ISSN No. 0976-5697

Volume 4, No. 8, May-June 2013

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Memory Reclamation by Garbage Collectors: SPECjvm 2008

Nitan S. Kotwal
PG Department of Computer Science & IT
University of Jammu, India
nitankotwal@hotmail.com

Shubhnandan S. Jamwal
PG Department of Computer Science & IT
University of Jammu, India
jamwalsnj@gmail.com

Abstract: In earlier languages the memory management is done explicitly by the programmer himself. Now with the advent of modern object oriented languages like Java and C# the programmer is relived from explicitly managing the memory. A special program thread known as garbage collector takes care of managing the memory implicitly. The process of automatically reclaiming memory from dead objects (the objects that are not referenced from program or any other live object) is known as garbage Collection (GC). There are various metrics that affect the performance of the mutator. In the current research paper we have experimentally tested the four garbage collectors on various benchmarks of SPECjvm2008 and calculated how much memory is reclaimed after each (minor and major) collection.

Keywords: Collectors, Minor Collection, Major Collection, Mutator, Benchmarks, Memory Reclaimed.

I. INTRODUCTION

GC is the process of automatic memory reclamation in which memory is reclaimed from the dead objects and added to the pool of free memory. Garbage collectors are gaining importance in modern compilers. Languages like Java and C# have incorporated garbage collectors for automatic memory management. There are four garbage collectors in JDK 1.7.0.

The selection of a particular collector depends on the class of the machine. If the machine class is server then by default Parallel collector is selected. If the machine class is client the default collector is serial collector.

The other collectors are Parallel Old, Conc Mark Sweep.

We have also a choice to explicitly activate the garbage collector through command line.

a. ***Serial Collector:*** With *serial collector* both young and old generations are collected serially in a stop the world fashion and is usually adequate for small applications (requiring heap up to 100 mb). In this collector application execution is halted while collection is taking place [1].

b. ***Parallel Collector:*** With *parallel collector* minor collections are performed simultaneously while the major collections are performed serially. It is suitable for those applications that have large data sets. The *parallel collector* is appropriate on multiprocessor systems. It is selected by default on server-class machines. It can be enabled explicitly with option -XX:+UseParallelGC[1].

c. ***Parallel Old Garbage Collector:*** The *parallel compacting collector* was introduced in J2SE 5.0 update 6. With ParallelOld collector minor as well as major collections are performed parallel with the use of multiple CPU's in stop the world fashion. The difference between parallelOld collector and the parallel collector is that parallelOld collector uses a new algorithm for old generation garbage collection. It can be enabled explicitly with option -XX:+Use Parallel Old GC [1].

d. ***Concurrent Mark-Sweep (CMS) Collector:*** With CMS collector minor collection are performed in the same way as performed by the parallel collector. While major collection is done concurrently with the execution of the application. The CMS collector is appropriate if application needs shorter garbage collection pauses and can afford to share processors with the garbage collector thread when the application is running. It can be enabled explicitly with option -XX:+UseConcMarkSweepGC[1].

II. REVIEW OF LITERATURE

Sunil Soman and Chandra Krintz [2] showed that application performance in garbage collecting languages is highly dependent upon the application behavior and on underlying resource availability. Given a wide range of diverse garbage collection algorithms, no single system performs best across all programs and heap sizes. They further presented a Java Virtual Machine extension for dynamic and automatic switching between diverse, widely used GC for application specific garbage collection selection. Further they described a novel extension to extant on-stack replacement (OSR) mechanisms for aggressive GC specialization that is readily amenable to compiler optimization.

J. Singer, G. Brown, I. Watson, and J. Cavazos [3] after obtaining and analyzing the results found that the no single garbage collector based on different algorithm is best suited for the all the different types of application.

Clement R. Attanasio, David F. Bacon, Anthony Cocchi, and Stephen Smith [4] observed that when resources are sufficient, all the collectors behave in similar manner. But when memory is limited, the hybrid collector (using mark-sweep for the mature space and semi-space copying for the nursery) can deliver at least 50% better application throughput. Therefore

parallel collector seems best for online transaction processing applications.

Katherine Barabash, Yoav Ossia, and Erez Petrank [5] presented a modification of the *concurrent collector*, by improving the throughput of the application, stack, and the behavior of cache of the collector without foiling the other good qualities (such as short pauses and high scalability). They implemented their solution on the IBM production JVM and obtained a performance improvement of up to 26.7%, a reduction in the heap consumption by up to 13.4%, and no substantial change in the pause times (short). The proposed algorithm was incorporated into the IBM production JVM.

