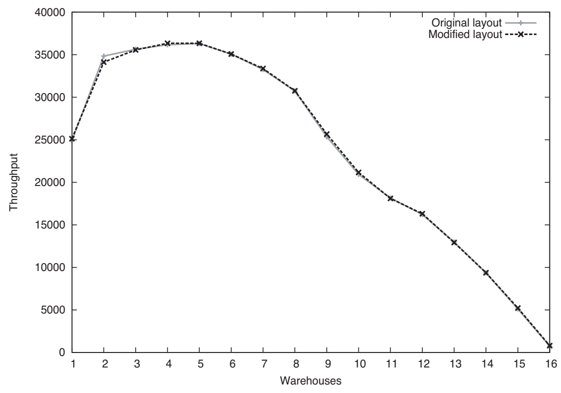


Figure 3: Comparing the throughput performance using the original and the modify heap layouts (SPECjbb2000)

3.2 Implementing Appel Collector

The modified heap layout allows us to adjust the nursery size freely. Thus, it is possible to implement an Appel collector using the modified HotSpot. Initially, the heap space, not including the permanent space, is divided into two equal portions, nursery and copy-reserve. Once the nursery is full, minor collection is called, and surviving objects are moved to the copy-reserve space. Once the minor collector finishes, the occupied space (64 KB alignment) is considered as the mature space. The remaining space is then split in half, one for the nursery and the other for the copy-reserve.

It is worth noting that there are numerous efforts to reduce the size of copy-reserve space [14, 26, 25, 19]. (Brief descriptions are provided in Section 6.) However, we chose not to use any of these techniques for two reasons: First, we want to fairly compare the performance of a basic Appel collector to that of a fixed-ratio collector. Second, most of the optimizations leverage an insight, based on studies of desktop applications, that only a small number of objects survive minor collection, thus, the copy-reserve size can be reduced. Our previous study of application servers show that there are times that most, if not all, objects survive minor collections [29, 28]. Thus, these optimizations may not work efficiently in our experimental settings.

When the available memory is smaller than 256 KB, full collection is invoked. This is because the minimum requirement for the two survivor spaces is 64 KB each. Thus, if the nursery is set to 128 KB, there is not enough memory to create the eden space. Also note that the only time that full collection is invoked back-to-back is when the amount of available memory is fewer than 256 KB after a full collection invocation.

As stated in Section 2, the default minor collector also leaves the *to* space partially occupied after each minor collection invocation, making resizing more difficult. To overcome this challenge, we modified the minor collector to copy all surviving objects from the nursery as well as the *from* space directly to the mature space. In other words, the nursery is completely empty after each minor collection to allow resizing of the nursery.

3.3 Experimental Environment

We conducted our experiment on an AMD Opteron system with two 2 GHz processors. The system has 4 GB of physical memory. We used our modified HotSpot with the new heap layout. It also supports both the fixed-ratio and the Appel collector. For our benchmark, we used SPECjbb2000 that was configured to go from one warehouse to sixteen warehouses in increments of one warehouse. We set the maximum heap size to 308 MB, which was the same as the maximum live-size. We used 308 MB to provide plenty of heap space when the memory demand was light; thus, garbage collection should perform efficiently. However, when the memory demands became heavy, the heap would still be large enough for SPECjbb2000 to execute, but without the necessary space to allow for efficient GC. Thus, this setting should closely emulate a server application facing unexpected demands. Also note that our maximum heap size (308 MB) was much smaller than our physical memory capacity (4 GB), meaning that paging did not affect the throughput performance. For the fixed-ratio approach, the nursery/mature ratio was set to 1:2 (the nursery occupies 33% of the heap), which yielded the highest throughput.

3.4 Comparing Throughput Performances

Figure 4 illustrates the differences in throughput performances due to different nursery/mature ratios. Notice that ratio 1:2 yields the best throughput performance until the number of warehouses is eight (we refer to this workload level as a *critical point*). After that, ratios with smaller nursery outperform ratio 1:2. Also notice that ratio 1:10 yields the best throughput once the number of warehouses is beyond 10. At 11-warehouse, the ratio 1:10 yields about 26% higher throughput than ratio 1:2.

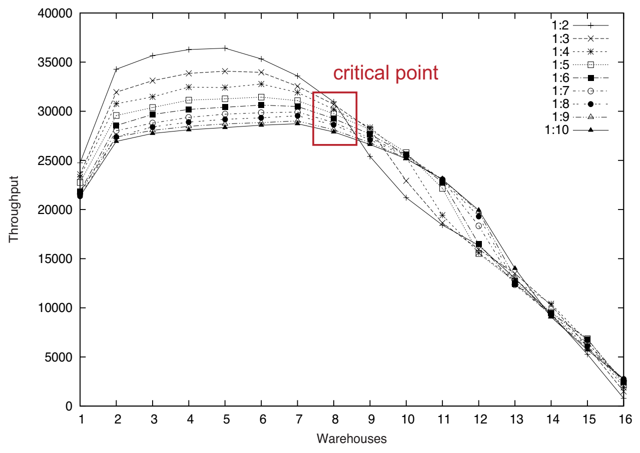


Figure 4: Throughput performances using different nursery/mature ratios in SPECjbb2000

Figure 5 depicts the throughput performances of two fixed-ratio collectors (1:2-collector and 1:10-collector) and the Appel collector. Our study of SPECjbb2000 revealed that the 1:2-collector mostly outperformed the Appel collector when the workload was less than 8 warehouses. However, once the workload level surpassed 8 warehouses, the Appel collector yielded much higher throughput performance than both of the fixed-ratio collectors, leading to more graceful degradation behavior.

