Tuning Serial and Parallel Collectors in SPECjvm2008

Nitan S. Kotwal[1] , Shubhnandan S. Jamwal[2]

[1, 2] PG Department of Computer Science & IT, University of Jammu, Jammu, India
[1]*nitankotwal@hotmail.com*, [2]*jamwalsnj@gmail.com*

ABSTRACT *Many object-oriented programming languages like Java and C# are gaining importance in the development of reliable and robust applications because of the presence of Garbage Collectors. There are multiple Garbage Collectors (GC) available in JDK 1.7.0_04. In the current research we compared the performance of serial and parallel collectors with respect to SPECjvm2008. It has been found that the behavior of the Garbage Collector is application specific. Not all the benchmarks require same size of heap for its invocation. There is considerable amount of difference in heap size for initialization of Garbage Collector. The value of optimal size of the heap varies from 100 megabyte (mb) to 400 mb for different benchmarks. In the current research, benchmarks available in SPECjvm2008 are considered over a wide range of Heap size.*

1. KEYWORDS

Serial, Parallel, Nunber of Pauses, Average pause time, Throughput, Benchmarks.

2. INTRODUCTION

The Java HotSpot™ virtual machine provides 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 data (gigabytes, terabyte's), the selection of a particular garbage collector affects the mutator. Amdahl [1] observed all workload cannot be perfectly parallelized and some portion of work should always be sequential because it does not benefit from parallelism.

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].

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.

3. 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

© 2013 Vidya Publications. **Authors are responsible for any plagiarism issues.**

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.

4. EXPERIMENTATION

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

ISSN: 2229-6913 (Print), ISSN: 2320-0332 (Online) -, Web Presence: http://www.ijoes.vidyapublications.com
© 2013 Vidya Publications. **Authors are responsible for any plagiarism issues.**

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

Pauses
Pauses are defined as the temporary suspension of the mutator so that the garbage collection can take place. Pauses can be of two types

1) Minor: Initially the objects are allocated in the young generation and when this generation fills up, it causes minor collection.
2) Major: The objects which survive after the minor collection are moved to the tenured generation. When this generation fills up major collection is invoked.

Throughput
Throughput is defined as the percentage of the total time not spent in garbage collection. The throughput of an application should be more. It is calculated with the help of the following formula.
Throughput = mutator time / (mutator time + garbage collection time) *100

Pause Time in Minor Collection
It is defined as the time interval in minor collection during which a garbage collector is removing unreferenced objects from young generation. During minor pauses the mutator/application is suspended temporarily for a short period of time. It occurs when the young generation fills up.

Pause Time in Major Collection
It is defined as the time interval in major collection during which a garbage collector is collecting garbage from tenured generation. During major pauses the mutator/application is suspended for long as compared to minor pauses. It occurs when the tenured generation fills up.

5. RESULTS

Minor Pauses
Serial collector performs better in case of *Startup, Compiler, Compress, Scimark.large and XML.* Parallel collector performs well in case of Serial benchmark. In rest of the benchmark the level of significance of difference is very less. In general the number of minor pauses in Serial collector is less than the number of pauses in Parallel collector in each benchmark except in the case of Serial benchmark. The benchmark such as startup, crypto, mpegaudio, scimark.small, and sunflow initialize at 20 mb heap size. While other benchmark such as compress and xml need 40 mb heap size to initialize. Compiler benchmark is initialized at 60 mb heap size. Benchmark Serial is initialized at 120 mb. For Serial collector benchmark scimark.large is initialized at160 mb while for Parallel collector it is initialized at180 mb. It is the derby benchmark which takes 220 mb heap size in case of serial collector and 240 mb heap size in case of parallel collector to initialize. The result for minor pauses is shown in “Fig. 1”.

Pause Time in Minor Collection
In general the pause time for minor collection in all the benchmarks for parallel collector is less as compared to serial collector except for compress benchmark. The result is shown in “Fig. 2”.

Major Pauses

© 2013 Vidya Publications. Authors are responsible for any plagiarism issues.

