Thread-Aware Garbage Collection for Server Applications

Woo Jin Kim, Kyungbaek Kim, Jaesun Han, Keuntae Park and Daeyeon Park
Department of Electrical Engineering & Computer Science
Korea Advanced Institute of Science and Technology (KAIST)
Daejeon, Korea
wjkim,kbkim,jshan,ktpark@sslab.kaist.ac.kr, daeyeon@ee.kaist.ac.kr

Abstract

In recent years server applications using Java become popular. However, they have different performance requirements from other applications: high throughput and small response time. One of obstacles for achieving those requirements is a Java Virtual Machine (JVM). Among the services that a JVM provides, garbage collection affects server applications in throughput and latency. Some JVMs have various garbage collectors for server-side Java but they do not still consider the behavior of server applications.

We show that the lifetime pattern of objects is distinguished by the thread that allocates them in server applications. Separating objects and applying different collection policies according to threads, we propose that a garbage collector can achieve both high throughput and small pause time. Experiments show that the throughput of our collector is up to 1.7 times greater than that of previous generational collectors with the same pause time and the pause time of minor collection is smaller by almost 10% given the same throughput.

1. Introduction

Recently, Java server applications have become increasingly popular. Some of the examples are JavaServer Pages™ (JSP) and Enterprise JavaBeans™ (EJB) [13]. However, they have different performance requirements from other applications. Two important requirements are high throughput and small response time. That is, a server application should service as many requests as possible and deliver a response as soon as possible. The Internet explosion has led to a rapid increase in the number of users so that achieving these requirements becomes more difficult.

One of obstacles preventing server applications from satisfying these requirements is a Java Virtual Machine (JVM). A traditional JVM is based on technology that has originated from desktop computer environments such as a single processor. Garbage collectors in the JVM usually use only one processor and stop applications during garbage collection, which is inefficient for server computing environment such as multiprocessors. During garbage collection, only one processor is used for garbage collection and requests being processed before the collection should wait for the end of garbage collection. In result it drops throughput and increases response time of server applications.

Some JVMs such as the Java Hotspot Virtual Machine, v1.4.1 [12] and BEA Weblogic JRockit [4] aim for high performance server-side JVMs. The JVMs have some features to support server applications, such as adaptive optimizing compiler and parallel garbage collector or concurrent garbage collector. The parallel garbage collector uses all processors available to achieve both high throughput and small pause time, and the concurrent garbage collector performs while applications are running, which leads to high degrees of parallelism and non-disruptiveness. They are, however, only adapted to the server environment and do not consider the behavior of server applications.

The architecture and behavior of server applications differ from those of other applications. In Figure 1, a typical server application is based on multi-thread model, which uses many threads to handle requests from network. Each thread has its own role. The dispatcher thread in the figure is a main thread and keeps a pool of worker threads. It receives a request and passes it to one of worker threads. Then the worker thread processes the request and sends a response to the client. Observing the behavior of threads, there are some characteristics to distinguish one kind of threads from another kind. The dispatcher thread lasts to the end of running and manages overall processing of a server application and the worker thread makes objects related to a request and then makes other objects for the next request. From this point of view, objects that the worker thread uses can be regarded as temporal objects, and objects that the dispatcher thread makes live longer than those of worker threads. If a garbage collector knows the lifetime of an object, it can collect the object efficiently by concentrating on collecting

Proceedings of the 2004 International Symposium on Applications and the Internet (SAINT'04)
0-7695-2068-5/04 $20.00 © 2004 IEEE

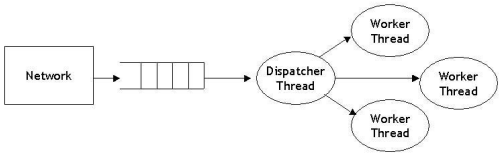


Figure 1. The architecture of a server application

short-lived objects. An example for that approach is a generational collector, which focuses on newly created objects that have a much lower survival rate than older objects [14], [9] . If it distinguishes which newly created object lives short or long, the efficiency can be improved further.

