International Journal of Information Technology and Knowledge Management
July-December 2010, Volume 2, No. 2, pp. 487-490

PARALLEL AND SERIAL GARBAGE COLLECTOR IN MULTITHREADED APPLICATIONS: A QUANTITATIVE ANALYSIS

Shubhnandan S. Jamwal[1] & Devanand[2]

The performance of garbage collector is largely dependent upon application execution behavior. We have taken the multithreaded applications for study in this paper. The execution of the threads is done in small stack sizes of 4mb, 8mb and 16mb. We observed the behavior of two collectors with respect to the number of pauses and application execution time in the multithreaded application. It is observed that the number of pauses in case of serial garbage collector remains more as compared to parallel garbage collector. If we look at the average of all the three multithreaded applications in case of 4mb stack the average number of pause are 9.5 and 1.8 for serial garbage collector and parallel garbage collector respectively. For 16mb stack the average number of pause are 4 and 1.9 for serial garbage collector and parallel garbage collector respectively. In paper we further observed that the average time of execution of all three applications in 4mb of stack with serial garbage collector is 366.73ms and with the parallel collector the average is 95.86ms and the average of all the application in 16mb of stack with serial garbage collector is 201.66ms and with the parallel collector the average time is 103.2ms.

Keywords: Garbage Collector, Serial, Parallel, Stack, Multithreaded, Virtual

Introduction

Garbage collection plays an increasingly important role in next generation Internet computing and server software technologies. However, the performance of collection systems is largely dependent upon application execution behavior and resource availability. When an object is no longer referenced by the program the heap space it occupies must be recycled so that the space is available for subsequent new objects. The Garbage collector must some how determine which objects are no longer referenced by the program and make available the heap space occupied by such unreferenced objects. In addition to freeing unreferenced objects, the garbage collector must also combat heap fragmentation. Heap fragmentation occurs through a course of normal program execution. New objects are allocated and unreferenced objects are freed such that the free blocks of the heap memory are left in between blocks occupied by live objects. Request to allocate new objects may have to be filled by extending the size of the heap even though there is enough unused space available in the existing heap.

One of the advantages of garbage collection is that the garbage collection ensures program integrity. Garbage collection is an important part of Java's security strategy. Java programmers are unable to accidentally crash the finalize and free unreferenced objects on the fly, but the programmers in the garbage collected environment have less control over the scheduling of the CPU time devoted to freeing objects that are no longer needed.

[1]Asst. Professor, email: jamwalsnj@gmail.com

[2]Professor and Head, email: dpadha@rediffmail.com

PG Department of Computer Science and IT, University of Jammu

Literature Review

Kim et al.[1] analyze the memory system behavior of several Java programs from the SPECJVM98 benchmark suite. One of the observations made in their work is that the default heap configuration used in IBM JDK 1.1.6 results in frequent garbage collection and the inefficient execution of applications. Although the direct overheads due to garbage collection in their environment appear to be more costly than in ours (both because the entire heap is swept on each collection and heap compaction is used), we believe that their results also demonstrate the need to improve techniques for controlling garbage collection and heap growth.

Dimpsey et al.[2] describe the IBM DK version 1.1.7 for Windows. This is derived from a Sun reference JVM implementation, and changes were made in order to improve the performance of applications executing in server environments. Their approach also considers the amount of physical memory in the system. 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 amount of space available 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 in this case. They report that when starting with an initial heap size of 2 MB, this approach increases throughput by 28% on the VolanoMark and pBOB benchmarks.

Other related work shows empirically that performance enabled by garbage collection is application-dependent. For example, Fitzgerald and Tarditi [3] performed a detailed study comparing the relative performance of applications using several variants of generational and non-generational copying collectors (the variations had to do with the write barrier implementations). They showed that over a collection of 20 benchmarks, each collector variant sometimes provided the best performance. On the basis of these measurements they argued for pro-file-directed selection of GCs. However, they did not consider variations in input, required different pre-built binaries for each collector, and only examined semi-space copying collectors. They further observed that no single collection system enables the best performance for all applications and all heap sizes and the difference in performance can be significant.

Ungar and Jackson [4, 5] conduct simulation studies to examine the impact that tenuring decisions have on the pause times in a generation-based scavenging garbage collector. They first show that when using a fixed-age tenuring policy, low-tenure thresholds (based on the amount of time an object has survived) produce the most tenured garbage and the shortest pause times. The authors then introduce feedback-mediated tenuring in which future tenuring decisions are based upon the amount of surviving data in the youngest generation. Their work is able to reduce pause times in their simulated environment, which can provide significant benefits in an interactive environment. However, they do not consider the cost of later collecting the increased amount of memory that has been tenured, nor the impact of page faults.

