Investigating the Effects of Using Different Nursery Sizing Policies on Performance

Xiaohua Guan, Witawas Srisa-an, and Chenghuan Jia

Department of Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115
{xguan,witty,cjia}@cse.unl.edu

Abstract

In this paper, we investigate the effects of using three different nursery sizing policies on overall and garbage collection performances. As part of our investigation, we modify the parallel generational collector in HotSpot to support a fixed ratio policy and heap availability policy (similar to that used in Appel collectors), in addition to its GC Ergonomics policy. We then compare the performances of 15 large and small multithreaded Java benchmarks; each is given a reasonably sized heap and utilizes all three policies. The result of our investigation indicates that many benchmarks are sensitive to heap sizing policies, resulting in overall performance differences that can range from 1 percent to 36 percents. We also find that in our server application benchmarks, more than one policy may be needed. As a preliminary study, we introduce a *hybrid* policy that uses one policy when the heap space is plentiful to yield optimal performance and then switches to a different policy to improve survivability and yield more graceful performance degradation under heavy memory pressure.

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

General Terms Experimentation, Languages, Performance

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 tedious and error prone task. As of now, one commonly adopted GC strategy is generational garbage collection.

Generational collectors concentrate their collection efforts in the *nursery*, a memory area used for object creations [23]. Because the nursery is usually configured to be smaller than the mature space (an area to host surviving objects from the nursery), these collectors often yield shorter GC pauses than most other GC strategies. The three common ways to set the size of the nursery are: (i) to use fixed nursery/mature ratio throughout execution (e.g., serial generational collector used in *HotSpot*, the flagship Java Virtual Machine from Sun Microsystems [7]); (ii) to adjust the nursery size based on the amount of time spent in minor and full collection (e.g., parallel generational collector in HotSpot); and (iii) to adjust the nursery size based on the amount available memory after each collection (e.g., Appel collectors such as the one in Jikes RVM [1, 8]).

As modern Java applications become more complex, they also demand more heap memory during execution. A recent release of *jvm2008* benchmark suite from SPEC clearly shows this trend [19]. This Java benchmark suite is a collection of desktop and scientific applications, which replaces the aging and well-studied jvm98 [20]. There are two notable runtime differences between jvm2008 and jvm98: (i) a much higher degree of execution concurrency in jvm2008 than that of jvm98; and (ii) a much higher level of heap usage in jvm2008 than that of jvm98. In jvm98, there is only one multithreaded application, *mtrt*, which employs two threads. On the other hand, all applications in jvm2008 are multithreaded (most applications employ eight threads except for *sunflow*, which employs over three hundred threads). All applications in jvm98 can run with as little as 16MB of heap [6]. However, several benchmarks in jvm2008 need the heap to be at least several hundred megabytes to run.

Furthermore, Java is now widely used in the development of application servers. An application server is a software system that delivers applications to clients. It also handles business logic and data accesses for these applications. 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 [9], JOnAS [13], and JBoss [10]. Previous investigations have shown that it is very common for application servers to be utilizing hundreds to thousands of concurrent threads [29, 30]. These threads allocate objects from heaps that can be several gigabytes in sizes.

This Work. It is unclear *if and how* using different nursery sizing policies can affect the performance of these concurrent and heap-intensive desktop, scientific, and server applications. Because nurseries are used for initial object allocations, they must be carefully sized to provide ample time for short-lived objects to die prior to minor collection. Therefore, we hypothesize that improperly sized nurseries can lead to sub-optimal performances due to excessive minor and full collection overheads.

In this work, we investigate the effects of using the three different heap sizing policies on performances of 15 multithreaded

ISMM'09, June 19–20, 2009, Dublin, Ireland.
Copyright © 2009 ACM 978-1-60558-347-1/09/06...$5.00

benchmarks from jvm2008 and DaCapo benchmark suites[1] [4]. We also investigate the effects of nursery sizing policies on *jbb2005* and *jAppServer2004*, two application server benchmarks from SPEC [17, 18].

There are two performance metrics that we use in our investigation: execution time and throughput. Execution time is used to evaluate multithreaded applications from DaCapo. Throughput is used to evaluate the remaining benchmarks. We observe garbage collection behaviors (minor and full collection invocations and time spent in each of them) and *Minimum Mutator Utilization* (*MMU*) to further analyze the impacts of using different nursery sizing policies on performance. Our investigation reveals that some benchmarks respond well to the fixed-nursery policy while others respond well to the variable-nursery policies. Furthermore, server applications that exhibit dynamic and fluctuating workload may require more than one policies to perform well throughout execution. We introduce a *hybrid* scheme that dynamically switches policies as workload profiles change.

The remainder of this paper is organized as follows. Section 2 provides the background information about HotSpot. Section 3 describes the three sizing policies used in this work. Section 4 describes the experimental methodologies used to conduct this work. Section 5 reports the results of our investigation. Section 6 describes our hybrid policy. Section 7 briefly discusses some of the existing related research efforts, and the last section concludes this paper.

2. Parallel Collector in HotSpot

Parallel collector in HotSpot uses copying to collect the nursery (*minor collection*) and mark-compact to collect the entire heap (*full collection*). Both types of collection can be set to utilize multiple collector threads. The nursery is further partitioned into three areas: *eden* and two survivor spaces, *from* and *to*, which collectively account for 20% of the nursery in the default configuration of HotSpot. 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. 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 swapping 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 [21], 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, most of these objects are promoted directly to the mature generation.

Similar to most copying collectors, HotSpot uses a *copy-reserve* space to ensure that the amount of available memory in the mature generation is large enough to accommodate surviving objects from a minor collection. It is possible that all objects in the nursery survive a 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 is invoked. 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. 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 [11]. The last phase slides objects toward the lowest address of the mature space.

To be able to adjust the size of the nursery space, the parallel collector initializes the starting address of the mature space at the lowest address of the heap. In this layout, the compaction process slides objects toward the lowest address, leaving unused memory at the top (higher-address) of the mature space. Right above the mature space is the nursery. After each minor collection, the eden space is empty, allowing a straight-forward adjustment of the nursery size.