In general the numbers of major pauses for serial and parallel collector are nearly same for all the benchmarks. The result is shown in “Fig. 3”.

Pause Time in Major Collection
In general for small heap sizes the pause time in major collection for serial is less but for the large heap sizes the parallel collector is performing better than serial collector. The result is shown in “Fig. 4”.

Throughput
In general the throughput of serial and parallel collector is nearly same in all the cases except for Compiler, compress, scimark.large benchmarks where the difference between throughput is also minor. The result is shown in “Fig. 5”.
Let P_{1s} and P_{1p} be the number of minor pauses in serial and parallel collector respectively. P_{2s} and P_{2p} be the number of major pauses in serial and parallel collector respectively.
Let P_{T1s} and P_{T1p} be the average pause time of minor collection in serial and parallel collector respectively. P_{T2s} and P_{T2p} be the average pause time of major collection in serial and parallel collector respectively.
From figure 1 and figure 2

$P_{1s} < P_{1p}$ and $P_{T1s} > P_{T1p}$

Although Parallel collector stops the mutator for more number of times as compare to Serial collector. But the average pause time taken by each collection in case of parallel collector is less as compared to serial collector. If there is restriction on pause time (e.g. interactive application) we should use parallel collector. Otherwise we can use serial or parallel collector.

From figure 3 and Figure 4

For small heap size

$P_{2s} < P_{2p}$, $P_{T2s} < P_{T2p}$

As the heap size increases

$P_{2s} \approx P_{2p}$, $P_{T2s} \approx P_{T2p}$

From above we deduct that if there is constraint on memory then we should use serial collector otherwise we can use serial or parallel collector.

6. CONCLUSION

Using the results we conclude that startup benchmark should be executed at 100 mb heap size and Serial Collector should be invoked for this benchmark. Compiler benchmark gives optimal performance at 220 mb heap size and Serial Collector should be invoked for it. Compress benchmark gives optimal performance at 240 mb and Serial collector should be invoked for it. Crypto benchmark gives optimal performance at 240 mb and Parallel Collector should be invoked for it. Scimark.large benchmark gives optimal performance at 400 mb and Serial Collector should be invoked for this benchmark. Derby, mpegaudio, scimark.small, serial, sunflow, and xml benchmarks gives optimal performance at 400 mb and Serial Collector should be invoked for this benchmark. We also purpose to optimize the ParallelOld and ConcurrentMarkSweep Future GC for SPECjvm2008. We also wish to find the optimal values for all the GC for DaCapo-9.12-bach benchmark suite.

© 2013 Vidya Publications. Authors are responsible for any plagiarism issues.

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://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf

[3] 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.

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

[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.

© 2013 Vidya Publications. **Authors are responsible for any plagiarism issues.**

Figure 1. Number of Minor Pauses for Serial and Parallel Collectors in Benchmarks of SPECjvm2008.