As stated in Section 2, one major benefit of the fixed-ratio approach is that the size of the nursery can be optimally tuned to match the lifespan characteristic of an application. However, in applications with varying lifespan characteristic (e.g. application servers), the ratio becomes sub-optimal as soon as the lifespan characteristic begins to change. Because the optimal ratio in our experiment was 1:2, the nursery occupied a large portion of the heap. Therefore, it became exceedingly difficult to maintain a large

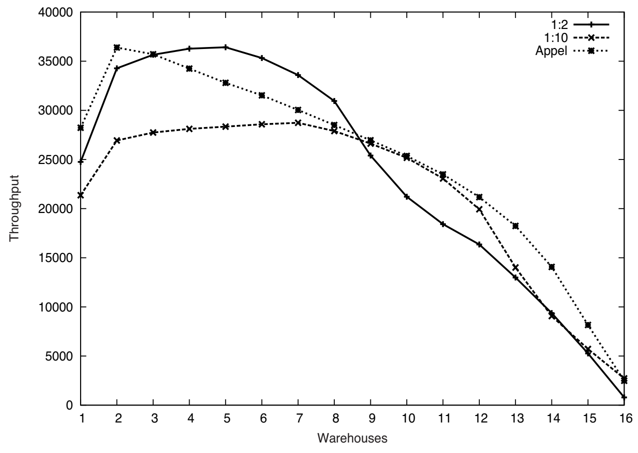


Figure 5: Throughput performances of 1:2-collector, 1:10-collecto, and Appel collector (SPECjbb2000)

enough copy-reserve space to allow successful minor collection invocations.

On the other hand, it is more difficult to tune the nursery size of the Appel collector because the size is determined dynamically by the amount of available heap memory. Thus, when the demand is light, the nursery is often too small to give enough time for objects to die, leading to higher collection time due to larger volumes of surviving objects. However, once the workload becomes heavy, the Appel collector rarely suffers from the problem of copy-reserve space being too small. Thus, minor collection is still invoked under heavy workload, leading to shorter pauses and more time for mutator execution.

Remark. The result of our experiment clearly indicates that techniques not suffering from consecutive full collection invocations during heavy demands will allow the throughput to degrade more gracefully. In the next section, we will propose two techniques that leverage the fixed-ratio approach to achieve high throughput performance when the workload is light, but avoid suffering from repetitive full collection invocations when the workload is heavy.

4. INTRODUCING NMFLUX

As stated in the last section, one approach to make the throughput performance of a fixed-ratio collector degrade more gracefully is to prevent consecutive full collection invocations during heavy workload. NMFLUX is created to accomplish this specific task. The main component of NMFLUX is the decision process to switch from a fixed-ratio collector to a variable-ratio collector when the workload is heavy and then switch back when the workload is light.

Detecting critical point. We experimented with various run-time parameters such as live-size, mature generation usage, and allocation rate only to discover that these parameters indicate application specific behavior and do not always yield accurate predication or representation of critical points. On the other hand, our investigation of SPECjbb2000 showed that two or more consecutive full collection invocations only occur at critical points. (To reiterate, a *critical point* is an execution location where a collector with smaller nursery/mature ratio outperforms a collector with larger ratio.)

NMFLUX leverages this insight to decrease the nursery/mature ratio whenever it detects two consecutive full collection invocations, and increase the ratio when there is enough heap memory to support the optimal fixed-ratio nursery size. To support dynamic nursery enlargement, NMFLUX computes the amount of available memory in the mature space after each full collection invocation. When there is enough copy-reserve memory to support a larger ratio, the system automatically enlarges the nursery to the new ratio.

By how much should we decrease the nursery size? The framework provides the locations to switch from fixed-ratio to variable-ratio collectors and vice-versa. However, the actual implementation of a collector based on this framework must also select the reduced nursery size after a critical point is detected. We implemented two variations of generational collectors that leverage NMFLUX: *dynamic-ratio* and *hybrid*.

4.1 Dynamic-Ratio Collector

As shown in Section 3, smaller ratios allow the throughput performances of SPECjbb2000 to degrade more gracefully than the 1:2 ratio. Thus, one approach is to incrementally reduce the ratio each time a critical point is reached. Our dynamic-ratio collector reduces the size of the heap from 1:m to 1:m+1 (where m indicates the number of times the mature space is larger than the nursery space) each time a critical point is detected. Note that the reduction can be made until the ratio is 1:15, the minimum ratio allowed by HotSpot.

Instead of relying on NMFLUX to provide the switching points, it is also possible to use different criteria (e.g. number of threads, allocation rate, etc.) to trigger nursery reduction. However, the following constraints must be followed:

- *Nursery reduction without maintaining to/from spaces ratio.* As stated earlier, HotSpot set the combined size of the to/from spaces to minimally be 20% of the nursery. If this ratio does not have to be maintained, the new ratio can be applied after each *minor collection* as the *eden* space is empty. However, the new ratio must result in a nursery that is larger than the 128KB due to the minimum size requirement of the *to* and *from* spaces.