In our idea, the classification is based on the thread that allocates the object. Our collector applies different policies for the allocation and collection of objects according to threads. The policy we used for worker threads is to allocate their objects in a large nursery, which can be collected quickly because there are few live objects in the nursery; In result, it leads to high throughput, the amount of reclaimed memory space per a unit garbage collection time. On the other hand, the policy for main threads is to allocate objects in a small nursery. Collecting the nursery takes proportional time to the size of the nursery because the amount of live objects increases proportional to the size of the nursery. Therefore the size should be small to achieve small pause time. Applying different collection policies to objects according to the thread that created them, both high throughput and small pause time are achieved.

For our experiments, we use the Jikes Research Virtual Machine (RVM) [2] . It provides basic generational copying collectors with fixed size nursery and one with variable size nursery. Modifying these collectors we implemented the 'Thread-aware collector' to prove the hypothesis for the server behavior presented above and to evaluate the effect of exploiting the behavior. Using a web server based on the Staged Event-Driven Architecture (SEDA) [15] as a benchmark server, the experiments show that the throughput of thread-aware collector is up to 1.7 times greater than that of basic collectors while the pause time of minor collection is almost the same. In addition, the pause time of minor collection is smaller by almost 10% on the condition of similar throughput.

The remainder of this paper is organized as follows. Section 2 introduces basic generational collectors in Jikes RVM in short and Section 3 explains our idea based on the behavior of server applications. Section 4 shows our experiment environment and results. Section 5 presents related works and Section 6 concludes.

2. Background

In the Jikes RVM [2] , there are two generational copying collectors. One is with fixed size nursery (Fixed nursery collector or FN collector) and the other is with variable size nursery (Variable nursery collector or VN collector). The latter is also known as the Appel style generational collector [3] and its nursery is set to the maximum as much as possible.

There are two performance parameters of garbage collection: throughput and collection time. The collection time, the time taken for a minor or major collection, is proportional to the amount of live objects that are traced and copied. The collection time of minor collection is generally proportional to the size of a nursery because there are more live objects in large nursery than in small nursery. However, the throughput of garbage collection increases in large nursery because it gives more time for objects in nursery to die and less objects that will be garbage soon are copied to mature space; it reduces the number of major collections. In short, high throughput is in conflict with small collection time for deciding the size of nursery.

3. Proposed Idea

The basic generational copying collectors concentrate on newly created objects that have a much lower survival rate than older objects by allocating them to the nursery and collecting only the nursery mostly [14], [9]. It is an efficient strategy for high throughput and small pause time. However, it can achieve the best efficiency in server applications if it is adapted to the behavior of server applications.

In Section 1, we propose that the architecture and behavior of server applications differ from those of other applications. Main threads such as a dispatcher thread are responsible for server management while worker threads are for processing requests. They have two different characteristics in the memory management aspect: the object lifetime pattern and the allocation rate. First of all, main threads usually make long-lived objects while worker threads allocate temporal objects for a request. In result, objects that worker threads used are likely to be garbage. In addition, worker threads make many objects for processing requests so that the object allocation rate of worker threads is typically higher than that of main threads.

Current generational collectors allocate and collect all objects in one nursery regardless of these features. Short-lived objects of worker threads is mixed with long-lived objects of main threads and the amount of live objects is proportional to the size of nursery due to the long-lived objects. To achieve small collection time, the size of heap should be kept below some threshold. In result, garbage collectors

Proceedings of the 2004 International Symposium on Applications and the Internet (SAINT'04)
0-7695-2068-5/04 $20.00 © 2004 IEEE

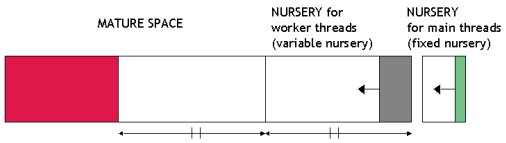


Figure 2. Thread-aware generational copying collector

cannot attain the best throughput due to the restricted size of nursery.