***© 2013 Vidya Publications.* Authors are responsible for any plagiarism issues.**

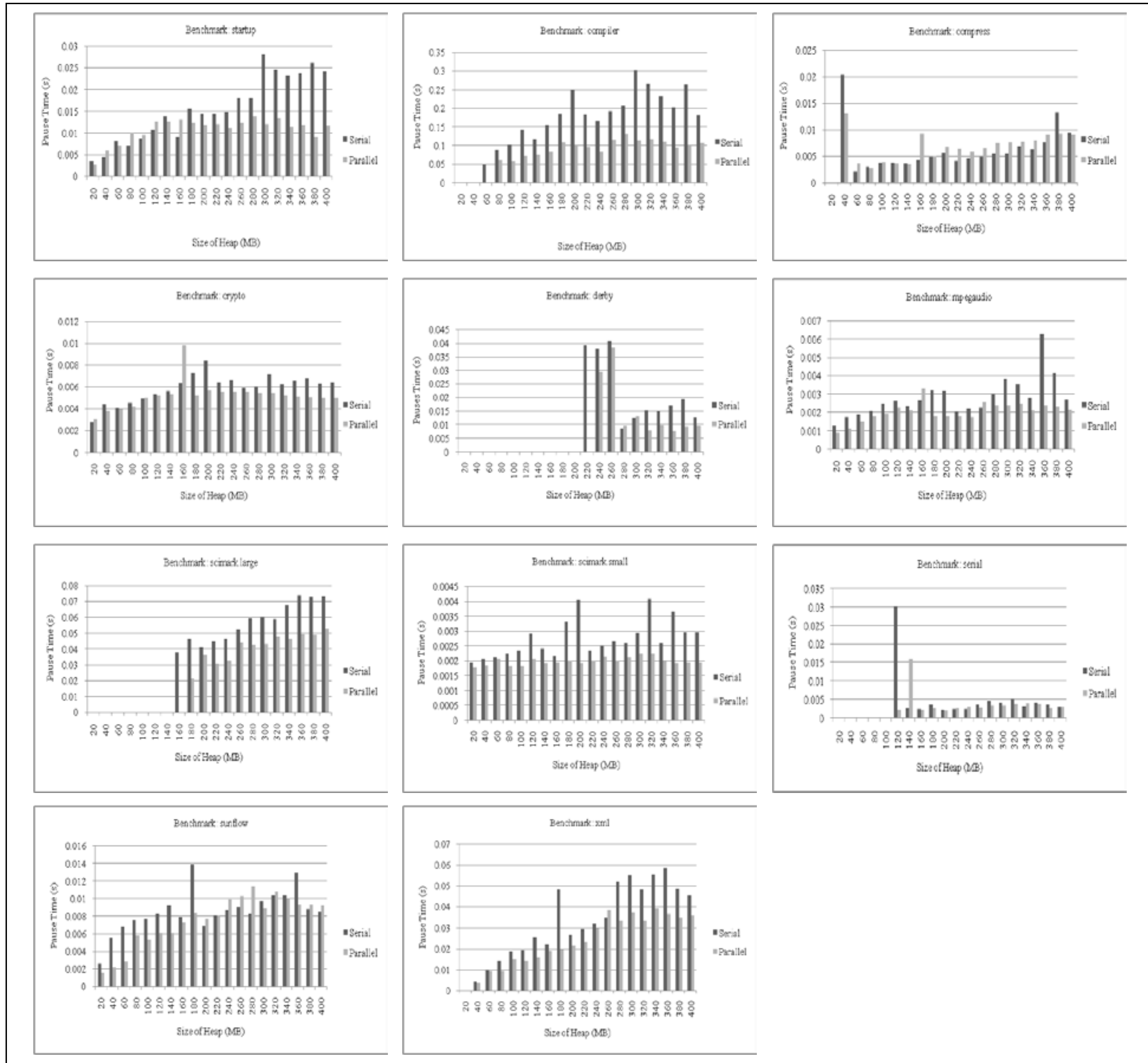

Figure 2. Pause Time in Minor Collection for Serial and Parallel Collectors in benchmarks of SPECjvm2008.

ISSN: 2229-6913 (Print), ISSN: 2320-0332 (Online) -, Web Presence: http://www.ijoes.vidyapublications.com
***© 2013 Vidya Publications.* Authors are responsible for any plagiarism issues.**