- *Nursery reduction while maintaining to/from space ratio.* Prior to a full collection invocation, the new boundary address (between nursery and mature spaces) is calculated. Full collection slides objects to the beginning of the new boundary and copies surviving objects in the nursery to the mature space. At that point, the nursery is empty so new to/from spaces can be configured to maintain the 20% ratio. In summary, this type of adjustment can only be done through *full* collection.

4.2 Hybrid Collector

Our collector is initially configured to use the optimal fixed-ratio between the nursery and mature spaces (e.g. 1:2 ratio for SPECjbb2000). However, once a critical point is reached, the system switches to Appel collector. The switch becomes effective after a full collection invocation has completed.

In the next section, we will evaluate the performances of these two collectors by comparing them against the performance of an optimally tuned fixed-ratio collector.

5. EVALUATION

The goal of our collectors is to provide the best of both worlds performance by utilizing fixed-ratio collector when a server application is facing light memory demands and variable-sized-nursery collector when the application is facing heavy memory demands. Thus, our benchmarks must have *varying demands in memory usage* similar to a long-running server application. That is we want

Benchmark	Description	Input configurations	Total allocations		Maximum	Number of
			objects (million)	bytes (MB)	live-size (MB)	threads
xalan (DaCapo)	Transforms XML documents into HTML.	-s default	161	60	26	1
javac (SPECjvm98)	JDK 1.0.2 Java compilers.	problem size = 100	5.9	178	7.2	1
SPECjbb2000-16	A Java program emulating 3-tier sytem focusing on the middle tier.	16 warehouses	788	41000	308	21
SPECjbb2000-32	A Java program emulating 3-tier sytem focusing on the middle tier.	32 warehouses	3573	188000	768	37
SPECjbb2005-16	A Java program emulating 3-tier sytem focusing on the middle tier.	16 warehouses	5757	325000	620	21
SPECjbb2005-32	A Java program emulating 3-tier sytem focusing on the middle tier.	32 warehouses	6002	339000	1200	37

Table 1: Benchmark Characteristics

Benchmark (heap size)	Fixed-ratio		Appel		Dynamic-ratio		Hybrid	
	Minor GCs Calls (seconds)	Full GCs Calls (seconds)	Minor GCs Calls (seconds)	Full GCs Calls (seconds)	Minor GCs Calls (seconds)	Full GCs Calls (seconds)	Minor GCs Calls (seconds)	Full GCs Calls (seconds)
xalan (64MB)	926 (27.82)	243 (27.60)	909 (26.83)	232 (25.24)	959 (28.67)	268 (30.19)	917 (26.81)	236 (25.62)
javac (20MB)	21 (0.33)	46 (3.79)	205 (1.41)	6 (0.17)	75 (0.75)	26 (1.96)	245 (1.62)	10 (0.49)
jbb2000-16 (308MB)	2049 (143.31)	1326 (808.56)	25159 (1002.38)	2 (0.04)	2891 (224.15)	1265 (782.63)	19091 (829.33)	279 (114.72)
jbb2000-32 (768MB)	1602 (242.17)	322 (1146.19)	15429 (1071.39)	2 (0.05)	2120 (343.53)	275 (925.83)	11152 (932.43)	71 (140.99)
jbb2005-16 (620MB)	2530 (226.16)	623 (1220.14)	27259 (1126.76)	2 (0.05)	3571 (329.17)	515 (978.81)	19009 (929.15)	128 (160.71)
jbb2005-32 (1.2GB)	1777 (201.84)	357 (379.61)	8074 (613.10)	2 (0.04)	2010 (265.06)	310 (313.67)	5158 (453.43)	180 (119.33)

Table 2: Comparing GC behaviors when fixed-ratio, Appel, dynamic-ratio, and hybrid collectors are used. Also note that all applications, the optimal ratio is 1:2.

(i) the application to be heap intensive and maintain a large live-size over the entire execution, and (ii) the heap requirement and live-size to increase and decrease over time.

We also experimented with applications that do not meet these criteria. For example, our experiments with the SPECjvm98 suite showed that our collectors did not provide any advantages over the optimally tuned fixed-ratio collector in most applications (the only exception was *javac*). This was because the lifespan characteristics of these applications did not change drastically over time. In fact, there were no instances of back-to-back full collection invocations when the heap was set to be twice the maximum live-size. We also conducted experiments using DaCapo benchmarks [4]. Our results, once again showed that only one benchmark could slightly benefit from our techniques (xalan). It is worth noting that we initially expected *hsqldb* to work well with our technique. However, its optimal nursery:mature ratio was only 1:11, meaning that it already used a very small nursery. Thus, our technique could not provide any performance benefits.

In summary, we included the following benchmarks in our experiments: *xalan*, *javac*, *SPECjbb2000* (16 and 32 warehouses), and *SPECjbb2005* (16 and 32 warehouses). The basic characteristic of our selected benchmarks are given in Table 1.

Our methodology was to execute these applications ten times to monitor various performance metrics (e.g. GC behavior, throughput or execution time, and minimum mutator utilization). The following subsections report the average values of these metrics.

5.1 Basic GC Behavior

It is expected that different collectors yield different garbage collection performances. In terms of throughput performance and pause time, a collector that invokes more frequent minor collection should outperform a collector that invokes more frequent full collection. Thus, the focus of this section is on the differences in the number of minor and full collection invocations and the time spent in each type of collections. Table 2 reports the experimental results.