3. Nursery Sizing Policies

In this section, we described three common sizing policies that we will study in this paper. The first policy is GC Ergonomics, used by parallel collector in HotSpot. (We refer to GC Ergonomics as *default*.) This policy dynamically adjusts the ratio based on minor and mature collection overheads. The second policy is maintaining the same ratio between nursery and mature space throughout execution. This is the policy used by the serial collector in HotSpot. (We refer to this policy as *fixed ratio*.) The third approach is adjusting the nursery size based on the object occupancy in the mature space [1, 8]. (We refer to this policy as *heap availability*.)

3.1 GC Ergonomics Policy (Default)

HotSpot's parallel collector uses this policy to strike a balance between maintaining small heap size and incurring small garbage collection overhead. In this policy, the collector monitors the overheads of minor and full collection invocations. Such information is then used to determine the size of to/from spaces, the value of the tenuring threshold, the size of the nursery, and the size of the entire heap. By adjusting these parameters, the policy attempts to: (i) keep GC pauses below a specified goal, (ii) achieve a specified throughput goal, and (iii) achieve small heap footprint [22].

To achieve these three goals, the implementation of GC ergonomics policy makes sizing decisions in the following order:

1. If a GC pause time is greater than the desired goal, the policy adjusts the corresponding generation size in hope of achieving the desired goal in the next collection invocation. It can also adjust tenuring threshold to reduce minor collection overhead by promoting more objects into the mature space [24, 25].
2. If the pause time goal is met, the policy considers the throughput performance. If the throughput is below the desired goal, it adjusts the heap in hope of meeting the throughput goal.
3. If the pause time and throughput goals are met, the policy decreases the generation sizes to reduce footprint.

Specifically, to determine if the nursery size should be enlarged or reduced to meet a pause time requirement, a pause estimator is used to make decisions. The estimator monitors the relationship between nursery and mature space sizes, X, and pause time, Y, to find the slope of the function Y=f(X). If the slope is bigger than 0, it decreases the generation size to reduce collection pause time for this generation. To meet a throughput requirement, the estimator first determines if the actual throughput (recorded as *mutator_cost*) meets the requirement, if it does not, the size of the generation with larger GC time is adjusted. There is also an estimator to adjust the tenuring threshold [24, 25]. If both of these goals are met, the policy tries to reduce the heap size to conserve memory usage, while ensuring that the two goals continue to be met.

[1] The version of DaCapo benchmarks that we used is dacapo-2006-10.

3.2 Fixed Ratio Policy (FR)

We implement this policy so that 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 X86-64 processors). Our command-line interface is similar to HotSpot's serial collector, which uses this policy. Once set, the ratio stays fixed throughout an execution. For example, if the initial heap size is 99MB, using the fixed ratio approach with 1:2 ratio, the nursery size is set to 33MB, and the mature space size is set to 66MB. If later on, the heap is enlarged to 198MB, the ratio between the nursery and mature space remains the same at 1:2; that is, the nursery size is now 66MB and the mature space size is 132MB.

3.3 Heap Availability Policy (HA)

In this policy, 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. The availability based policy 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 or back-to-back allocation failures in the nursery occur (we use the latter criterion in our implementation). At that point, the system makes a full collection invocation. Appel generational collectors and Jikes RVM generational collector have utilized the HA policy [1, 8]. Note that our implementation of this policy utilizes *no survival spaces* (i.e., no *to* and *from* spaces).

4. Experiment

Computing Platform. We conduct our experiment on an Intel Xeon system with four 2 GHz dual-core processors (total of 8 processors). The system has 16 GB of physical memory. We used the parallel collector in HotSpot with necessary modifications to support three nursery resizing policies. When we run jAppServer2004, the described system is used to host JBoss, a widely used open-source Java application server.

Benchmarks. The focus of our work is to investigate the effects of sizing policies on performance of heap-intensive, multithreaded applications. We choose to focus on this type of benchmarks because they are more representative of modern applications, which rely more on thread-level parallelism and create heavy allocation pressure on heap allocators. Based on this selection criteria, we need to subset DaCapo benchmark suite to only include four multithreaded applications. We include every application in the jvm2008 suite. However, we do not include the start-up versions of these applications, which are mainly used to evaluate performances of code optimizers. For server-side benchmarks, we use two benchmarks from SPEC: *jbb2005* [18] and *jAppServer2004* [17]. The basic characteristics of these benchmarks are provided in Table 1. Note that we determine the minimum heap requirement for each application by using the parallel collector that utilizes the GC Ergonomics policy.

Metrics. We investigate the garbage collection performance by observing the number of minor and full GC invocations and the time spent in each GC invocation. We also report *Minimum Mutator Utilization* as a metric that describe disruptions of application execution due to garbage collection [2, 5]. Mutator utilization is expressed as a fraction of time that an application or a mutator executes within a given time window. For example, given an execution window of 100 ms and within this time, the mutator runs for 70 ms and the garbage collection runs for 30 ms. Therefore, the mutator utilization is 70%. MMU is the minimum utilization across all execution windows of the same size.

In terms of performance, benchmarks from the DaCapo suite report their results using time (seconds). On the other hand, all SPEC benchmarks report their scores using either operations per minute (jvm2008), Business Operation Per Second or BOPS (jbb2005), or jAppServer Operations Per Second or JOPS (jAppServer2004). We also report our results using these performance metrics.

Because large servers must be able to handle varying workload while yielding graceful performance degradation, we also conduct experiments to investigate the effects of using different sizing policies on the performance degradation behaviors of jbb2005 and jAppServer2004.

HotSpot Configuration. We follow Sun's suggestion by setting the initial ratio between the nursery and mature space to 1 to 2 (i.e., the mature space is twice as big as the nursery). Our investigation shows that the ratio has very little effect on performance of most applications. However, in the two server benchmarks, the suggested ratio yields the best throughput performance leading up to the targeted workload level. The difference can be as much as 20% in jbb2005. We also use the default configuration of GC Ergonomics; that is, the pause target is left as undefined, and the throughput target is left as 99% (mutator utilization is 99%). Based on this configuration, the default collector makes generation sizing decisions to *maximize throughput performance* and *not to control GC pause*. We choose this configuration to force the GC ergonomics policy to keep trying to maximize throughput but without having to be concerned with maintaining low GC pause time.

For each application in jvm2008 and DaCapo, we initialize the *minimum* and *maximum* heap sizes to be twice as large as the minimum requirement shown in Table 1. We set up the size this way to ensure that all three policies have the same amount of heap throughout execution. Otherwise, the GC Ergonomics policy may reduce the heap size (i.e., when the throughput goal is met); an action that can negatively impact performance. We find that such initial heap sizes allow many applications to execute with good efficiencies while invoking reasonable numbers of GC invocations.