To exploit different object lifetime patterns of threads, different collection policies can be applied to each kind of threads. First of all, worker threads make temporal objects and the amount of live objects at a time is almost constant regardless of the amount of objects they allocated; only objects related to currently processed requests are live. In this case, the throughput of collection increases as the heap becomes large but the collection time is almost constant irrespective of the size of heap. Therefore the appropriate collection policy for worker threads is to allocate objects to a large heap in order to achieve good throughput.

On the other hand, main threads usually make long-lived objects. For that reason, the amount of live objects is proportional to the size of heap and the collection time as well as the throughput is proportional to the size of heap. To keep the pause time of collection below an acceptable value, the size of heap should be less than some threshold even though the throughput of garbage collection is not the best it could get. However, the throughput is not reduced much because of the lower allocation rate of main threads; even if the heap for main threads is small, it is filled at the slow rate and gives more time for objects to die. Consequently the proper collection policy for main threads is to allocate objects of main threads to a small fixed size heap to guarantee small pause time.

In short, applying different collection policies to threads according to the object lifetime pattern and the allocation rate, garbage collection becomes more efficient, and in result, provides higher throughput and smaller response time of server applications.

3.1 Thread-aware Collector Architecture

There are several ways to implement the proposed idea. Our Thread-aware collector(TA) is designed to be simple and easy to understand and analyze for the purpose of this paper: to prove the existence of different object lifetime patterns and allocation rate of threads in server applications and to show the effect of exploiting these features for garbage collection.

First of all, in Figure 2, our collector is basically generational copying collector and uses two nurseries for threads. One is a fixed size nursery (fixed nursery or FN) for main threads and the other is a variable size nursery (variable nursery or VN) for worker threads. To apply the collection policies discussed before, the size of fixed nursery is set to an appropriate value to limit maximum pause time and the variable nursery is managed similarly to the nursery of the Appel style collector [3] to make it as large as possible. The size of variable nursery is at least larger than quarter of the entire heap because the amount of live objects in mature space can grow up to at most half of the entire heap; if it exceeds half of the heap, a major collection begins. The size of fixed nursery is set to an appropriate value to limit maximum pause time.

Deciding collection policy of a thread can be performed by JVM automatically and dynamically, but in our collector, we let programmers of a server application decide the policy statically by setting a boolean value in the constructor of Thread class. This is simple but programmers should figure out lifetime patterns of threads by experiments or intuition. This mechanism works well as shown in Section 5.

Sequence of collection is similar to basic collectors. When the fixed nursery or the variable nursery is full, minor collection is initiated for the exhausted nursery. If the empty space of mature space is not enough after minor collection, major collection is needed. However, the major collection is performed in next collection in our collector while basic collectors start major collection immediately. This is because variable nursery should be collected before major collection to empty the space of variable nursery for major collection. In next collection, minor collection is performed for both two nurseries and then major collection begins.

4. Implementation

There are several changes applied to the basic collectors and Thread-aware collector from original Jikes RVM GC architecture. In the original architecture, a large object heap is used to allocate larger objects than 2KB to reduce copying cost of generational copying collector. Objects in the large object heap are managed by mark-and-sweep method instead of copying collection . For our purposes objects should be allocated separately according to threads and only one large object heap is not appropriate for our approach. Accordingly it is only used to allocate stacks of threads, which is a restriction of the Jikes RVM architecture[1], and all objects are allocated to the heap managed by generational copying collectors.

The original architecture considers shared-memory multiprocessor (SMP) configurations so that all garbage collectors are parallel collectors. Jikes RVM multiplexes Java

[1]The Jikes RVM is written in Java and all objects for the virtual machine are Java objects. So stacks of threads are also Java objects and they can be garbage collected.