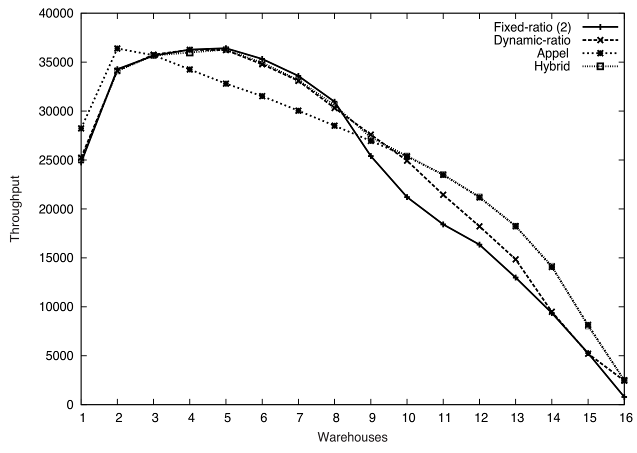
The Appel collector invoked the highest number of minor collection (as many as ten times more than that of the fixed-ratio collector). In doing so, it significantly reduced the number of full collection invocations. However, these changes in the number of minor and full collection invocations did not mean that less time was spent in GC. For example, in SPECjbb2000 with 16 warehouses, the fixed-ratio collector spent about 952 seconds on GC while the Appel collector spent about 1000 seconds. However, the average minor collection time for the Appel collector was much smaller. This was mainly due to smaller nursery sizes.

5.2 Throughput Performance

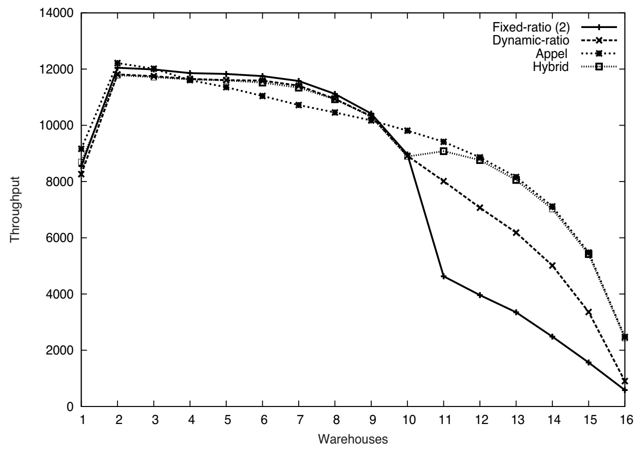
Table 2 shows that different collectors yield different GC performances. However, it is unclear how these differences affect the overall throughput performance and its degradation behavior of each server application. Because the focus of this section is mainly on throughput, we only observed the performances of SPECjbb2000 and SPECjbb2005. Figure 6 illustrates our experimental results.

Notice that the two proposed collectors yielded nearly the same peak throughput performances as the fixed-ratio collector. In addition, the dynamic-ratio collector was able to maintain higher throughput performances during heavy demands by making several nursery reductions. Under heavy workload, the throughput improvements in the 16 warehouses settings were as much as 16% (at 11 warehouses) and 75% (at 12 warehouses) in SPECjbb2000 and SPECjbb2005, respectively. In addition, the hybrid collector was able to maintain nearly the same throughput performances as the Appel collector, with the peak throughput performance improvements of 26% for SPECjbb2000 and 125% for SPECjbb2005, over the fixed-ratio collector.

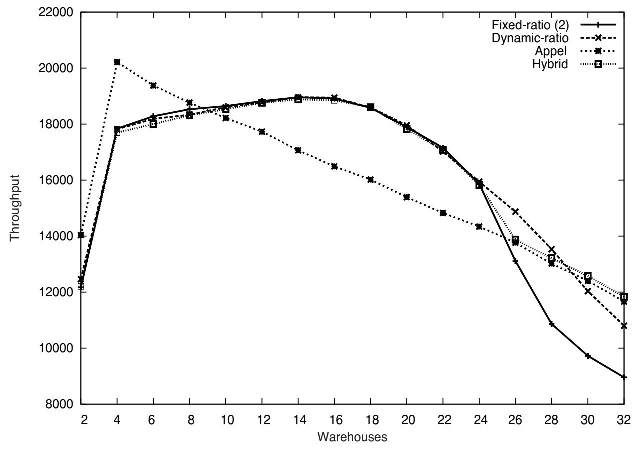
When our collectors were tested under more extreme memory demands (32 warehouses), they performed even better as the differences in throughput performances were higher than those of the 16 warehouses settings. Moreover, the throughput performance of the Appel collector during light to moderate workload was much worse than the other collectors in SPECjbb2000. In this scenario, the throughput performances at heavy workload of SPECjbb2000 were 23% (dynamic-ratio) and 28% (hybrid) higher than that that



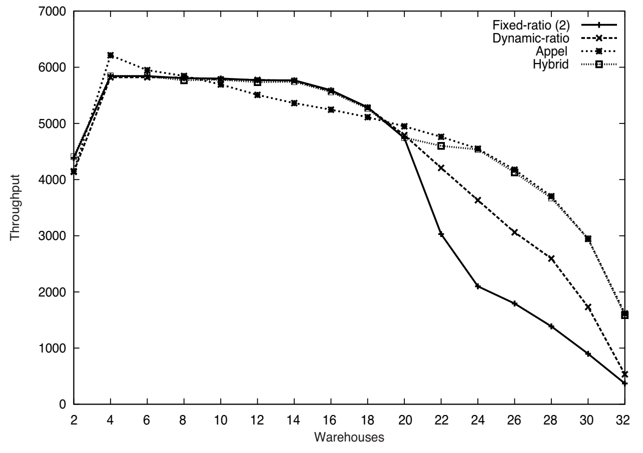
(a) SPECjbb2000 (16 warehouses)