For our server applications, we configure the heap size to yield the optimal throughput for a specific workload level. We set the heap size of jbb2005 to 1GB. This heap size is large enough for jbb2005 to achieve the highest throughput performance when the workload is 7-warehouse. For jAppServer2004, the workload is controlled by increasing or decreasing the injection rate (Tx), transactions per second injected into the application server by the driver. In our experiment, we set the heap size to 256MB. With this setting, jAppServer2004 exhibits a linear throughput performance improvement when the injection rate is between 1 and 40. To achieve good CPU utilization during garbage collection, we configure HotSpot to utilize 8 minor collection threads and 8 full collection threads.

Methodology. We execute each benchmark *five times and report the best, the worst, and the average scores*. The average scores are also used to provide graphical illustrations in the next section. For MMU illustrations, we randomly pick one of the five runs to calculate mutator utilization. That same run is also used to report garbage collection performance for each application.

5. Evaluation

We report the performance of each benchmark in Table 2. Notice that each benchmark suite uses a different performance metric. For example, DaCapo suite reports the result of each benchmark in seconds. On the other hand, jvm2008 reports the results in operations

Benchmark	Description	Total allocations		Minimum	Number of
		objects (million)	bytes (GB)	heap requirement (MB)	threads
DaCapo (only include multithreaded applications running with "-s large" configuration)					
eclipse	Execute non-GUI JDT performance test of Eclipse IDE.	129	11.70	49	16
hsqldb	Execute a number of transactions against a model of a banking application.	10	0.60	323	407
lusearch	Perform a text search of keywords over a corpus of literature data.	17	2.59	9	70
xalan	Transforms XML documents into HTML.	59	7.05	33	14
jvm2008 (exclude start-up versions)					
compiler.compiler	Java compiler from OpenJDK 7 compiling itself.	392	21.10	313	8
compiler.sunflow	Java compiler from OpenJDK 7 compiling sunflow a sub benchmark from jvm2008.	449	25.33	194	8
compress	Compress data with a modified LZW method.	1	19.06	88	8
crypto.aes	Encrypt and decrypt data using the AES and DES protocols.	2	96.44	39	8
crypto.rsa	Encrypt and decrypt data using the RSA protocols.	504	37.15	6	8
crypto.svf	Sign and verify using various protocols.	90	108.56	15	8
derby	An open-source database written in pure Java.	597	28.01	463	8
mpegaudio	An MP3 encoder using JLayer	10	42.66	5	8
scimark.fft (large)	A subset of a floating point benchmark from NIST.	<1	5.41	579	8
scimark.lu (large)	A subset of a floating point benchmark from NIST.	<1	737	612	8
scimark.sor (large)	A subset of a floating point benchmark from NIST.	<1	0.73	353	8
scimark.sparse (large)	A subset of a floating point benchmark from NIST.	<1	4.86	511	8
scimark.mc	A subset of a floating point benchmark from NIST.	< 1	0.71	9	8
serial	Serializes and deserializes primitives and objects. using data from the JBoss benchmark.	547	28.73	333	8
sunflow	A multi-threaded global illumination rendering system.	316	0.55	19	316
xml.transform	Test implementations of java.xml.transform.	532	32.86	39	8
xml.validation	Test implementations of java.xml.validate.	543	61.08	109	8
jbb2005	A Java program emulating 3-tier sytem focusing on the middle tier (Wh = 8).	323	19.86	373	8
jAppServer2004	A JEE benchmark emulating an automobile maker and its network of dealers (Tx = 40).	2096	181.44	177	1116

Table 1. Benchmark Characteristics

per minutes. Thus, for the results of DaCapo benchmarks, a lower reported value means higher performance. For the rest, a higher reported value means higher performance. The best average performance for a particular benchmark appears in bold face.

5.1 Performances of DaCapo and jvm2008

In *eclipse*, *xalan*, *compiler.compiler*, the entire *scimark* suite, *serial*, and *xml.validate*, the performance differences are less than 10%, meaning that sizing policies have small impacts on the performance of these benchmark programs. For the remaining eight benchmark programs, the performance differences within an application, as illustrated in Figure 1, range from 10% to 36% when normalized against the policy that yields the lowest performance. The results also indicate that in three out of eight applications, the HA policy outperforms *default* and FR.

5.1.1 Discussion

We report the garbage collection behavior of each application in Table 3. Note that applications that show performance differences of 10% or more appear in bold face. For *lusearch*, *crypto.aes*, *crypto.svf*, *derby*, *sunflow*, and *xml.transform*, we can clearly see significant differences in the number of minor or full collection invocations among policies. However, in *hsqldb*, there is very little different in the number of both minor and full GC calls. In the remainder of this section, we analyze possible causes that result in performance differences in these eight applications.

hsqldb. As stated earlier, the number of GC calls are very similar across all policies (the HA policy invokes one more minor GC calls). However, the fixed and default policies outperform the HA policy by as much as 20%. Further analysis reveals that hsqldb spends a significant portion of total execution time on garbage collection (50% in the fixed ratio approach and 60% in the HA approach). This is an indication that setting the heap size to be twice the minimum heap requirement may not provide sufficient heap space for this application. By analyzing garbage collection time, we find that FR spends less time in full collection (3.76 seconds instead of 5.87 and 4.65 seconds as in HA and default, respectively).