Proceedings of the 2004 International Symposium on Applications and the Internet (SAINT'04)
0-7695-2068-5/04 $20.00 © 2004 **IEEE**

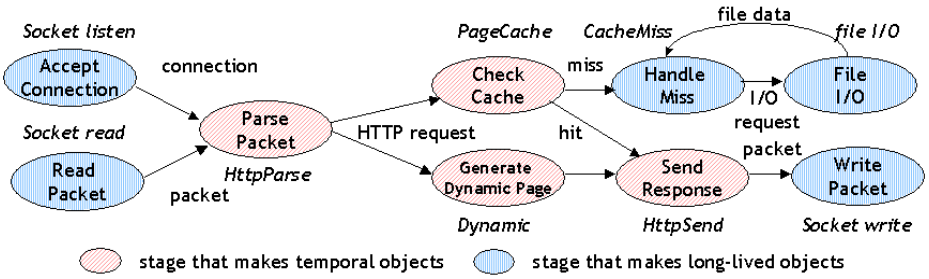


Figure 3. The architecture of Haboob web server

threads on virtual processors that are implemented on operating system threads. To reduce synchronization cost in allocation, a chunk, a large piece of nursery, is provided to each virtual processor. Objects are allocated to the chunk locally in a virtual processor and only the allocation of a chunk needs synchronization.

This approach has the internal fragmentation problem like paging system of virtual memory system. The problem is not serious in the original architecture because large objects are allocated to the large object heap. In our architecture, however, all objects including large objects have to be allocated in chunks and the ratio of fragmentation increases. We use direct allocation without chunk to avoid fragmentation and experiments are performed in single processor environment. Parallel version of Thread-aware collector is more complicated and this is a future work.

5. Experiments

5.1 Benchmark configurations

To evaluate the Thread-aware collector with server applications, we choose Haboob, a SEDA-based web server, as a benchmark server [15] . The architecture is an event-driven architecture with multiple stages. Figure 3 shows its stages.

Each stage has its own threads so that it makes the different object lifetime patterns of threads more clearly. Among the stages, by intuition, *Socket* stage for accepting and managing persistent connections and *Cache* stages for handling cache data of static pages may make more long-lived objects than the other stages. On the other hand, *HttpParse* stage that parses each request and *Dynamic* stage that generates dynamic contents may make many temporal objects. Accordingly *Socket* and *Cache* stages use fixed nursery while the other stages use variable nursery. The *Cache* stages manage fixed size cache for static pages.

We have chosen the load model from the SPECweb99 benchmark suite [10] as the basis for our measurements, which is similar to the model used for the Haboob evaluation [15] . We use static pages from SPECweb99 load, which constitute 70% of the SPECweb99 load mix. A dynamic page used in these experiments is borrowed from the SEDA bottleneck test; it makes several random numbers and generates a sum of them and returns an 8KB response to the client. We keep the static Web page file set fixed at 3.31 GB of disk files, corresponding to a SPECweb99 target load of 1000 connections. Files range in size from 102 to 921600 bytes and are accessed using a Zipf-based request distribution mandated by SPECweb99. More details can be found in [10] .

There are several parameters for performance of GC. The throughput of a collection is the amount of reclaimed space per unit collection time for the collection (KB/ms). Copy ratio of a collection represents ratio of the amount of copied objects for the size of region that is collected. It is inversely proportional to the throughput because collection time is proportional to the amount of copied objects. For the collection time of garbage collection, only that of minor collection is comparable because major collection of Thread-aware collector is performed by the same method with that of basic collectors.

5.2 Evaluations

5.2.1 Performance comparison with various size of heap

We compare the throughput and response time of basic collectors (Fixed Nursery and Variable Nursery) and Thread-aware collector. The number of collection is small for large heap so that more requests are used to cause enough garbage collection to show the average behavior. 700,000 requests are used for 500MB heap and 1,000,000 requests for 600MB heap.

In Figure 4(a), the overall throughput of Thread-aware collector is about 1.4 - 1.7 times greater than that of Fixed Nursery collector and about 1.1 - 1.4 times greater than that of Variable Nursery collector. It is from the improved throughput of minor collection in Thread-aware collector, as shown in Figure 4(b). The throughput of minor collection for the variable nursery in Thread-aware collector is 2.1

Proceedings of the 2004 International Symposium on Applications and the Internet (SAINT'04)
0-7695-2068-5/04 $20.00 © 2004 **IEEE**