(b) SPECjbb2005 (16 warehouses)



(c) SPECjbb2000 (32 warehouses)



(d) SPECjbb2005 (32 warehouses)

Figure 6: Comparing the throughput performances of the proposed collectors against the Appel collector and the fixed-ratio collector (optimal ratio = 2)

of the fixed-ratio collector. For SPECjbb2005, the improvements were 68% (dynamic-ratio) and 133% (hybrid).

5.3 Execution Time

In this section, we report the effects of our collector on the execution times of *xalan* and *javac*. We also wish to report that for applications not benefiting from using our collectors, the execution times were virtually unchanged, implying that the overhead of critical point detection is negligible.

Execution time of *javac*. We set the heap to be about twice as large as the maximum live-size so that the application would still invoke a reasonable number of garbage collection, without being excessive. Our result indicated that the dynamic-ratio collector increased the execution time by 2% (10.13 seconds for the fixed-ratio collector and 10.36 seconds for the dynamic-ratio collector) and the hybrid collector reduced the execution time by 14% (8.68 seconds for the hybrid collector).

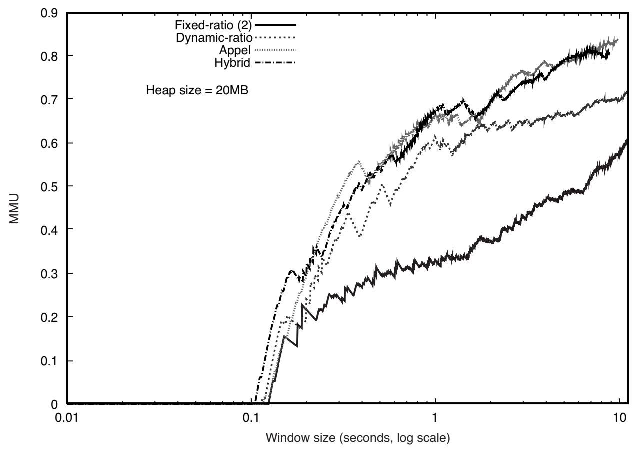
Execution time of *xalan*. Again, we set the heap size to be about twice as large as the maximum live-size. We initially specified 56MB for the heap but due to internal alignment and round-up, HotSpot assigned 64MB for the heap. Our result indicated that the both collectors reduced the execution time by about 7.5%.

5.4 Mutator Utilization

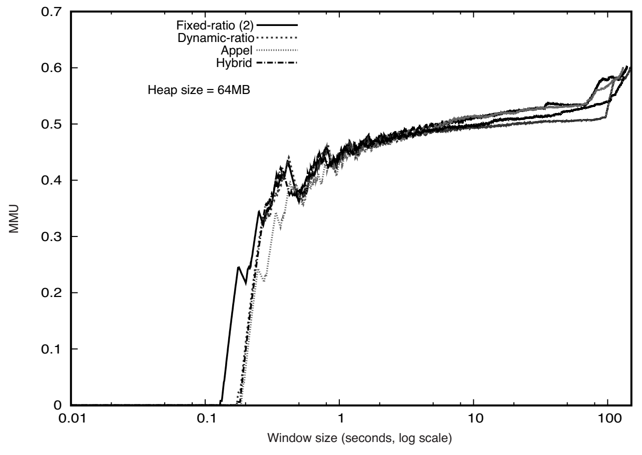
We used MMU (minimum mutator utilization) [3] to measure the pause time and mutator utilization. Mutator utilization is the fraction of the time that an application (or mutator) executes within a given window. For example, given an execution window of 10 ms, within that time the collector runs for 4 ms, and the mutator runs for 6 ms. Thus, the mutator utilization is 60%. The *minimum mutator utilization* (MMU) is the minimum utilization across all execution windows of the same size. For example, an MMU of 40% at 10 ms means that the application will at least execute 4 ms out of every 10 ms. Figure 7 depicts the MMU of each collector for each benchmark. The x-intercept indicates the maximum pause time, and the asymptotic y-value indicates the mutator utilization.

Typically, an application that invokes minor collection more frequently and seldom invokes full collection often yields shorter pauses and higher overall mutator utilization. In *javac* the mutator utilization of the hybrid approach was the highest. This was because it invoked a large number of minor collections. On the other hand, *xalan* showed very little effects from different collectors because the number of minor collection and full collection invocations only changed slightly when different collectors were used.