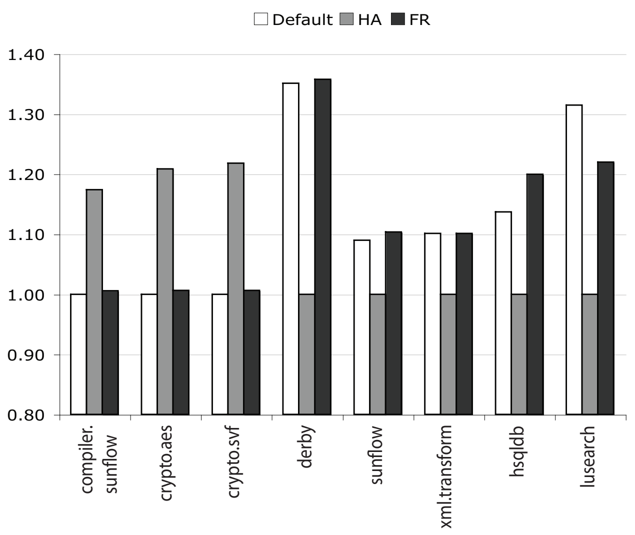


Figure 1. Performance improvements in applications that are sensitive to sizing policies

Benchmark (heap size)	Default			HA			FR		
	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
DaCapo (Each benchmark report result in seconds; therefore, lower value is better.)									
eclipse (98MB)	62.61	65.07	**63.63**	64.11	67.28	65.60	63.08	65.44	63.92
hsqldb (646MB)	8.39	9.50	8.79	9.11	11.01	9.99	8.02	8.68	**8.33**
lusearch (18MB)	6.69	7.45	**7.10**	8.90	9.61	9.33	7.38	7.99	7.65
xalan (66MB)	33.77	34.29	33.99	32.19	34.02	**33.24**	32.98	35.26	33.97
jvm2008 (Each benchmark reports result in operations per minute; therefore, higher value is better.)									
compiler.compiler (626MB)	241.2	243.24	242.05	254.09	266.6	**262.38**	240.58	246.28	243.99
compiler.sunflow (388MB)	76.3	77.5	76.97	88.92	91.75	**90.36**	75.86	78.87	77.40
compress (176MB)	224.33	236.34	**229.17**	210.41	219.42	216.66	218.45	237.56	228.54
crypto.aes (78MB)	33.42	34.01	33.67	39.94	41.1	**40.68**	33.71	34.01	33.88
crypto.rsa (12MB)	184.74	190.02	187.08	183.45	187.16	185.40	185.22	189.31	**187.76**
crypto.signverify (30MB)	151.84	154.28	153.00	185.5	187.18	**186.38**	152.69	155.36	153.95
derby (926MB)	72.5	75.16	73.25	53.05	54.64	54.21	72.91	74.69	**73.59**
mpegaudio (10MB)	64.23	66.33	65.47	61.64	62.84	62.44	64.32	66.54	**65.61**
scimark.fft (1.16GB)	20.57	20.83	20.74	20.51	20.94	**20.77**	20.59	20.78	20.69
scimark.lu (1.22GB)	5.19	5.67	5.50	5.56	5.71	**5.61**	5.13	5.78	5.48
scimark.mc (18MB)	159.06	159.78	159.46	150.33	160.09	157.81	159.43	164.46	**160.66**
scimark.sor (706MB)	29.48	30.6	30.28	30.57	30.99	**30.81**	29.83	30.57	30.24
scimark.sparse (1.02GB)	11.27	12	11.75	12.37	12.88	**12.68**	11.28	12.06	11.60
serial (666MB)	51.4	54.33	52.78	48.52	50.14	49.36	52.24	53.75	**53.17**
sunflow (38MB)	28.55	33.13	32.18	28.37	32.82	29.52	31.64	33.1	**32.58**
xml.transform (78MB)	97.37	98.33	**97.75**	87.56	90.06	88.74	95.9	98.64	97.70
xml.validate (218MB)	198.08	203.87	201.23	200.78	205.55	**203.28**	196.91	200.85	198.76
Java Server Benchmarks (Each benchmark reports result in transactions per second; therefore, higher value is better.)									
jbb2005 at 7WH (1GB)	102977	116372	112559.20	91916	98001	95081.20	109211	116124	**113721.00**
jAppServer2004 at 40Tx (256MB)	26.68	27.63	**27.29**	26.46	26.79	26.70	26.80	27.26	27.06

Table 2. Overall performance of each benchmark. We run each benchmark five times and report the worst, the best, and the average performance scores. For each of the two server benchmarks, we report the peak performance when running a sequence of increasing workload. We also report the workload setting that yields the peak throughput performance and the heap size that we use.

Therefore, FR yields the shortest execution time. In terms of GC pause time, the HA approach yields the shortest maximum pause time at about 500ms but has lowest MMU throughout execution. On the other hand, the FR approach yields the highest MMU throughout execution.

lusearch. For this application, the default approach yields the best performance. This is because the overall cost of garbage collection is dominated by minor collection. Because HA makes many more minor GC invocations, its performance is much lower than those of default and FR. The additional minor collection time in HA implies that constantly reducing the nursery size may have resulted in more promotion of objects to the mature space, and thus, higher minor collection cost and one more full collection invocation. In terms of MMU, the default policy has slightly lower utilization than those of the other two approaches. This is because the longest full collection time of the default policy (51 ms) is higher than the longest full collection times of HA and FR (33 ms and 37 ms, respectively).

compiler.sunflow. In this application, the costs of minor and full collection in default and FR are about the same. On the other hand, HA makes about 10% fewer minor collection invocations. By observing the heap sizing behavior when HA is used, we see that most objects can be reclaimed in minor collection, and therefore, the accumulation of lived objects in the mature space occurs gradually. In such a scenario, there is a long period of time that the HA approach would be able to maintain larger nursery than the initial size of the other two approaches (33% of the heap size). A larger nursery means fewer minor collection invocations and more time for objects to die in the nursery. In terms of time, HA reduces the minor collection time by 20%, and therefore, resulting in higher MMU as the time window surpasses 25 seconds. However, it also yields longer full collection time.

derby, sunflow, and xml.transform. In these three applications, the HA policy invokes more minor collections and full collections. Further investigation reveals that by constantly reducing the size of the nursery, the HA approach promotes significantly more objects from the nursery to the mature space, spending over 60% more time in performing minor collection in the case of sunflow. However, these objects die soon after promotion. As a result, the HA approach spends up to 30% less time performing the full collection. However, because the overall cost of garbage collection is dominated by minor collection cost, larger minor collection times in HA significantly degrade overall performance of these applications. In terms of MMU, we find that when the nursery space is reduced to a very small size in HA, full collection can take much longer to complete. For example, in derby, the longest full collection time is 1.2 seconds. In the other two policies, the longest full collection times are less than 900 ms. Moreover, by making twice as many minor collection invocations as the other two policies, HA also yields significantly lower MMU throughout execution.

crypto.aes and crypto.svf. In these two applications, HA significantly reduces the number of minor collection invocations but increases the number of full collection invocations. To better understand such behavior, we observe the total allocation requests made by each of these applications and the minimum heap space needed to successfully execute each of these programs. We see that these two applications are extremely dynamic memory intensive (crypto.aes allocates 96 GB of objects while crypto.svf allocates 108 GB) but yet, most objects are short-lived as indicated by small minimum heap requirements (39MB for crypto.aes and 15MB for crypto.svf). Moreover, these applications also allocate many large objects (the average object sizes of crypto.aes and crypto.svf are 48KB and 1.2KB, respectively).

