Empirical Analysis of Serial and Parallel Collectors: SPECjvm2008

[1]Shubhnandan S. Jamwal, [2]Nitan S. Kotwal
[1,2]Dept. of Computer Science & IT, University of Jammu, Jammu, India

Abstract
Garbage Collection (GC) is a process of automatic memory reclamation from the objects that are no longer required by the mutator. The execution time of the application and memory reclaimed by garbage collector are important factors that influence the selection of a specific garbage collector while selecting a specific one. The current research paper selects a suitable garbage collector on the basis of the above mentioned two factors. In this paper the execution time of SPECjvm2008 benchmarks and memory reclaimed by a garbage collector while executing these application is determined in real JVM. We further propose the optimal values of the above two issues that can be compromised, if the an application is to be executed with the garbage collector.

Keywords
Serial, Parallel, Minor Collection, Major Collection, Mutator, Benchmarks.

I. Introduction

GC is the process of automatic memory reclamation. Garbage collectors are becoming the essential part of compilers. Most of the high level languages like Java and C# have incorporated garbage collectors for automatic memory management. JDK 1.7.0 and higher versions provide multiple garbage collectors.
Selection of a particular collector depends on the choice of the class of the machine between server VM or client VM on which the application is to be executed. The default choice for server class VM is Parallel collector (-XX:+UseParallelGC) and the default choice for client class VM is Serial collector (-XX:+UseSerialGC). However users can select a particular garbage collector depending on the type of application [2]. There is less or negligible effect of garbage collectors on applications that have very small amount data (kilobytes) but applications with large amount of code in execution and data (gigabytes, terabyte's), the selection of a particular garbage collector affects the mutator. Amdahl [1] observed that much of the workload cannot be perfectly parallelized; some portion is always sequential and does not benefit from parallelism. This is also true for the Java™ platform.

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 [2].

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 [2].
There are several issues in GC. Memory Reclamation and Mutator Execution time are the two important issues to be discussed.

II. Review Of Literature

Sunil Soman and Chandra Krintz [3] 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.
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.
Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley [6] 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 VolanoMark and pBOB benchmarks.

III. Experimentation

A. Benchmarks

SPECjvm2008 benchmark suite is used in the current research. All the eleven benchmarks available in SPECjvm2008 are studied in real JVM. No simulators are being used in the experimentation. All the eleven 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 and Parallel collector 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 are over.

1. 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. 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.

2. 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.

C. Mutator Execution Time

It is defined as the time spent by the mutator/application during its execution. The mutator execution time should be less.

IV. Results

A. Memory Reclamation by Minor Collection

It has been observed that Serial collector is reclaiming more memory in case of Startup, Compiler, Compress, Scimark.large, Sunflow, and XML benchmarks whereas Parallel collector reclaims more memory in case of Derby, Serial benchmarks. In rest of the benchmark the level of significance of difference is very less. In general for both serial and parallel collectors the percentage of memory reclamation increases with the size of heap. The result for memory reclamation in minor collection for Serial and Parallel collectors in Benchmarks of SPECjvm2008 is shown in fig. 1.