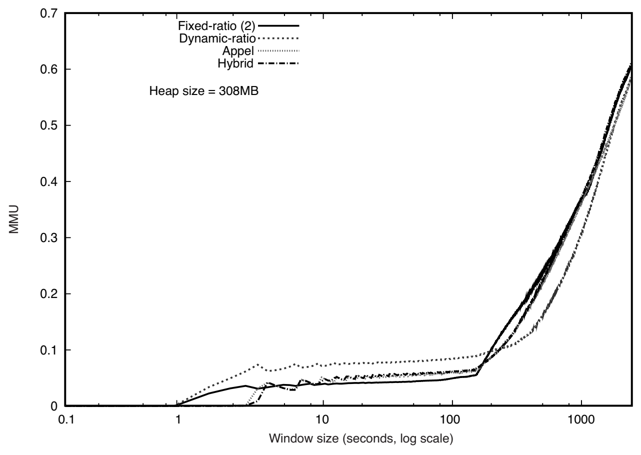
For SPECjbb2005, both the dynamic-ratio and hybrid collectors



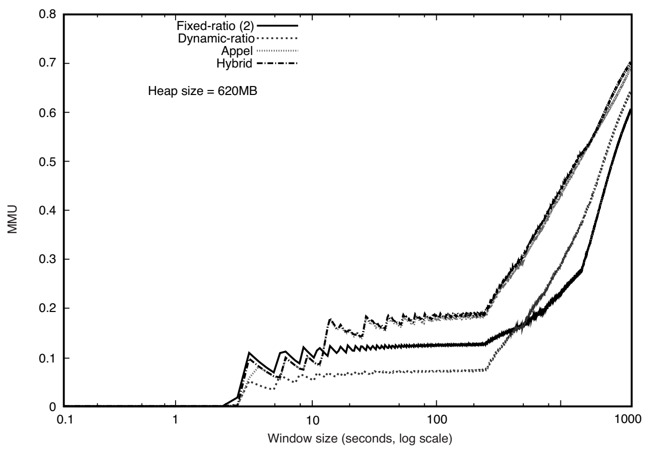
(a) javac



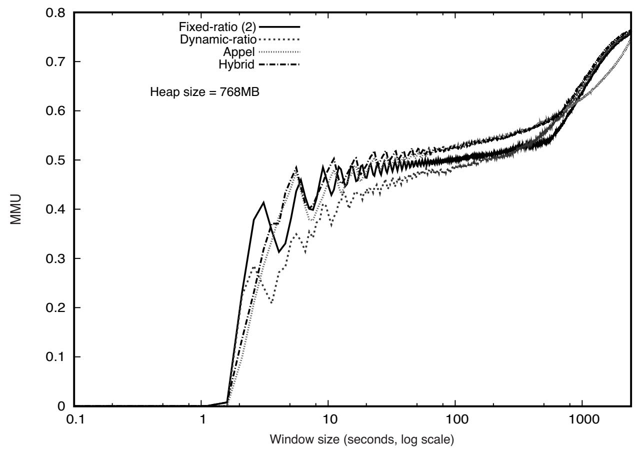
(b) xalan



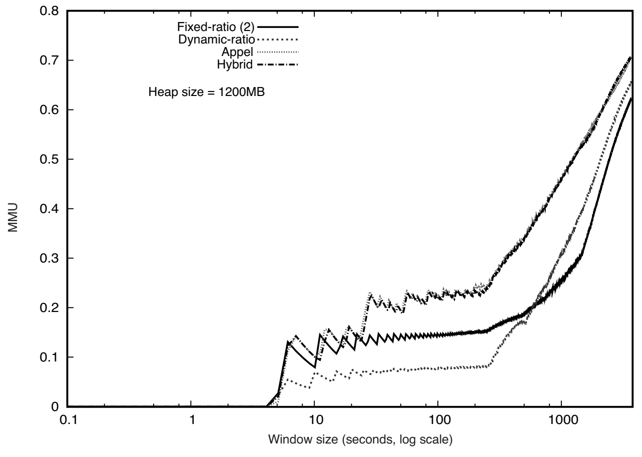
(c) SPECjbb2000 (16 warehouses)



(d) SPECjbb2005 (16 warehouses)



(e) SPECjbb2000 (32 warehouses)



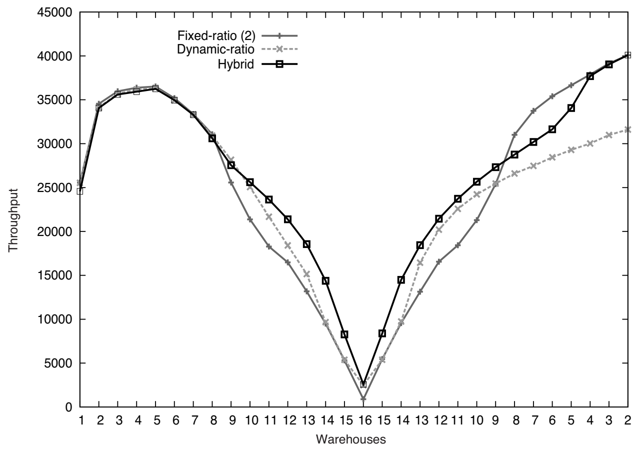
(f) SPECjbb2005 (32 warehouses)

Figure 7: Comparing minimum mutator utilizations (MMUs)

significantly impacted the throughput performances, especially at higher workload. Moreover, the two collectors significantly increased the number of minor collection invocations and reduced the number of full collection invocations; thus, major differences in the MMUs were observed. For SPECjbb2000, the differences in MMUs were not as wide ranging. This might be due to less performance impacts from our collectors.

5.5 Ability to Switch Back

We modified SPECjbb2000 to decrease the workload after 16 warehouses. Basically, the application starts destroying its warehouses one at a time until it reduces the number of warehouses to 2. We used this setting to emulate decreasing demands in server applications. Figure 8 depicts our collectors' abilities to switch back to the optimal ratio once the workload has lightened.