Table 1. Performance of web servers and collectors

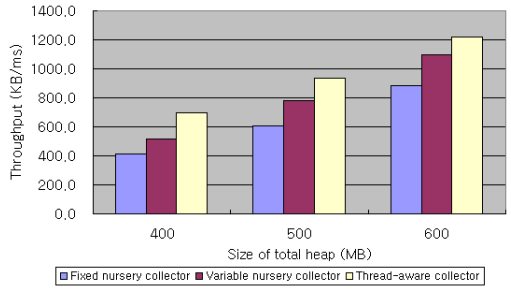
Collectors and Heap Configurations			Performance of Garbage Collection								Performance of Web Server	
			GC Count				copy ratio (%)			Pause Time (ms)		
H	Collector	NH	Total	FN	VN	M	FN	VN	M	Nursery (FN, VN)	90% RT	TP
400	FN	50	108	73	0	35	22.3	0	77.7	57.2	1615	358.4
	VN		124	0	108	16	0	26.5	61.0	58.1	1580	370.8
	TA	30	86	50	28	8	27.5	6.3	56.8	57.5 (61.3, 50.8)	1461	380.1
500	FN	70	115	95	0	20	16.0	0	64.5	62.4	1639	347.5
	VN		120	0	110	10	0	20.7	48.8	63.1	1619	351.3
	TA	40	99	59	34	6	22.5	5.1	45.8	64.4 (71.0, 53.1)	1512	363.2
600	FN	100	108	94	0	14	11.4	0	58.4	65.9	1767	312.9
	VN		114	0	107	7	0	16.5	40.9	68.0	1783	315.4
	TA	50	109	65	39	5	18.1	4.3	38.5	71.3 (80.0, 57.0)	1746	320.2

* FN : Fixed Nursery, VN : Variable Nursery, M : Major collection, TA : Thread-aware, CA : Copied Amount
* NH : The size of nursery (MB), 90% RT :90th percentile of response time (ms), TP : Throughput (reqs/s), H : The size of heap (MB)

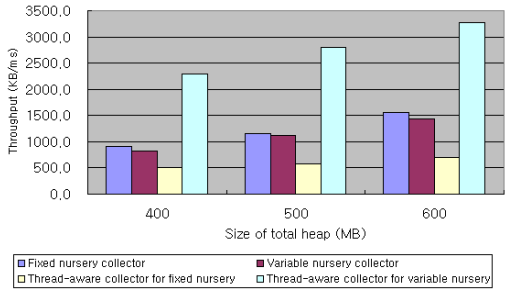
- 2.8 times greater than that of Fixed Nursery collector or Variable Nursery collector and it makes the overall throughput of Thread-aware collector better than those of the other collectors, even though the throughput for the fixed nursery in Thread-aware collector is lower than that of the other collectors. Notice that the throughput for the fixed nursery is almost constant for various heap sizes because the size of the fixed nursery is restricted to limit the pause time caused by the minor collection. It makes the collection time of minor collections small so that Thread-aware collector has high throughput while its collection time of minor collections is almost the same with that of other collectors, as shown in Table 1. In addition, the collection time of Thread-aware collector is smaller than that of other collectors on condition of similar throughput. The throughput of Thread-aware collector for 400MB heap is similar to that of other collectors for 500MB heap, but its average collection time is smaller by 8.4 - 10 % than the time of other collectors.

The improved throughput affects the performance of server applications. The performance of Haboob web server using each collector is shown in Table 1. Many factors such as I/O bandwidth affect the performance of server applications so that the effect of improved garbage collection does not appear apparently. However, the throughput of server using Thread-aware collector is improved by 2.0 - 6.0% compared with that of server using the other collectors.

For our purpose, the maximum response time should be compared but it depends on major collection. To examine the effect of improved minor collections, we compared 90th percentile of response time. In Table 1, the response time by Thread-aware collector is reduced by 1.2 - 10.5% compared to servers using the other collectors while the collection time of a minor collection for basic collectors and Thread-aware collector is similar with one another. The reason is