Tony Printezis, and David Detlefs [6] showed that the use of mostly-concurrent algorithm for older generation decreases pauses for old-generation collection for those programs whose promotion rates are sufficiently low to allow a collector thread running on a separate processor to meet its deadlines.The young generation collection is also slowed , but this slowdown can be more than offset by the offloading of collector work to the other processor.

Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley [11] analyzed that the overall performance of generational collectors as a function of heap size for each benchmark is mainly dictated by collector time. *Mark Sweep* does better in small heaps and Semi Space is the best in large heaps. But the results are not satisfactory in small memory. Garbage collection algorithms still trade for space and time which needs to be better balanced for achieving the high performance computing.

Stephen M Blackburn, Perry Cheng and Kathryn S McKinley [7], experimental design shows key algorithmic features and how they match program characteristics to explain the direct and indirect costs of garbage collection as a function of heap size on the SPEC JVM benchmarks. They find that the contiguous allocation of copying collectors attains significant locality benefits over free-list allocators. The reduced collection cost of the generational algorithms together with the locality benefit of contiguous allocation motivates a copying *nursery* for newly allocated objects. The above mentioned advantages dominate the overheads of generational collectors compared with non-generational.

Jurgen Heymann [8] presented an analytical model that compares all known garbage collection algorithms. The overhead functions are easy to measure and tune parameters and account for all relevant sources of time and space overhead of the different algorithms.

Kim, T., Chang, N., and Shin, H. [9] observed the memory management behavior of several Java programs from the SPECJVM98 benchmarks. The important observation is that the default heap configuration used in IBM JDK 1.1.6 results in frequent garbage collection and the inefficient execution of applications.

Dimpsey et al.[10] describe the IBM JDK version 1.1.7 for Windows. This is derived from a Sun reference JVM. The changes were incorporated in order to improve the performance of applications executing in server. Physical memory in the system was also taken into consideration.

They set the default initial and maximum heap size to values that are proportional to the amount of physical memory in the system. However, they do not explain what values are used or how they were chosen. They also make modifications to reduce the number of heap growths because they are quite costly in their environment. If the memory reclaimed after a garbage collection is less than 25% of physical memory or if the ratio of time spent collecting garbage to time spent executing the application exceeds 13%, the heap is grown by 17%. They report that ratio-based heap growth was disabled if the heap approached 75% of the size of physical memory, but they do not explain what was done. It was reported that when starting with an initial heap size of 2 MB, this approach increases throughput by 28% on the Volano Mark and pBOB benchmarks.

III. EXPERIMENTATION

A. Benchmarks:

The current research is carried on SPECjvm2008 benchmark suite. All the eleven benchmarks available in SPECjvm2008 are studied in real JVM and no simulators are being used in the experimentation. All the benchmarks specified in the SPECjvm2008 are executed over a wide range of heap size varying from 20 mb to 400 mb with an increment of 20 mb size. Each of the benchmark is executed 10 times in a fixed heap size and the arithmetic mean is obtained. The performance of the Serial, Parallel, ParallelOld, ConcMarkSweep collectors is measured over different heap sizes.

The Processor used in current research is Intel(R) Core(TM) Duo CPU T2250 @ 1.73GHz. 32 bit system with 2038 megabyte RAM. The frequency of the memory is 795MHz. The operating System used Microsoft Windows XP Professional Version 2002 Service Pack 2.

Java used for performing the tests is jdk1.7.0_04, Ergonomics machine class is client. JVM name is JavaHoTSpot(TM) Client VM in which the maximum heap size is estimated at 247.50 MB.

The issues considered for optimization in the current research are

B. Memory Reclamation:

Memory reclamation is defined as the process of freeing memory after each collection. Memory can be reclaimed after minor and major collection is over.

- ***a. Minor collection:*** When young generation fills up it causes minor collection. After minor collection the memory allocated to the objects in young generation are freed and added to the pool of free memory. But there are still some objects that are garbage (no longer alive) but that cannot be reclaimed. These objects are moved to tenured generation and sometimes may be referenced from the tenured or permanent generations.
- ***b. Major collection:*** Those objects that cannot be reclaimed after minor collection are reclaimed after major collection. The major collection occurs when the tenured generation fills up.