Benchmark	Default		HA		Fixed	
	Minor	Full	Minor	Full	Minor	Full
eclipse	583	14	599	25	593	14
hsqldb	8	5	9	5	8	5
lusearch	916	6	1946	7	915	6
xalan	1467	139	1142	155	1433	125
compiler.compiler	2048	186	2405	154	2038	183
compiler.sunflow	3001	344	2693	301	3034	354
compress	269	108	671	25	282	113
crypto.aes	6308	1010	2785	1540	6300	1023
crypto.rsa	6134	99	9112	84	6135	103
crypto.svf	12974	1731	5293	2884	13067	1716
derby	2639	3	5422	3	2622	3
mpegaudio	9347	163	25785	116	9820	209
scimark.fft	22	18	19	11	22	13
scimark.lu	22	10	14	13	20	11
scimark.mc	154	3	253	8	152	4
scimark.sor	6	3	4	2	6	3
scimark.sparse	36	9	17	7	32	8
serial	1293	3	2356	5	1302	3
sunflow	17060	198	35412	125	16658	194
xml.transform	19685	407	26552	709	19698	406
xml.validate	1467	139	1142	155	1433	125
jbb2005 (8-warehouse)	28	2	5217	51	1675	11
jbb2005 (23-warehouse)	13877	1312	41533	989	14054	1288
jAppServer2004 (25 Tx)	2523	99	11196	70	2527	86
jAppServer2004 (50 Tx)	205	2599	22664	238	164	2706

Table 3. Comparing garbage collection behavior when different sizing policies are used

In applications with such allocation and lifespan behaviors, nursery sizing can play an important role in determining performance. FR allocates about 33% of the heap for nursery, which is too small for such extreme allocation requests. Thus, minor GC is called very frequently. The default approach also initializes 33% of the heap for nursery. Even when its adaptive mechanism periodically enlarges the nursery size, such enlargements are not immediate so the system still suffers from excessive minor collection invocations. On the other hand, HA initially has a larger nursery size. Since these objects die very young, not many objects are promoted, and nursery sizes stay large. Having larger nurseries results in fewer minor collection invocations.

As these programs continue to execute, they start to maintain consistent heap residencies. While these residencies are small when compared to the total size of allocation requests (e.g., crypto.aes maintains an average residency of about 28MB or 36% of the heap), when programs reach these residencies, the nursery sizes of HA are often smaller than those of FR and default. Because these applications also allocate large objects, such allocations can trigger more frequent full collection invocations in HA, resulting in lower MMU throughout execution and longer maximum GC pause times.

5.2 Performances of jbb2005 and jAppServer2004

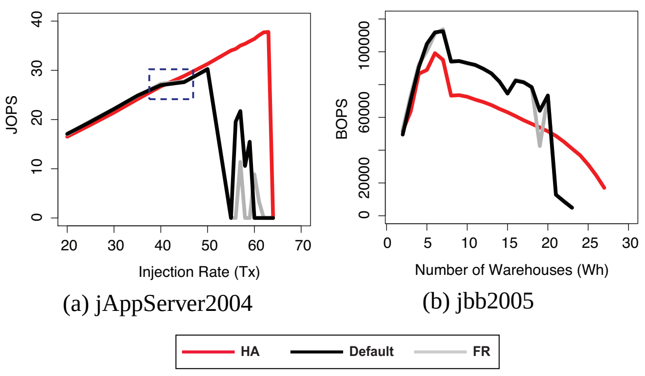
In this section, we investigate the throughput performance of our server benchmarks. Unlike other benchmarks used in this study, application servers are longer running, and their service demands can vary over time. In most instances, the periods of higher demands often "coincide with the times when the service has the most value" [28]. Thus, it is crucial for these servers to withstand heavy workload and have graceful performance degradation behavior when they are faced with unexpected heavy demands.

We set up our experiment to observe the peak throughput performance of each application and its degradation behavior once the workload is heavier than the optimal workload that the system is configured to handle. In these two benchmarks, we can increase workload by increasing the number of warehouses in jbb2005 and the number of transaction requests in a period of time (injection rate) in jAppServer2004. Theoretically, as we increase these two parameters (warehouse and injection rate), the throughput performance also increases. However, computing resource limitations and garbage collection disruptions can cause throughput performance to degrade as we increase these parameters. In the system that we use, increasing workload can cause the system to fail in unpredictable ways. Failures sometimes appear as out-of-memory errors. Other times, failures can appear as random errors that cause the system to behave erratically.

As stated in the earlier section, we set the heap size to 1GB and 256MB for jbb2005 and jAppServer2004, respectively. With 1GB heap size and using the default policy, jbb2005 can run to completion when the number of warehouses (Wh) ranges from 1 to 23 warehouses. We set the ramp-up/ramp-down times to 2 minutes and measurement time to be 4 minutes in each run. For jAppServer2004, the operational injection rate (Tx) ranges from 1 to 54 when the default policy is used, and the heap is set to 256MB. We set the ramp-up/ramp-down times to 5 minutes and measurement time to 10 minutes in each run.

When the default approach is used, we observe a linear increase in JOPS when the injection rate ranges from 20 to 40. In this range, we get an increase of 2.5 to 3 JOPS for every increase in Tx by 5. When the injection rate is increased beyond 40, the throughput performance continues to increase but at a lower rate. To ensure that the observed behavior is not an expected throughput behavior, we also conduct an experiment with 1GB heap. With a larger heap, throughput improvement remains linear from 20Tx to 80Tx. Thus, we conclude that the non-linear performance improvement at 40Tx is due to a small heap size and increasing GC efforts. The application becomes unstable at 55Tx and after. All of our experimental runs give out-of-memory errors at 60Tx. When FR is used, the application exhibits unstable behaviors once the injection rate surpasses 50. The system sometimes fails due to errors such as "connection refused" or "divided by 0".

