IJITKM 0973-4414 Volume-8, Number-2 Jan-Jun 2015 pp. 225-228 **Impact Factor 2.8** available online at www.csjournalss.com

Garbage Collection Time for Minor Collection

Shubhnandan S Jamwal, Nitan Singh
Department of Computer Science & IT, University of Jammu, Jammu, India
Jamwalsnj@gmail.com

Abstract: Garbage Collection (GC) is the process of automatic memory reclamation in which those objects which are no longer referenced by program or any other live object are collected. The memory occupied by these objects is freed and added to the pool of free memory. The programming thread that is responsible for GC is known as Garbage Collector. There are various metrics that affect the performance of the application. Time taken by minor collection is one of them. In the current research paper we have experimentally tested the four garbage collectors on various benchmarks of SPECjvm2008 and calculated how much time is required to perform minor collection.
Keywords: Benchmark, garbage collector, minor collection, pauses.

Introduction

Automatic memory management is gaining importance in the high level languages. Many high level languages like Java, C# have incorporated garbage collectors for automatic memory management. There are four garbage collectors in Java. These are Serial, Parallel, Parallel Old, and Concurrent Mark Sweep garbage collectors. A heap in java is divided into three regions. The regions are young generation, old generation and permanent generation. Initially objects are allocated to young generation. While those objects that survive two or more minor collections are promoted to old generation. Some large objects may also be allocated directly in the old generation. Permanent generation is a non-heap memory area that contains data of the virtual machine itself, such as classes, java methods and reference objects.

Young generation **:** The young generation is further divided into three regions: one *eden* and two *survivor spaces from space* and *to space*. Initially objects are allocated to *eden* and one of the survivor space (*from space*). One of the *survivor space (to)* is empty all the time. Those objects that are large in size and cannot be accommodated in the young generation are allocated to the old generation.

When garbage collector is running to collect the garbage (dead objects) in the young generation, the application is stopped. Garbage collection time is defined as the time spent in collecting the garbage. During garbage collection the application is stopped and the garbage collector is running. During young generation collection application is paused to collect minor collection. While during old generation collection application is paused to collect major collection.

Review of Literature

Sunil Soman and Chandra Krintz [1] 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. Katherine Barabash, Yoav Ossia, and Erez Petrank [2] 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 [3] 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. 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 [2] 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 [3] 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 [5], 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 freelist 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 [6] 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. [7] 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.

Experimentation

Benchmarks

We have used SPECjvm2008 benchmark suite in the current research. All the eleven benchmarks available in SPECjvm2008 are studied in real JVM and executed and 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, Parallel, Parallelold and Concurrent mark sweep 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.

Garbage Collection time for minor collection

It is defined as the time spent in collecting the garbage due to minor collection. When garbage collector is running in the young generation to collect the garbage, the application is paused during that time. The garbage collection time for minor collection should be as short as possible.

Conclusion/Future Work

It has been observed that if the size of the heap is increased, the time spent in collecting the garbage due to minor collection is decreased for all the garbage collectors except for derby and scimark.large benchmarks. But in case derby and scimark.large benchmarks the garbage collection time due to minor collection increases with the increase in the size of heap. The result for time spent garbage collection for minor collection is shown in figure 1. In future work we would find the time spent by all garbage collectors on all the benchmarks of SPECjvm2008, due to major collection

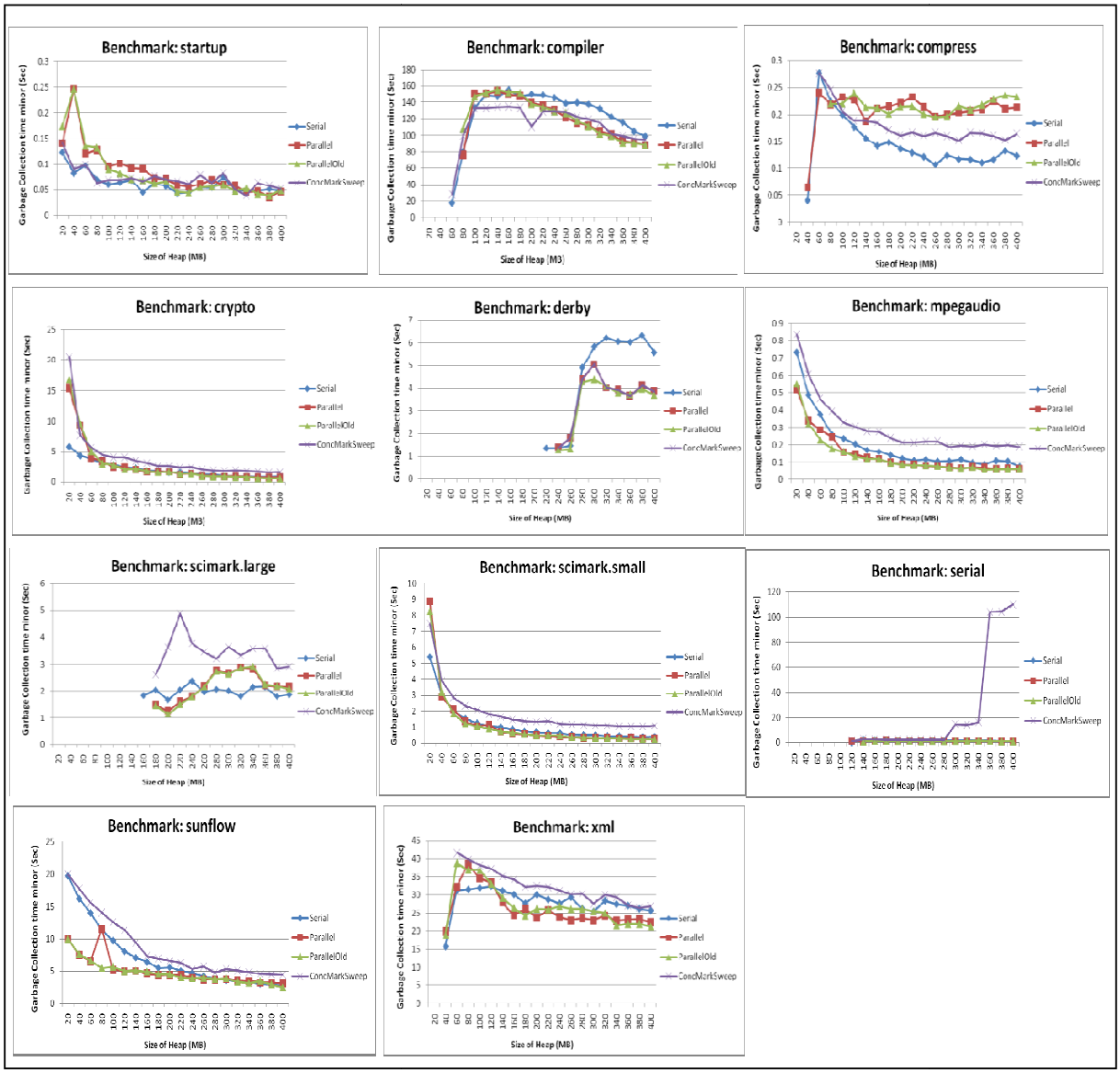

Figure 1. Time spent by garbage collectors for all the benchmarks of SPECjvm2008 due to minor collection .

References

[1]. 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.

[2]. 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.

[3]. 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.

[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]. 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.

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

[7]. 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.