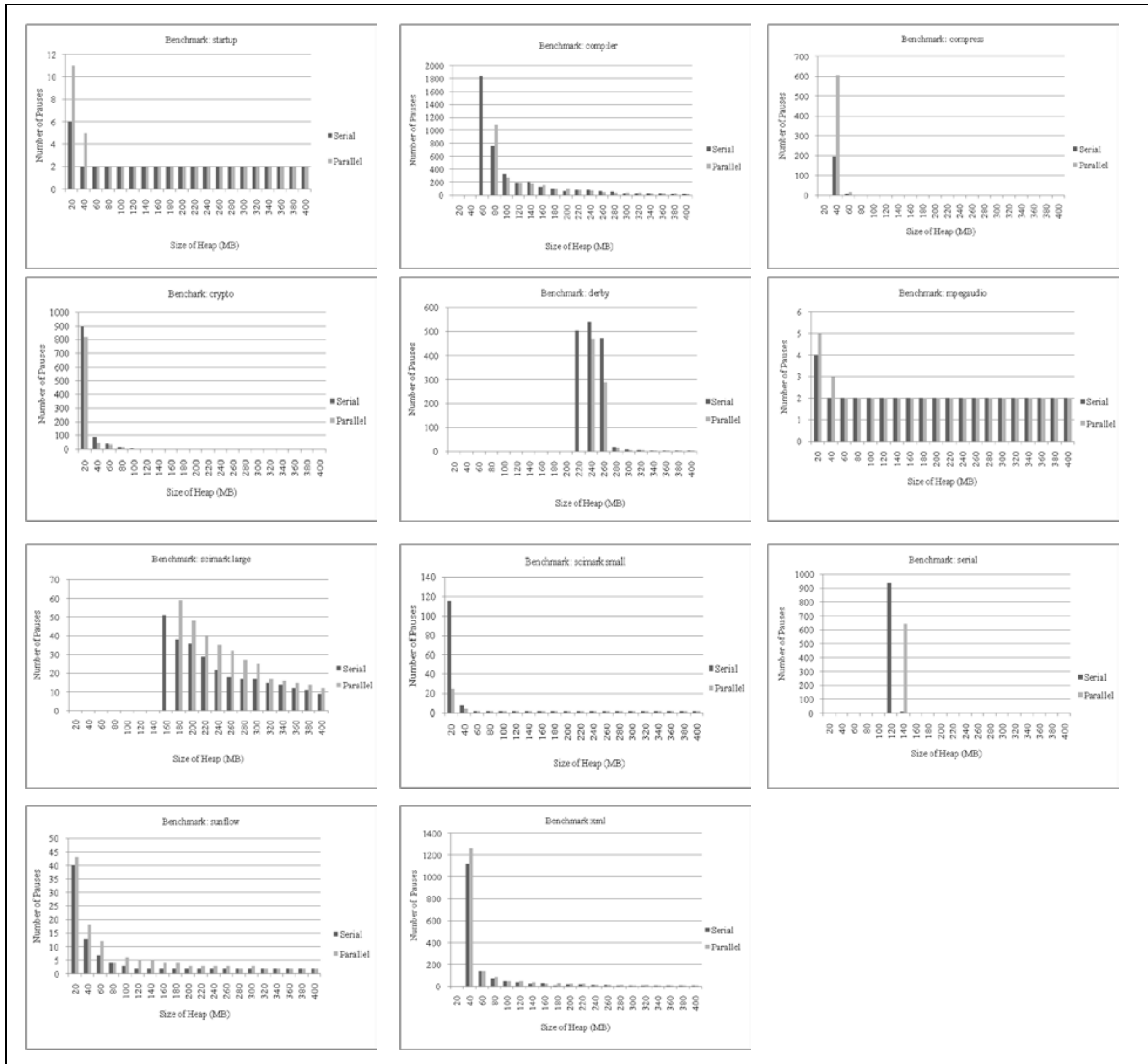

Figure 3. Number of Major Pauses for Serial and Parallel Collectors in Benchmarks of SPECjvm2008.