IV. RESULTS

A. Memory Reclamation by Minor Collection:

It has been observed that *Serial and ConcMarkSweep collectors are* reclaiming more memory in case of *Compiler, Compress, Crypto, and XML* benchmarks. *Serial collectors* is reclaiming more memory in *Sunflow* benchmark. Whereas *ConcMarkSweep collector* reclaims more memory in case of *Scimark.large and Xml* benchmarks. *Parallel and ParallelOld collectors* are reclaiming more memory in case of *Serial benchmark.* In rest of the benchmark the level of significance of difference is very less. In general for all the collectors the percentage
of memory reclamation increases with the size of heap. The result for memory reclamation in minor collection for *Serial, Parallel, ParallelOld, and ConcMarkSweep* collectors in Benchmarks of SPECjvm2008 is shown in "Fig. 1".

B. Memory Reclamation by Major Collection:

Serial and ConcMarkSweep collectors reclaims more memory in case of *Startup, compress, crypto, mpegaudio, scimark.small, serial, and sunflow* benchmarks. *Serial collector* is reclaiming more memory in case of *compiler, derby, scimark.large* benchmarks. While for *xml* benchmark the level of significance of difference is very less. In general for serial collector the percentage of memory reclamation by major collection increases relative to the increase in the size of the heap while for ConcMarkSweep collector, percentage of memory reclamation by major collection increases relative to the increase in the size of the heap except for *derby* where it decreases relative to the increase in the size of the heap. For parallel and parallelold collectors, percentage of memory reclamation by major collection decreases with the increase in the size of the heap except for *compiler, scimark.large, and xml* benchmarks whereas it increases with the increase in the size of heap. The results are shown in "Fig. 2".

V. CONCLUSION

It is observed that if the size of the heap is increased the percentage of memory reclaimed after minor collection also increases for all the collectors. But in case of major collection if we increase the size of heap for serial collector, it is observed that for most of the benchmarks memory reclaimed after major collection increases. This is also true in case of concmarksweep except for *derby.* In case of *parallel and parallelold collectors* for most of the benchmarks memory reclaimed after major collection decreases as the heap size increases except for *compiler, scimark.large,* and *xml.* From the results obtained we conclude that the memory reclaimed after minor and major collection are not related to one another. We also wish to perform these tests for DaCapo-9.12-bach benchmark suite.

VI. REFERENCES

[1]. Sun Microsystems (2006) "Memory Management in the Java HotSpot Virtual Machine". [Online]. Available: http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf

[2]. S. Soman and C. Krintz, "Application-specific Garbage Collection", J. of Sys. and Software, Elsevier Science Inc. New York, NY, USA, vol. 80, No. 7, pp. 1037-1056, July 2007.

[3]. J. Singer, G. Brown, I. Watson, and J. Cavazos, "Intelligent Selection of Application-Specific Garbage Collectors", In Proc. Of the 6th International Symposium on Memory Management, ACM New York, USA, pp. 91-102, 2007.

[4]. C. R. Attanasio, D. F. Bacon, A. Cocchi, and S. Smith, "A Comparative Evaluation of Parallel Garbage Collector and Implementations", LCPC'01 Proc. of the 14th Int. Conf. on Languages and Compilers for Parallel Computing, Springer-Verlag Berlin, Heidelberg, LNCS 2624, pp. 177–192, 2003.

[5]. K. Barabash, Y. Ossia and E. Petrank, "Mostly Concurrent Garbage Collection Revisited", OOPSLA '03 Proc. of the 18th Annual ACM SIGPLAN Conf. on Object-Oriented Prog., Systems, Languages, and App., pp. 255-268, ACM New York, NY, USA , 2003.

[6]. T. Printezis, and D. Detlefs, "A Generational Mostly Concurrent Garbage Collector", In Proc. Of the 2nd International Symposium on Memory Management, ACM New York, USA, vol. 36, no. 1, pp. 143-154, Jan. 2001.

[7]. S. M. Blackburn, P. Cheng and K. S. McKinley, "Myths and Realities: The Performance Impact of Garbage Collection", Proc. of the Joint Int. Conf. on Measurement and Modeling of Compu. Sys., June 12–16, ACM Press, New York, NY, USA, 2004.

[8]. J. Heymann, "A Comprehensive Analytical Model for Garbage Collection Algorithms", ACM SIGPLAN Notices, vol. 26, No. 8, August 1991.

[9]. Kim, T., Chang, N., and Shin, H., "Bounding Worst Case Garbage Collection Time for Embedded Realtime Systems", RTAS '00 Proc. of the Sixth IEEE Real Time Tech. and Appl. Symp.(RTAS 2000), pp. 46, IEEE Compu. Society Washington, DC, USA, 2000.