Moon [6] points out that user of some early Lisp machines found that garbage collection made interactive response time so poor that they preferred to turn garbage collection off and reboot once the virtual address space was consumed. He also demonstrates that some applications execute fastest with garbage collection turned off.

Cooper et al. [7] show how the performance of Standard ML can be improved by applying optimizations to a simple generational collector introduced by Appel [8]. In addition to utilizing Mach's support for sparse address spaces and external pagers, they propose and study a modification to Appel's algorithm for deciding how to grow the heap. For each of the three applications studied, they use a brute force approach to determine optimal values (i.e., those that produce the fastest execution time) for the two parameters used by Appel's algorithm and the three used by their own. By modifying the algorithm for growing the heap, they are able to significantly reduce the number of page faults and the execution time of two of the three applications studied (the performance of the third was not changed significantly). Although this aspect was not the focus of their study, it is interesting to see that the set of optimal parameters is different for each application and that it varies with the other approaches used to reduce paging.

Zorn [9,10] points out that the efficiency of conservative garbage collection can be improved if more garbage can be collected during each collection phase, and suggests that one way to achieve this is to wait longer between collections. However, he also warns that there is a trade-off between the efficiency of collection and program address space. In addition, he describes a policy for scheduling garbage collection that is based on an "allocation threshold." This means that the collector runs only after a fixed amount of memory has been allocated (e.g., after every 2 MB).

ENVIRONMENT AND BENCHMARKS

We measured the performance three multithreaded benchmarks of bubble sort (BuS), selection sort (SelS) and quick sort (QkS) on a random numbers of arraylets of size of 50000 numbers. Different arraylets of same size are created and then sorted by using iterative sorting algorithms. The user defined functions for bubble sort, selection sort and quick sort are called for five, ten, fifteen, twenty and twenty five threads. The different threads are allowed to finish before the static method main. Synchronized behavior of the different threads is created by calling appropriate synchronized blocks to allow the completion of the user created threads before the system threads so that the minor changes in the behavior of the different garbage collectors on multithreads can be noted. The results in tables depict the average of five runs for each entry. The execution of the threads is done in small stack sizes of 4mb, 8mb and 16mb. The minimum stack size was kept as 4mb, 8mb and 16mb where as the maximum size of the stack for all the application was set as 32mb.

The hardware used for conducting the tests was Intel core(R) Core(TM) 2 CPU T5600, 1.83 GHz, L1 cache of 64KB, L2 cache of 2048KB and 512 MB RAM. Microsoft Windows XP Professional Version 2002 Service pack 2 was used as an operating system. Java version "1.5.0" Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-b64), Java HotSpot(TM) Client VM (build 1.5.0-b64, mixed mode) was used as software environment.

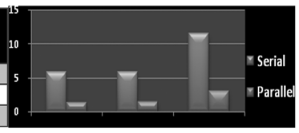
ANALYSIS OF NUMBER OF PAUSES

It is observed that the numbers of pauses in case of serial garbage collector are much ore as compared to parallel garbage collector in 4mb of stack but as the stack size grows

from 4mb to 16mb the number of pauses in case of serial garbage collector are reduced to some extent but they are still unavoidable as compared to the parallel garbage collector. When the average number of pause in all the number of application are measured in 4mb stack it is observed that in case of BuS the average number of pauses for serial garbage collector is 7.2 and for parallel garbage collector is 1.2. For the application of SelS the average number of pauses for serial garbage collector is again 7.2 and for parallel garbage collector is 1.4. In the application of QkS the average number of pauses for serial garbage collector is again 14.2 and for parallel garbage collector are 3. If is recommended that for systems with small memory such as video games, embedded systems stop the world collector is not suitable over the parallel garbage collector. When the size of the stack is increased from 4mb to 16mb the average number of pauses in case of BuS and SelS for serial garbage collector are 3 and for parallel garbage collector the average number of pause are 1.4 but in QkS the number of pause for serial garbage collector are 6 and for parallel garbage collector the average number of pause are 3. The number of pauses in case of serial garbage collector remains more as compared to parallel garbage collector in all the applications. If we look at the average of all the three multithreaded applications in case of 4mb stack the average number of pause are 9.5 and 1.8 for serial garbage collector and parallel garbage collector respectively. For 16mb stack the average number of pause are 4 and 1.9 for serial garbage collector and parallel garbage collector respectively.