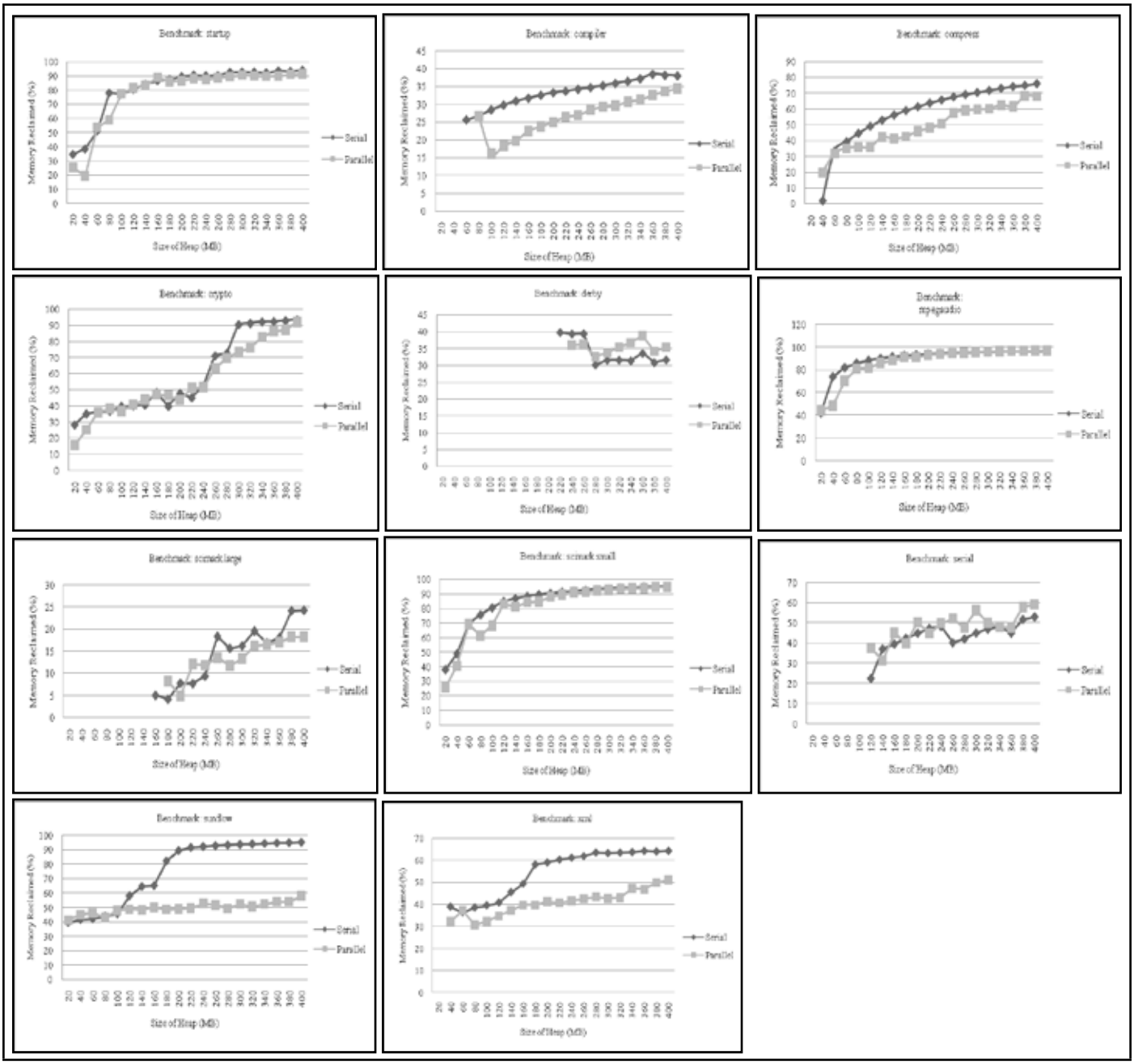


Fig. 1: Memory Reclamation After Minor Collection for Serial and Parallel Collectors in Benchmarks of SPECjvm2008

B. Memory Reclamation by Major Collection

Serial collector reclaims more memory in case of startup, compiler, compress, derby, mpegaudio, scimark.large, scimark.small, and serial benchmarks. While for crypto, sunflow, and xml benchmarks 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 parallel collector, 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 decreases with the increase in the size of heap. The results are shown in fig. 2.