(a) javac

Figure 8: Switching back to fixed-ratio collector once the workload has lightened

Notice that the hybrid collector was able to easily switch back when the number of warehouses was reduced to 4. This was because the hybrid collector, once operated in the Appel style, checked the size of the mature space after each minor collection, which occurred very frequently. On the other hand, the dynamic-ratio experienced a long delay because switching could only occur after full collection. Prior to the workload reduction, the nursery was very small while the mature space was very large. Because objects were not prolifically created during the workload reduction process, full collection was never called so switching never took place. One possible approach to overcome this long delay is to use another criterion such as a reduction in number of threads to enforce switching.

6. RELATED WORK

There have been numerous research efforts to reduce the copy-reserve overhead and improve the performance of Appel collectors. Though the focus of our work is not on reducing the size of copy-reserve space, it is worth mentioning some of these efforts as they present opportunities to further improve the performance of our proposed collectors.

Work by Velasco *et al.* [26, 25] reports that the volume of surviving objects from the nursery during minor collection rarely exceeds 20% of the nursery; however, a collector often reserves 100% of the nursery to ensure successful minor collection. Their technique leverages information from prior GC invocations to safely reduce the size of the copy-reserve space. In doing so, the space is more efficiently utilized and the frequency of GC invocations is also reduced.

Their experimental results show a 16% speed-up of garbage collection time. The heap usage is also reduced by 19% to 40%. One possible issue with this approach is that objects in server applications can be much longer living than objects in desktop applications. The assumption that only a small portion of objects survives minor collection does not always hold and can cause their algorithm to fail.

Work by Sachindran and Moss [19] attempts to reduce the copy reserve space in the mature generation by partitioning the heap into small windows. Thus, the size of copy-reserve is limited by the size of each window. The copying phase is done in several passes, and each pass only "copying a subset of windows in the old generation" [19]. Because the HotSpot collector uses mark-compact with no copy-reserve space for full collection, this technique does not apply to our work.

Work by McGachey and Hosking [14] also reduces the copy-reserve space by exploiting the insight similar to Velasco *et al.* that only a small portion of objects survives a garbage collection invocation. However, their technique uses compaction to as a back-up in the case that their prediction is wrong. The back-up collector recovers additional copy-reserve space to ensure that all surviving data are "accommodated" [14].

In their technique, the copy-reserve space is set to be only a fraction of, instead of equal to, the nursery. In an instance that the volume of surviving objects from the nursery is larger than the copy-reserve space, an algorithm similar to mark-compact used in HotSpot is activated. In a way, their approach is more advance than HotSpot because it can recover from a failing minor collection by switching to compaction on the fly. If this scenario occurs in HotSpot, the failed minor collection would be partially completed. The objects that cannot be promoted stay in the nursery. The next allocation failure will result in full collection invocation.

Another related area to this work is dynamic switching of algorithms. Work by McGachey and Hosking switches from copying-based minor collection to compaction-based full collection on the fly [14]. The main criterion for switching is failing minor collection due to insufficient copy-reserve space. This criterion is the same as ours except that our algorithms do not invoke full collection, but instead reactively reduce the nursery size to allow more minor collection invocations. Work by Soman *et al.* [20] switches to different garbage collection techniques based on changes in execution profiles. An annotation-based technique is used to guide the selection process. Their work is based on Jikes RVM with MMtK [2] so many garbage collection techniques are readily available.

Work by Printezis uses hot-swapping to switch between mark-sweep and mark-compact to perform full collection [18]. The work does not modify the copying algorithm used for minor collection. The heuristic is that mark-compact can allocate objects much faster due to pointer-bumping algorithm; thus it is used when there is plenty of space in the mature generation (e.g. during initial start-up or after heap expansion). However, mark-sweep has lower execution cost due to non-compacting nature. Thus, when the heap is tight and full collection needs to be called frequently, mark-sweep should be used.

In effect, his approach tries to achieve the best of both worlds with these two algorithms. The goal of our work is similar to Printezis's in that we also try to achieve the best of both worlds through fixed-ratio and variable-ratio collectors. However, our focus is on the performance and efficiency of minor collection instead of full collection. Combining their work and ours will create an opportunity for further improvement that will be investigated as future work.

7. FUTURE WORK

In this paper, we have shown that the proposed dynamic-ratio and hybrid collectors can significantly improve the throughput degradation behaviors of the two server benchmarks, SPECjbb2000 and SPECjbb2005. Our collectors are based on the standard generational collector in HotSpot and not the concurrent collector [23], which supposes to yield the best throughput performance when used in multiprocessor environment [5]. However, a study by Xian *et al.* shows that the throughput degradation behavior of the concurrent collector is very similar to the standard generational collector [29]. Thus, integrating NMFLUX into the concurrent collector may make the throughput performance degrade more gracefully. This integration is outside the scope of this work but will be interesting to investigate in the future.

Work by Xian et al. [28] introduces a high-throughput generational collector for application servers or AS-GC. Their collector leverages the *key object* notion to segregate local and remote objects into two independent nurseries. Their result shows a 20% improvement in throughput performance and an ability to handle 10% higher workload before the memory is exhausted [28]. However, their optimization does not improve the degradation behavior so integration with our work would make an interesting study in the future.