***© 2013 Vidya Publications.* Authors are responsible for any plagiarism issues.**

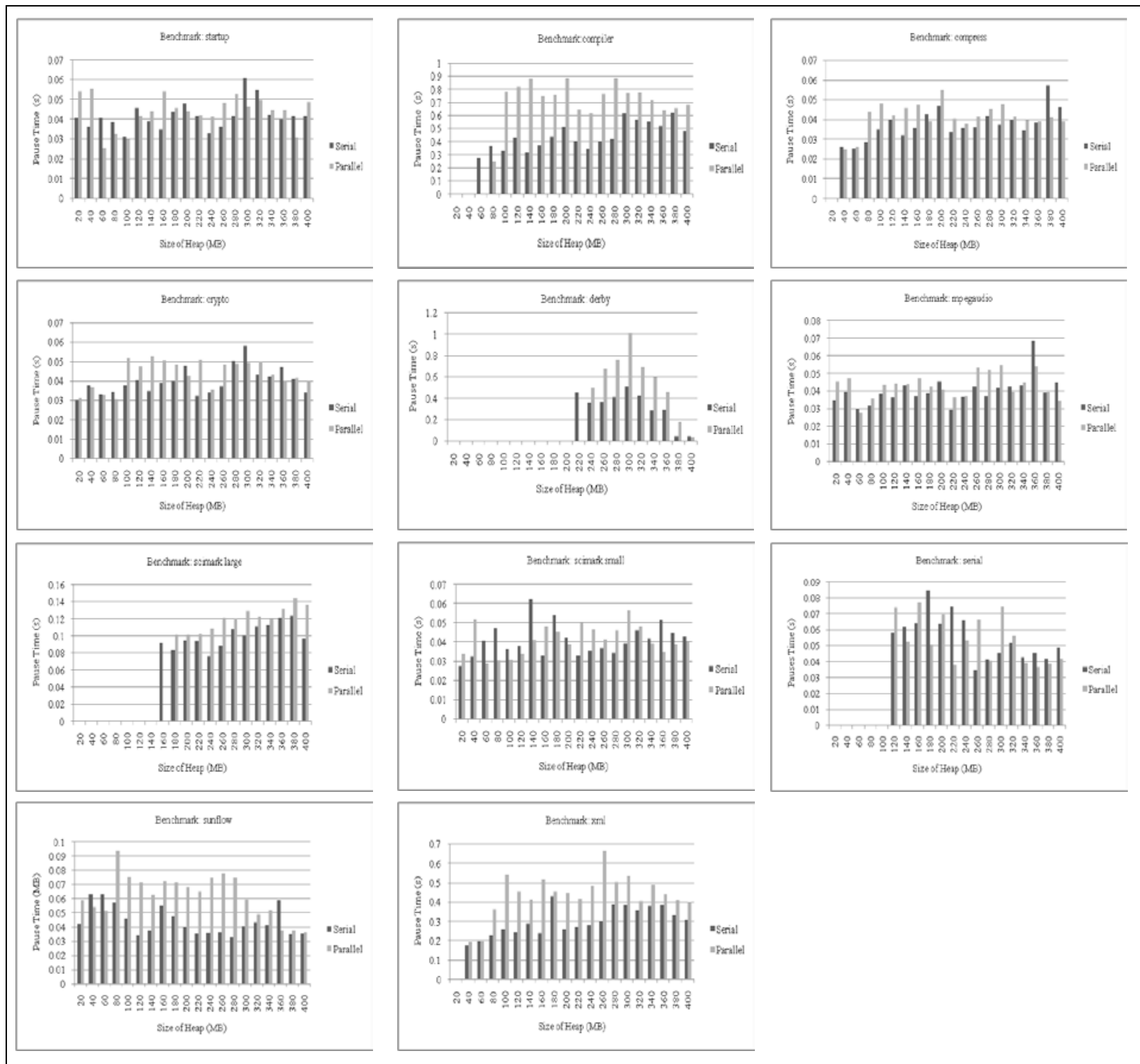

Figure 4. Pause Time in Major Collection for Serial and Parallel Collectors in benchmarks of SPECjvm2008.