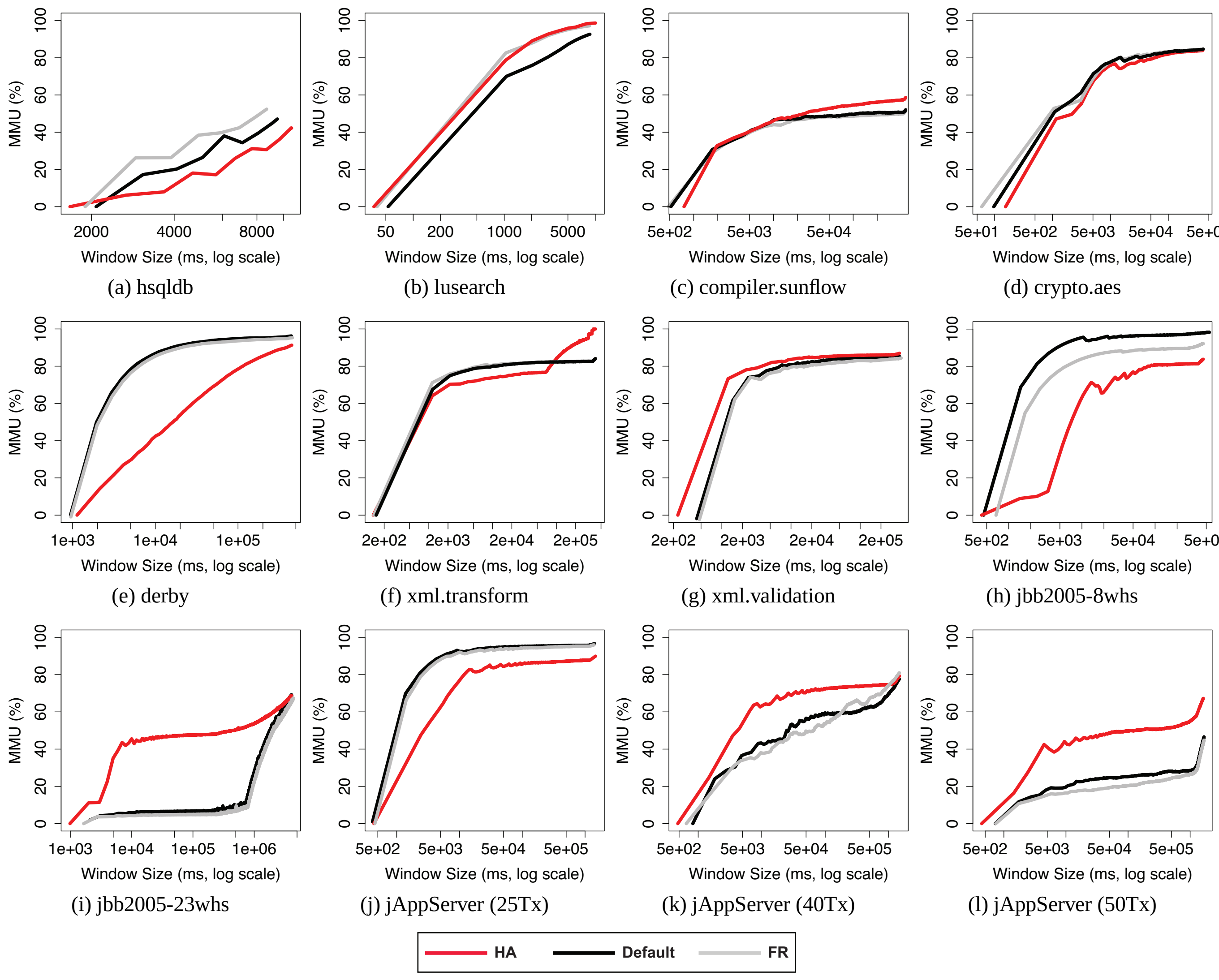
We report the throughput performance of these two benchmarks in Figure 3 and Table 2. As shown in Table 2, jbb2005 achieves its peak throughput performance when the workload is equal to 7-warehouse. For jAppServer2004, we report the throughput performance at 40Tx. This is not the workload that achieves the peak throughput performance, but it is the last workload before the throughput improvement becomes non-linear (highlight by the dotted box in Figure 3(a)). As shown in Figure 3(a) and (b), when employing HA, these two server benchmarks can handle higher workload than those of the default and FR policies before failing.



(a) jAppServer2004

(b) jbb2005

Figure 3. Throughput performance of jAppServer2004 and jbb2005



(a) hsqldb (b) lusearch (c) compiler.sunflow (d) crypto.aes
(e) derby (f) xml.transform (g) xml.validation (h) jbb2005-8whs
(i) jbb2005-23whs (j) jAppServer (25Tx) (k) jAppServer (40Tx) (l) jAppServer (50Tx)

Figure 2. Minimum mutator utilization (MMU)

5.2.1 Discussion

To better understand the impacts of sizing policies on throughput performance, we investigate the MMUs of these two benchmarks under different workload settings. Figure 2(h) and (i) illustrates MMUs of jbb2005. With 8-warehouse, the MMUs of FR and default are higher than that of HA throughout execution. Moreover, the MMUs of these two policies surpass 80% for most of the execution windows. They both are also higher than that of HA. Because default also adjusts nursery size, it can reduce the longest GC pause to be the same as HA. However, as the system is pushed to its limit (i.e., when Wh is equal to 23), the MMU of HA degrades much more gracefully than those of FR and default, which virtually show no utilization for most of the window sizes.

Figure 2(j), (k), and (l) illustrate MMUs of jAppServer2004 at 25Tx, 40Tx, and 50Tx, respectively. Again, at 25Tx, the MMUs of FR and default are higher than that of HA. However, as the workload increases to 40Tx, the MMUs of FR and default become lower than that of HA. It is worth noticing that when the window size is 5 seconds (5e+03ms), the MMUs of default is 90% at 25Tx, but drops to 40% at 40Tx. When the workload is 50Tx, the MMU at the same window size is only 20%. On the other hand, the MMUs of HA at the same window size are 60%, 57%, and 40% at 25Tx, 40Tx, and 50Tx, respectively. From this result, the MMU performance of HA appears to degrade more gracefully. Furthermore, it also appears that as the workload becomes heavier, the maximum GC pause remains the same for HA. On the contrary, FR and default experience longer pauses as the workload increases.

As stated in Section 2, copying generational collectors often utilize copy-reserve space, a space equaled to the size of the nursery located in the mature generation, to ensure that 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, and therefore, full collection is invoked.