It is quite common for the heap size of a large server application to exceed the physical memory capacity when facing unexpected heavy demands. In this scenario, paging activities become a major factor that limits and degrades throughput performance. So far, our study has not investigated the effect of our collectors on paging behavior. For example, because our collectors invoke fewer full collections, they may improve paging performance as full collection has been known to induce a large number of page faults due to heap traversal [7, 12, 16]. We are currently conducting such an investigation.

8. CONCLUSION

In this paper, we introduce NMFLUX, a framework to switch between a fixed-ratio collector and a variable-ratio collector for optimal throughput performance and graceful degradation behavior. Our framework leverages an insight that when copy-reserve space becomes too small, it is a sign that the nursery should be reduced, as the lifespan characteristic is no longer conform to the one initially used to tune the nursery size. We then utilized this framework to construct:

1. A dynamic ratio collector that incrementally reduces the nursery size each time an instance of two consecutive full-collection is detected.

2. A hybrid collector that combines the fixed-size collector with an Appel style collector. In our hybrid collector, the Appel collector replaces the fixed-size collector whenever an instance of two consecutive full collection invocations is detected.

Both schemes can switch back to fixed-ratio collector once the workload has lightened.

Our study has shown that these two collectors have very limited use in desktop-like applications. However, our experimental result indicates that both techniques, especially the hybrid collector, can make the throughput degradation behavior of server application more predictable and graceful. In effect, it can improve the serviceability of server applications under heavy memory demands as the throughput performance can improve by as much as 133%.

9. REFERENCES

[1] A. W. Appel. Simple Generational Garbage Collection and Fast Allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[2] S. M. Blackburn, P. Cheng, and 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)*, pages 137–146, Scotland, UK, May 2004.

[3] P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 125–136, Snowbird, Utah, USA, 2001.

[4] DaCapo Group. Dacapo benchmarks. http://dacapobench.org/.

[5] A. Gupta and 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/.

[6] M. Hertz and E. Berger. 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*, pages 313–326, San Diego, CA, USA, 2005.

[7] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 143–153, Chicago, IL, USA, June 2005.

[8] H. Hibino, K. Kourai, and 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, July 2005. http://www.csg.is.titech.ac.jp/paper/hibino-dsn2005.pdf.

[9] IBM. Jikes Research Virtual Machine. http://jikesrvm.sourceforge.net.

[10] IBM. Ibm websphere. http://www-306.ibm.com/software/webservers/appserv/was/, last visited June 2007.

[11] JBoss. Jboss Application Server. Product Literature, Last Retrieved: June 2007. http://www.jboss.org/products/jbossas.

[12] R. Jones and R. Lins. *Garbage Collection: Algorithms for automatic Dynamic Memory Management*. John Wiley and Sons, 1998.

[13] H. B. M. Jonkers. A fast garbage compaction algorithm. *Information Processing Letters*, 9(1):26–30, 1979.

[14] P. McGachey and A. L. Hosking. Reducing generational copy reserve overhead with fallback compaction. In *International Symposium on Memory Management*, pages 17–28, Ottawa, Ontario, Canada, June 2006.

[15] Microsoft. About the Common Language Runtime (CLR). http://www.gotdotnet.com/team/clr/about_clr.aspx.

[16] D. A. Moon. Garbage collection in a large lisp system. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 235–246, Austin, TX, 1984.

[17] ObjectWeb. JOnAS: Java Open Application Server. White Paper, Last Retrieved: June 2007. http://www.jonas.objectweb.org.

[18] T. Printezis. Hot-swapping between a mark&sweep and a mark&compact garbage collector in a generational environment. In *JVM'01: Proceedings of the JavaTM Virtual Machine Research and Technology Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 20–32, Monterey, California, April 2001.

[19] N. Sachindran and J. E. B. Moss. Mark-copy: fast copying GC with less space overhead. *SIGPLAN Not.*, 38(11):326–343, 2003.

[20] S. Soman, C. Krintz, and D. F. Bacon. Dynamic selection of application-specific garbage collectors. In *ISMM '04: Proceedings of the 4th International Symposium on Memory Management*, pages 49–60, Vancouver, BC, Canada, 2004.

[21] W. Srisa-an, M. Oey, and 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)*, pages 1065–1082, Agia Napa, Cyprus, 2005.

[22] Sun. Performance Documentation for the Java HotSpot VM. On-Line Documentation, Last Retrieved: June 2005. http://java.sun.com/docs/hotspot/.

[23] Sun Microsystems. Java Technology is Everywhere, Surpasses 1.5 Billion Devices Worldwide. Press Release, February 2004. http://www.sun.com/smi/Press/sunflash/2004-02/sunflash.20040219.1.html.

[24] 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*, pages 157–167, 1984.

[25] J. M. Velasco, K. Olcoz, and F. Tirado. Adaptive tuning of reserved space in an appel collector. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, pages 543–559, Oslo, Norway, June 2004.

[26] V. Velasco, A. Ortiz, K. Olcoz, and F. Tirado. Dynamic management of nursery space organization in generational collection. *interact*, 00:33–40, 2004.

[27] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 230–243, Chateau Lake Louise, Banff, Canada, October 2001.

[28] F. Xian, W. Srisa-an, C. Jia, and 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)*, pages 126–150, Berlin, Germany, July 2007.

[29] F. Xian, W. Srisa-an, and H. Jiang. 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*, pages 40–49, Mannheim, Germany, 2006.