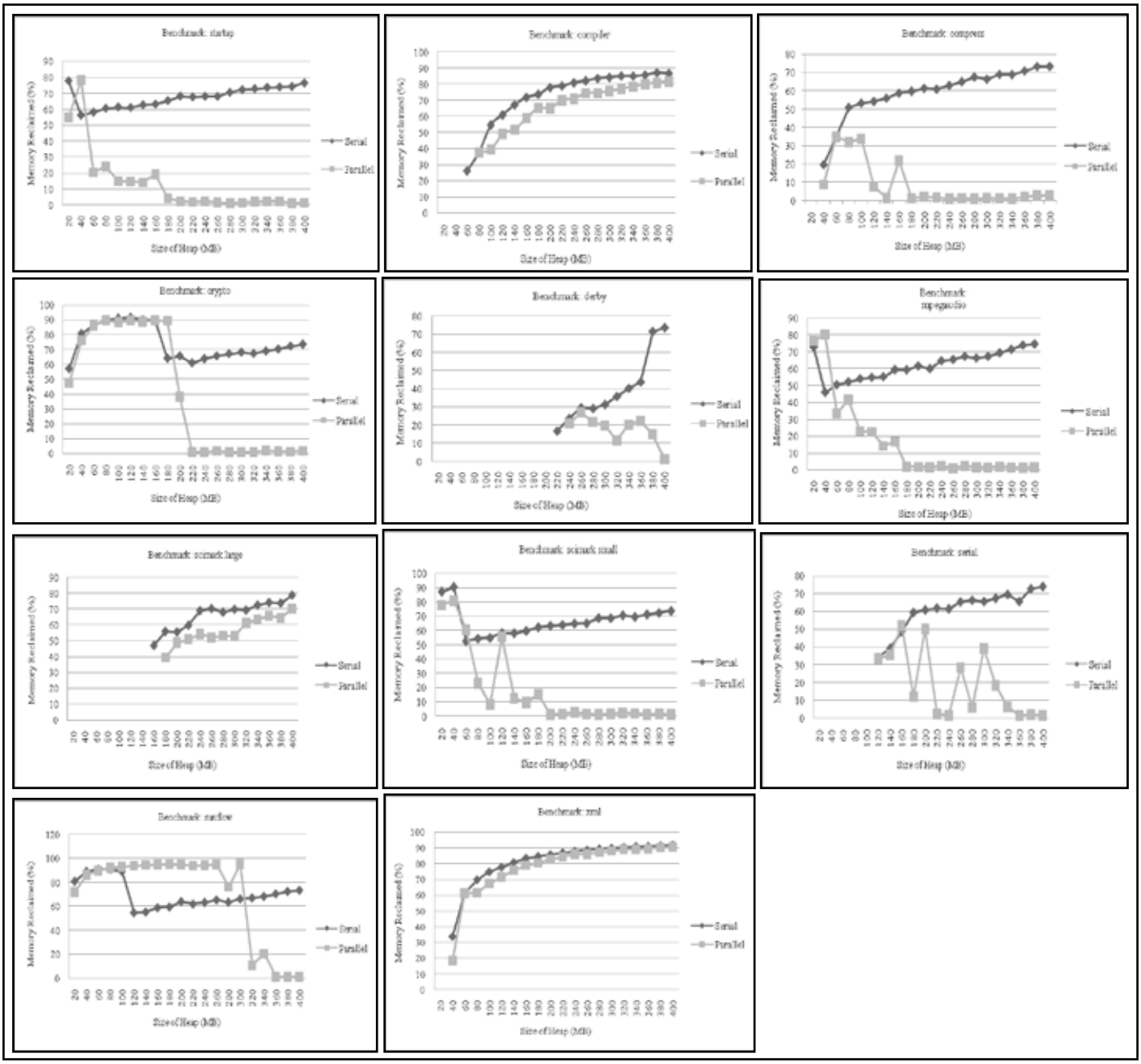


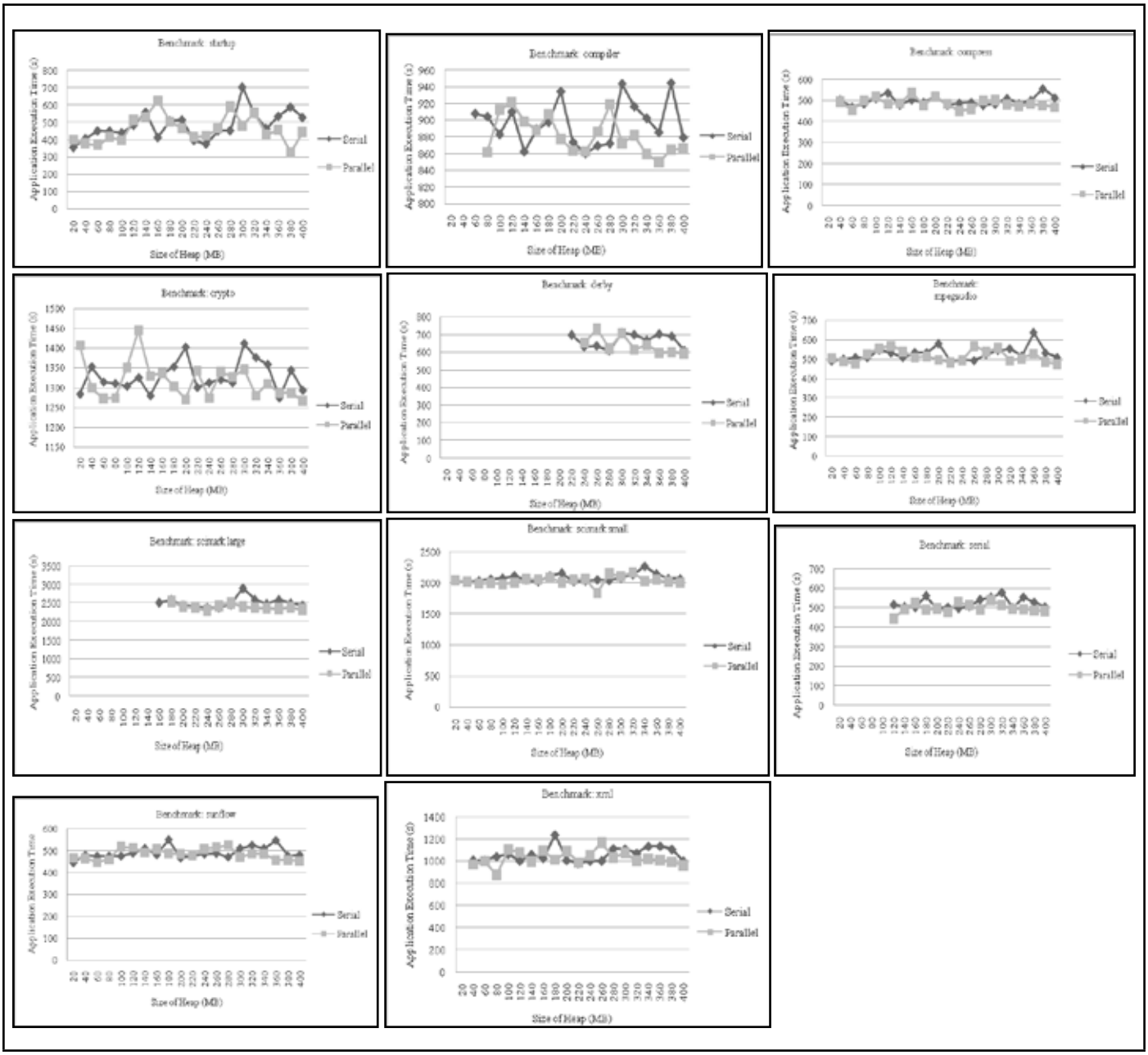
Fig. 2: Memory Reclamation After Major Collection for Serial and Parallel Collectors in Benchmarks of SPECjvm2008.

B. Mutator Execution Time

The application in execution takes less time with parallel collector in all the benchmarks of SPECjvm2008 as compared to serial collector and the performance in all the benchmarks is depicted in the fig. 3.

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 both serial and parallel collector. 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 except for crypto and sunflow. In case of parallel collector for most of the benchmarks memory reclaimed after major collection decreases except for compiler, scimark.large, and xml. In case of parallel collector, the execution time of the application for all the benchmarks of SPECjvm2008 is less as compared to serial collector. From the results obtained we conclude that the memory reclaimed after minor and major collection and application execution time are not related to one another. The execution time of an application is not related to the size of heap. We also wish to perform these tests for all the garbage collectors in jdk1.7.0_4 for DaCapo-9.12-bach benchmark suite.



References

[1] Sun Microsystems (2003),"Tuning Garbage Collection with the 5.0 Java [tm] Virtual Machine", [Online].Available: http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html

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

[3] S. Soman, 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.

[4] C. R. Attanasio, D. F. Bacon, A. Cocchi, 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, 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] O. Agesen, D. L. Detlefs,"Finding References in Java Stacks", Submitted to OOPSLA'97 Workshop on Garbage Collection and Memory Manag., Atlanta, GA, October 1997.

[7] S. M. Blackburn, P. Cheng, 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., 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.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.