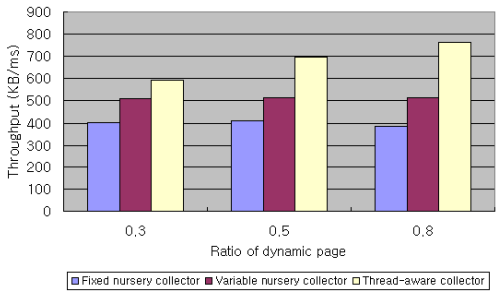
(a) Throughput of garbage collections



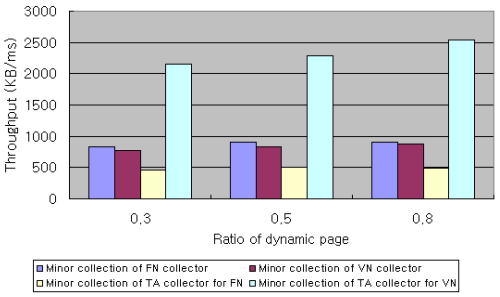
(b) Throughput of Minor Collections

Figure 4. Throughput of each collector for various heap size

Proceedings of the 2004 International Symposium on Applications and the Internet (SAINT'04)
0-7695-2068-5/04 $20.00 © 2004 **IEEE**



(a) Throughput of garbage collection



(b) Throughput of minor collections

Figure 5. Throughput of each collector for various types of workload

that the number of garbage collection for Thread-aware collector is reduced and the time interval between garbage collection becomes long.

5.2.2 Performance comparison with various workload

In this experiment, we examine the performance of collectors for various types of workload that have the different ratio of dynamic page requests. In Figure 5(a), the throughput of Thread-aware collector is increased as the ratio becomes large while that of the other collectors is almost the same. At the ratio of 0.3, the throughput of Thread-aware collector is 1.5 times better than that of Fixed Nursery collector, but it is 2.0 times better at the ratio of 0.8. This improvement comes from the more efficient minor collection of variable nursery.

In Figure 5(b), the throughput for variable nursery is increased while that of the other collector is not changed as the ratio becomes large. This implies that the lifetime pattern of threads that Thread-aware collector exploits is more distinct for larger dynamic requests ratio. In addition, as shown in Figure 6, only the number of minor collections for variable nursery increases for the large ratio in Thread-aware collector while the number of minor collections for fixed nursery is nearly constant. It means the performance of minor collection for variable nursery has more weight in total performance for large ratio of dynamic requests. Therefore, we expect that Thread-aware collector will work better for web application servers (WAS) than for web servers servicing only static pages.

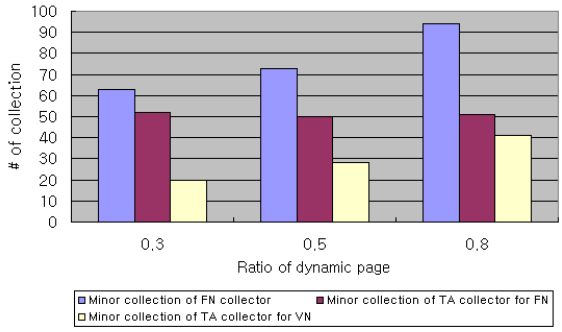


Figure 6. Number of minor collections for various types of workload

6. Related Work

Some JVMs use small thread-local nurseries but their purpose is to avoid excessive synchronization among threads during allocation of objects [12], [4]. On the other hand there is an approach that one heap is assigned to one thread and only exhausted heap is collected to avoid unnecessary stopping of other threads [11], [7]. In this scheme, objects in each thread-local heap should be *local* to the heap; they should not be used by objects in the other heaps and should not have references toward the other heap. To pick out global objects that is used by two or many threads, complex mechanisms such as static escape analysis [11] or runtime analysis [7] are used.

Our collector uses several heaps that are similar with thread-local heaps but assigns each heap for threads that have similar object lifetime patterns. In addition, it applies different strategies for each heap according to the characteristics of the heap to improve the efficiency of garbage collection. The thread-local heap approach do not care of differences of object lifetime and allocation rate among threads. If our idea combines with the thread-local heap, more efficient memory management is possible: scheduling collections for heaps and choosing the heap that has large garbage and few live objects.