ISSN: 2229-6913 (Print), ISSN: 2320-0332 (Online) -, Web Presence: http://www.ijoes.vidyapublications.com
© 2013 Vidya Publications. **Authors are responsible for any plagiarism issues.**

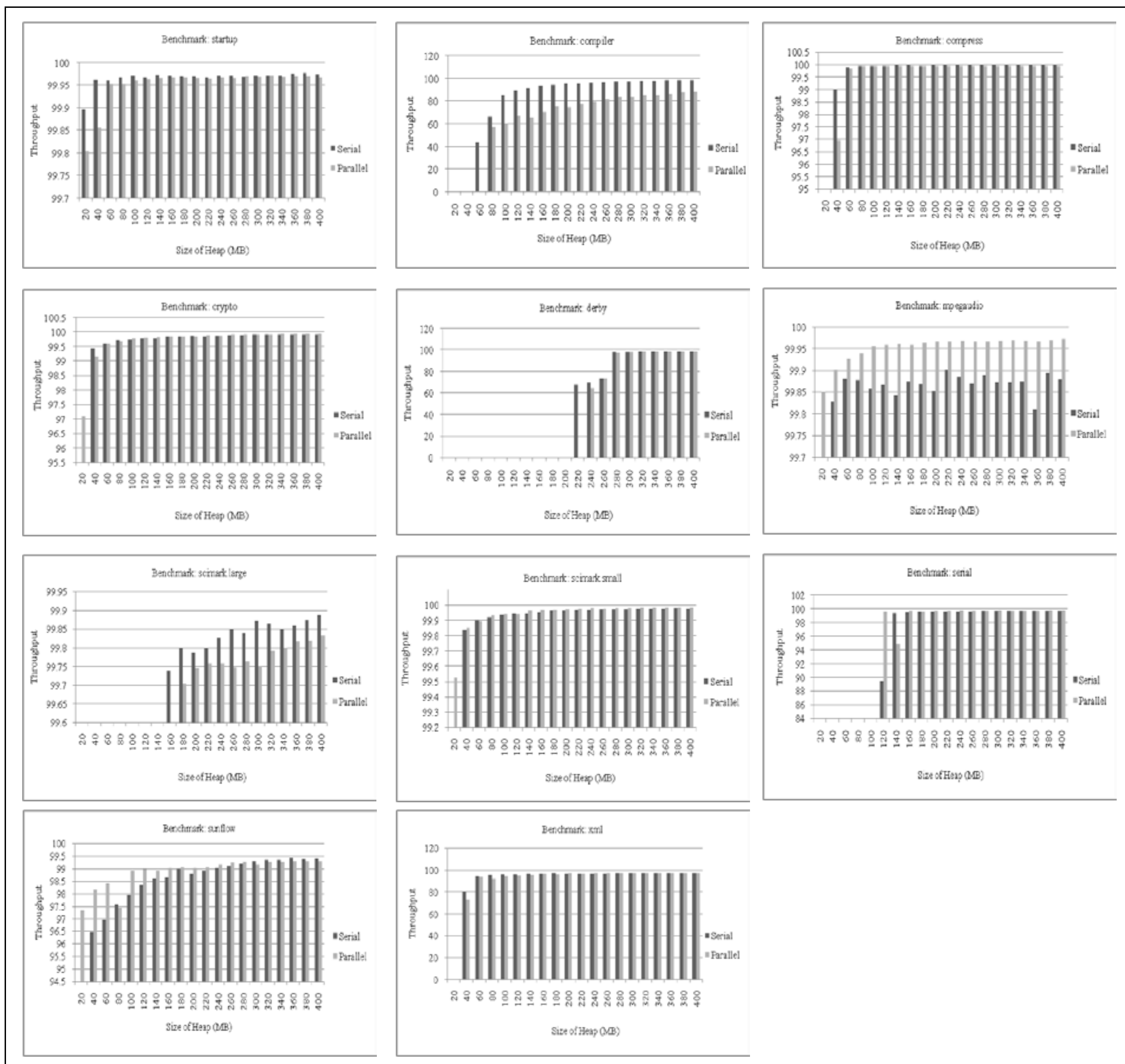

Figure 5.Throughput of Serial and Parallel Collectors in benchmarks of SPECjvm2008.

ISSN: 2229-6913 (Print), ISSN: 2320-0332 (Online) -, Web Presence: http://www.ijoes.vidyapublications.com
***© 2013 Vidya Publications.* Authors are responsible for any plagiarism issues.**