[10]. Dimpsey, R., Arora, R., Kuiper, K., "Java Server Performance: A Case Study of Building Efficient Scalable Jvms", IBM Systems J., vol. 39, No. 1, pp. 151-174, 2000.

[11]. O. Agesen and D. L. Detlefs. "Finding References in Java Stacks", Submitted to OOPSLA'97 Workshop on Garbage Collection and Memory Manag., Atlanta, GA, October 1997.

Figure. 1 Memory Reclamation after minor collection in Benchmarks of SPECjvm2008.

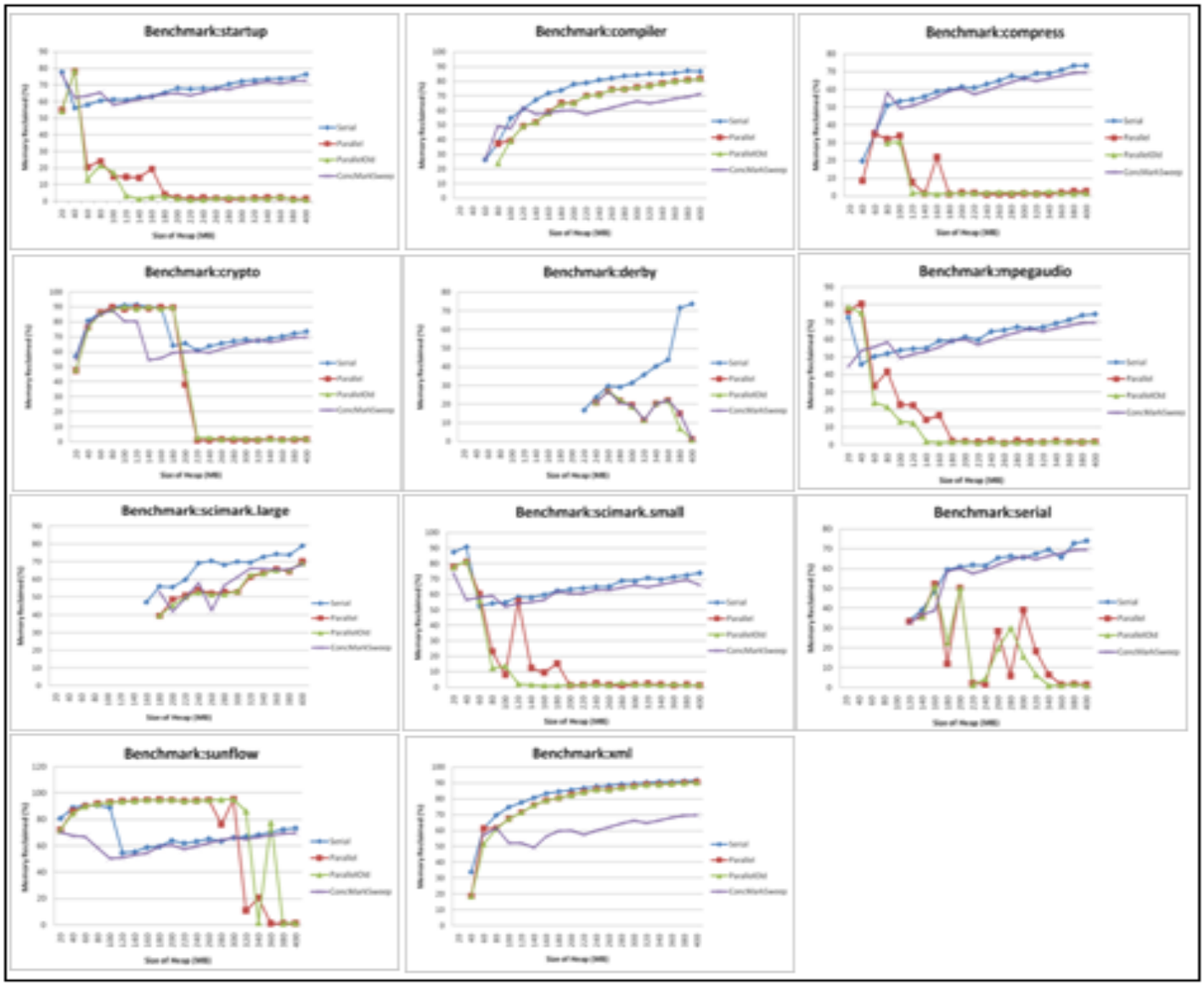

Figure. 2 Memory Reclamation after major collection in Benchmarks of SPECjvm2008.