There are many approaches to figure out and exploit lifetime patterns of objects. One approach of them is pretenur-

Proceedings of the 2004 International Symposium on Applications and the Internet (SAINT'04)
0-7695-2068-5/04 $20.00 © 2004 **IEEE**

ing [6], [5], [8] . In the approach, the memory manager tries to find out which objects have long lifetimes and allocate such objects directly in the mature space. Cheng et. al. [6] use profiling to determine allocation sites that tend to allocate long-lived objects. This profiling is then used to classify allocation sites. Harris [8] uses dynamic sampling to predict the lifetimes of objects. Sampling reduces the cost of obtaining statistics so that it avoids the need for a separate profiling phase and allows pretenuring decisions changed during runtime. Finally, Blackburn et. al. [5] improve pretenuring by combining profiling results from multiple applications for common library code with results for objects allocated by the runtime code.

Our idea and the pretenuring have a common property of using lifetime patterns of objects. On the other hand, while the granularity of lifetime patterns in pretenuring approach is an allocation site, that of our idea is a thread, based on the observation of the server architecture and behavior. In addition, our idea uses the lifetime patterns to allocate objects that have similar lifetime in one heap while pretenuring uses them to allocate long-lived objects directly to the mature space. Our collector shows that special management for short-lived objects is also as important as pretenuring of long-lived objects.

7. Conclusions

We propose and prove the different object lifetime patterns according to threads in server applications. Our Thread-aware collector exploits the patterns to gather short-lived objects altogether and applies different policies to each heap. As a result the throughput of GC is up to 1.7 times greater than that of the generational copying collector with the similar collection time. For the pause time, it is smaller by 8.4 - 10 % given the similar throughput. Consequently it will help server applications achieve both high throughput and small response time.

References

[1] *ISMM 2000*, volume 36(1) of ACM SIGPLAN Notices, Minneapolis, Minnesota, USA, 2001. ACM Press.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeño virtual machine. *IBM System Journal*, 39(1), February 2000.

[3] A. W. Appel. Simple generational garbage collection and fast allocation. *Software Practive and Experience*, 19(2), 1989.

[4] BEA Systems, Inc. *BEA WebLogic JRockit - The Server JVM*. Available at http://www.bea.com/products/weblogic/jrockit/.

[5] S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinley, and J. B. Moss. Pretenuring for java. In *ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 36(10) of ACM SIGPLAN Notices, Tempa, FL, 2001. ACM Press.

[6] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of SIGPLAN'98 COnference on Programming Languages Design and Implementation*, volume ACM SIGPLAN Notices, pages 162–173, Montreal, 1998. ACM Press.

[7] T. Domani, G. Goldshtein, E. K. Kolodner, and E. Lewis. Thread-local heaps for java. In *Proceedings of the Third International Symposium on Memory Management, ISMM '02*, volume 37 of ACM SIGPLAN Notices, Berlin, Germany, 2002. ACM Press.

[8] T. L. Harris. Dynamic adaptive pre-tenuring. In *Proceedings of the Second International Symposium on Memory Management, ISMM 2000* [1].

[9] B. Hayes. Using key object opportunism to collect old objects. In *ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 26(11) of ACM SIGPLAN Notices, Phoenix, Arizona, USA, 1991. ACM Press.

[10] Standard Performance Evaluation Corporation. *The SPECweb99 benchmark*. Available at http://www.spec.org/osg/web99/.

[11] B. Steensgaard. Thread-specific heaps for multi-threaded programs. In *Proceedings of the Second International Symposium on Memory Management, ISMM 2000* [1].

[12] Sun Microsystems, Inc. *The Java HotSpot™ Virtual Machine, v1.4.1*. Available at http://java.sun.com/products/hotspot/.

[13] Sun Microsystems, Inc. *Java™ 2 Platform, Enterprise Edition*. Available at http://java.sun.com/j2ee/.

[14] D. Ungar. Generation scarvenging: A non-disruptive high performance storage reclamation algorithm. In *Preceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environment*, Pittsburgh, Pennsylvania, 1984. ACM Press.

[15] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, Banff, Alberta, Canada, 2001. ACM Press.

Proceedings of the 2004 International Symposium on Applications and the Internet (SAINT'04)
0-7695-2068-5/04 $20.00 © 2004 **IEEE**