Because the optimal ratio for the default and FR approaches is 1:2, the nursery occupied a large portion of the heap. Therefore, it becomes exceedingly difficult under heavy demands to maintain a large enough copy-reserve space to allow successful minor collection invocations. When such condition occurs, the system invokes full collection instead of minor collection. We discover that under heavy workload, this condition occurs repeatedly, leading to many consecutive full collection invocations.

For example, in the FR policy, jbb2005 invokes garbage collection 14502 times before running out of memory at 24-warehouse (not shown in the table). During the first 13506 GC invocations, only 645 of these invocations are full collection. The remaining 12861 invocations are minor collection. This is expected in any generational collector as most of the collection efforts should be concentrated in the nursery area. However, the last 995 invocations prior to memory exhaustion are full collection. Each of these full collection invocations takes 600 to 800 milliseconds to complete. Prior to each of these invocations, the heap is about 90% full; however, each full collection invocation only reclaims about 5% of the heap.

When Tx is equal to 50, jAppServer2004 also exhibits a similar behavior when FR or default is used. Full collection is invoked 16 times and 12 times more often than minor collection in FR and default, respectively. On the other hand, HA invokes 95 times more minor collection than full collection. This is because HA rarely suffers from the problem of copy-reserve space being too small. Thus, minor collection is still invoked under heavy workload. This is a major reason why the two benchmarks can handle higher workload when the HA policy is used. In the case of jbb2005, throughput performance also degrades more gracefully.

Remark. The result of our experiment indicates that techniques not suffering from consecutive full collection invocations during heavy demands will allow (i) the system to handle higher workload; and (ii) the throughput to possibly degrade more gracefully. In the next section, we introduce a hybrid technique that leverages the default or FR approach to achieve high throughput performance when the workload is light, but avoid suffering from repetitive full collection invocations when the workload is heavy.

6. A Hybrid Policy

So far, we have observed that the FR and default policies can perform well when the workload ranges from light to moderate. Once the workload becomes heavy, the HA policy allows garbage collection to be more efficient, enabling the system to handle higher workload. This insight indicates that a hybrid policy that complements default or FR with HA can provide high performance under expected workload and greater survivability under heavy workload.

In the remainder of this section, we briefly discuss a preliminary investigation of a hybrid policy that can switch between the default policy and the HA policy. We implement the proposed hybrid policy in HotSpot and conduct a preliminary evaluation using the two server benchmarks.

6.1 Implementation

In terms of programming, the proposed hybrid policy is simply an integration of two nursery resizing algorithms. However, one important challenge is deciding when to switch from the default policy to the HA policy. There are several criteria that can be used for switching. One approach that we contemplated was to monitor the throughput performance. However, this approach requires that a certain threshold be set to indicate that an application has reached an actual performance degradation zone and not just a temporary degradation due to other factors. Measuring performance differences also incurs runtime overheads.

Instead, we rely on a simple observation that consecutively invoking full collection often occurs right before our server applications reach performance degradation zones. Therefore, our current implementation simply records the number of consecutive full collection invocations. Once the number reaches $consecFailure$, which is a predefined value, our hybrid policy simply switches from the default policy to the HA policy. To identify a suitable value of $consecFailure$, we conduct an experiment using different values across several applications. Our experiment indicates that we can detect these performance degradation zones by monitoring for two consecutive full collection invocations. In the next subsection, we conduct a preliminary evaluation of our implementation by setting $consecFailure$ to 2.

6.2 Evaluation

As can be seen in Figure 4(a) and (b), our hybrid policy utilizes the default policy when the workload is light, then switches to the HA policy when it first detects back-to-back full collection invocations. For jAppServer2004, the hybrid approach produces stable performance and linear throughput improvement up to 63Tx, which is 8 Tx more than the highest workload that default can handle and 13 Tx more than the highest workload that FR can handle. This result is achieved while using the same heap size as the other two policies. For jbb2005 (see Figure 4(b)), the hybrid policy yields the highest performance up to 16-warehouse and then allows the throughput performance to degrade more gracefully after 21-warehouse. Furthermore, it allows jbb2005 to handle up to 27-warehouse while the default and FR approaches can only handle up to 23-warehouse when given the same heap size.

We also notice that it is possible to achieve higher performance in jbb2005 if our policy was to switch after 18-warehouse instead 15-warehouse. We can achieve this by making the switching criteria greater than 2 consecutive full collection invocations. This is because jbb2005 experiences a drop in throughput at 15-warehouse but recovers shortly after. In that drop, there is a back-to-back full collection. By changing the $consecFailure$ to be greater than 2, the hybrid policy does not switch until 18-warehouse. On the other hand, any value that is greater than 2 has no impacts on the performance of jAppServer2004. It is likely that the optimal criteria will be application dependent. For future work, we will investigate more adaptive ways to identify optimal criteria to switch from one policy to the next.

We also experiment with using the hybrid approach in other benchmarks. The results indicate that the proposed policy yields very little if any benefit to desktop and scientific benchmarks in the DaCapo and jvm2008 suites. Only scimark.fft shows a very modest improvement over the result of the previously best policy. This is expected as these benchmarks rarely have back-to-back full collection invocations when the heap is set to be twice the minimum memory requirements.

In the current implementation of jbb2005, we can only increase the number of warehouses in each run, but not decrease the number of warehouses. In real-world settings, workload can fluctuate so it is important for our proposed scheme to be able to switch back from HA to default if the workload should become lighter. To be able to test this feature, we create a modified version of jbb2005 (*jbb2005_mod*), in which the workload can be decreased. For example, if we configure the application to run from 1-warehouse to 23-warehouse with the warehouse increment of 1, our modified jbb2005 increases the workload in a similar way to the unmodified version. However, once a specified ending warehouse number (e.g., 23 in our experiment) is reached, instead of terminating, the application continues to execute by incrementally reducing the number of warehouses back to 1 in an increment of 1 as initially configured. Currently, the switch-back threshold is the number of consecutive minor GC invocations that results in the nursery sizes that are greater than or equal to the initial nursery size. In our experiment, the threshold is set to 10. The throughput performances of the modified jbb2005 is shown in Figure 4(c).