Table1
Average Number of Pauses of Three Different Stack Sizes of all Three Application

Application	Serial Collector	Parallel Collector
BuS	5.8	1.33
SelS	5.93	1.4
QkS	11.6	3

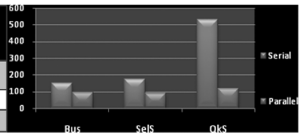


ANALYSIS OF EXECUTION TIME OF THE APPLICATION

We have also measured the garbage collector times in three multithreaded applications. It is observed that executing the application with serial garbage collector is much more expensive than the executing the same application with the parallel garbage collector. We have observed that in 4mb of stack the average time of the execution of BuS application is 191.6ms while executing the same application in same stack size with the parallel collector costs only 82.2ms. For the SelS application the average time of the execution is 196.4ms where as executing the same application with the parallel collector costs only 99.4ms. QkS application takes 622.2ms and 106ms with the serial and parallel garbage collector respectively. In a stack of 16mb the average time of the execution of BuS application is 131.6ms while executing the same application with the parallel collector costs only 104ms. For the SelS application the average time of the execution is 152ms where as executing the same application with the parallel collector costs only 84ms. QkS application takes 321ms and 121ms with the serial and parallel garbage collector. The average time of execution of all three applications in 4mb of stack with serial garbage collector is 366.73ms and with the parallel collector the average is 95.86ms and the average of all the application in 16mb of stack with serial garbage collector is 201.66ms and with the parallel collector the average time is 103.2ms.

Table 2
Average Execution Time of Garbage Collector in Three Different Stack Sizes for all Three Applications

Application	Serial Collector	Parallel Collector
BuS	150.4	94.46
SelS	173.46	88.2
QkS	527.86	118.46



CONCLUSION

The parallel garbage collector is the choice for most multithreaded applications over the serial garbage collector. Garbage collection plays an increasingly important role in next generation Internet computing and server software technologies. However, the performance of garbage collection systems is largely dependent upon application execution behavior and resource availability particularly the stack size. Garbage collection can occur in a number of situations. For example, when the amount of memory remaining in available memory falls below some pre-defined level, garbage collection is performed to regain whatever memory is recoverable. Also, a program or function can force garbage collection by calling the garbage collector. Finally, the garbage collector may run as a background task that searches for objects to be reclaimed. The overhead introduced by selection of the wrong GC can be significant. In this work, we implement and evaluate the performance of serial garbage collector and parallel garbage collector in small footprints for multithreaded environments. It is concluded that the use of the serial GC is not recommended over the parallel GC in memory starved applications.

REFERENCES

[1] KIM, T., CHANG, N., AND SHIN, H. 2000. "Bounding Worst Case Garbage Collection Time for Embedded Real-time Systems". In Proceedings of the IEEE Real Time Technology and Applications Symposium (RTAS).

[2] DIMPSEY, R., ARORA, R., AND KUIPER, K. 2000. "Java Server Performance: A Case Study of Building Efficient", Scalable JVMs. IBM Syst.

[3] Fitzgerald, R., Tarditi, D., 2000. "The Case for Profile-directed Election of Garbage Collectors. In: ACM

SIGPLAN International Symposium on Memory Management (ISMM).

[4] UNGAR, D. M. AND JACKSON, F. 1988. "Tenuring Policies for Generation-based Storage Reclamation". ACM SIGPLAN Notices.

[5] UNGAR, D. M. AND JACKSON, F. 1992. "An Adaptive Tenuring Policy for Generation Scavengers". ACM Trans. Program. Lang. Syst.

[6] MOON, D. A. 1984. "Garbage Collection in a Large LISP System". In Conference Record of the ACM Symposium on Lisp and Functional Programming (Austin, TX), G. L. Steele, Ed. ACM Press.

[7] APPEL, A. W. 1989. "Simple Generational Garbage Collection and Fast Allocation". Soft, Pract. Exper.

[8] COOPER, E., NETTLES, S., AND SUBRAMANIAN, I. 1992. "Improving the Performance of SML Garbage Collection using Application-specific Virtual Memory Management". In ConferenceRecord of the ACM Symposium on Lisp and Functional Programming (San Fransisco, CA). ACM.

[9] ZORN, B. 1990. "Comparing Mark-and-sweep and Stop-and-copy Garbage Collection". In Conference Record of the ACM Symposium on Lisp and Functional Programming (Nice, France). ACM Press.

[10] ZORN, B. 1993. "The Measured Cost of Conservative Garbage Collection". Softw. Pract. Exper.