When the workload is getting heavier, the hybrid policy switches from default to HA to avoid promotion failures. This switch can be

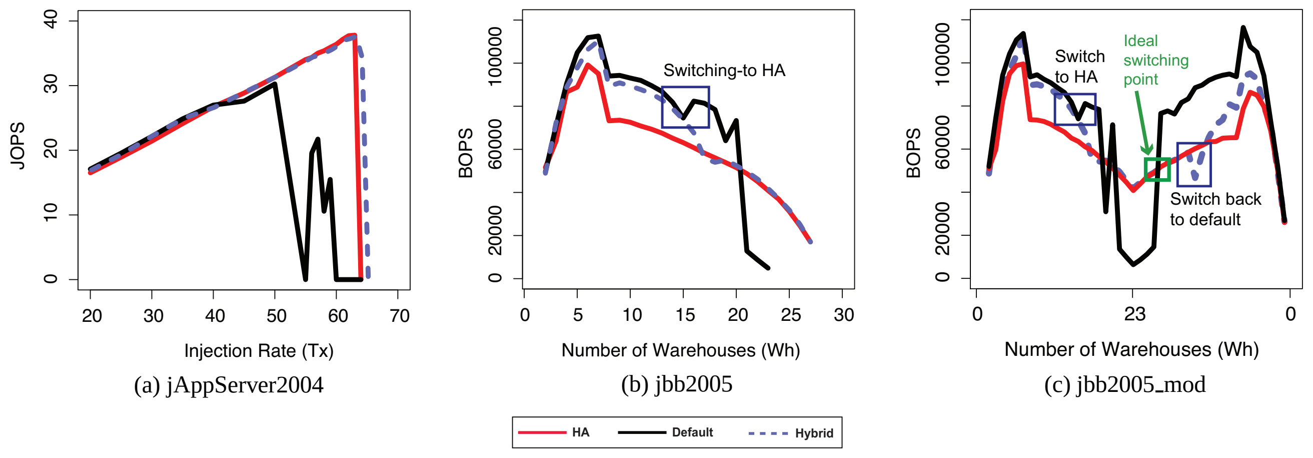


Figure 4. Comparing throughput performances of *default*, HA, and the proposed hybrid policy

done easily after a minor collection. On the other hand, switching from HA back to default is not as simple. This is because we need to invoke full collection to compact live objects, which in turn, increase the nursery size. Since we do not use other criteria to trigger this full collection, it often takes a long time to occur (e.g., after many minor collection invocations). Once it occurs and the new nursery size is larger than the initial nursery size of default, the system keep tracks of nursery sizes over the next several minor collection invocations before it can decide whether to switch. This switching delay is illustrates in Figure 4(c). For future work, we plan to create more benchmarks that have complex and fluctuating workload and investigate other possible criteria or runtime behaviors that can reduce the delay time in triggering this mechanism.

7. Related Work

There have been numerous research efforts to reduce the copy-reserve overhead and improve the performance of Appel collectors. Work by Velasco *et al.* [27, 26] 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 efficiently utilized and the frequency of GC invocations is 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 [15] 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" [15]. 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 [12] 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 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" [12].

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 [12]. 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.* [16] switches to different garbage collection techniques in Jikes RVM [3] based on changes in execution profiles. An annotation-based technique is used to guide the selection process.

Work by Printezis uses hot-swapping to switch between mark-sweep and mark-compact to perform full collection [14]. 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.

8. Conclusion

In this paper, we investigate the effects of using three different nursery sizing policies on garbage collection performance and overall performance. We conduct our investigation using 15 benchmarks. Our results indicate that most of these benchmarks are sensitive to nursery sizing policies and a favorable policies can increase the performance by as much as 36%.

Based on this insight, we develop a hybrid policy that can switch between the default policy and the HA policy for optimal throughput performance and graceful degradation behavior. Our policy exploits an observation that when the 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.

Our study has shown that our hybrid policy has very modest benefit in desktop-like applications. However, our experimental result indicates that our proposed policy can make the throughput degradation behavior of server applications more predictable and graceful. In effect, it can improve survivability by making server applications more capable of withstanding higher workload.

Acknowledgments

This work was sponsored in part by NSF under Awards CNS-0411043 and CNS-0720757. We thank Matthew Dwyer and Sebastian Elbuam for their contributions during the early stage of this work. We also thank the reviewers for their helpful comments.

References

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

[2] D. F. Bacon. Realtime garbage collection. *Queue*, 5(1):40–49, 2007.

[3] 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.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Eliot, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, Portland, Oregon, USA, 2006.

[5] 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, United States, 2001. ACM Press.

[6] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 java benchmarks. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'99)*, pages 92–115, Lisbon Portugal, June 1999. Springer Verlag.

[7] 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/.

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

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

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

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

[12] 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.

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

[14] T. Printezis. Hot-swapping between a mark & sweep and a mark & compact garbage collector in a generational environment. In *Proceedings of the JavaTM Virtual Machine Research and Technology Symposium on JavaTM Virtual Machine Research and Technology Symposium (JVM01)*, pages 20–32, Monterey, California, April 2001.

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

[16] 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.

[17] Standard Performance Evaluation Corporation. SPECjAppServer2004 User's Guide. On-Line User's Guide, 2004. http://www.spec.org/osg/jAppServer2004/docs/UserGuide.html.

[18] Standard Performance Evaluation Corporation. SPECjbb2005. On-Line Documentation, 2005. http://www.spec.org/jbb2005.

[19] Standard Performance Evaluation Corporation. Spec jvm2008 benchmarks, Last Retrieved: December 2008. http://www.spec.org/osg/jvm2008.

[20] Standard Performance Evaluation Corporation. Spec jvm98 benchmarks, Last Retrieved: June 2005. http://www.spec.org/osg/jvm98.

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

[22] Sun Microsystems. Garbage Collection Ergonomics. On-line article, 2005. http://java.sun.com/j2se/1.5.0/docs/guide/vm/gc-ergonomics.html.

[23] 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.

[24] D. Ungar and F. Jackson. Tenuring policies for generation-based storage reclamation. In *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 1–17, San Diego, CA, USA, 1988.

[25] D. Ungar and F. Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transacations on Programming Languages and Systems*, 14(1):1–27, 1992.

[26] 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, Norwary, June 2004.

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

[28] 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.

[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.

[30] F. Xian, W. Srisa-an, and H. Jiang. Contention-aware scheduling: Unlocking execution parallelism in multithreaded java programs. In *Conference on Object-Oriented Programming Systems, Languages, and Applications OOPSLA*, pages 163–179, Nashville, Tennessee, October 2008.