International Journal of Information Technology and Knowledge Management
July-December 2011, Volume 4, No. 2, pp. 339-341

RUDALICS DISTRIBUTED COPYING COLLECTOR: ANALYTICAL STUDY

Shubhandan S. Jamwal

In distributed processing more than one computer is used for the completion of the application. Distributed processing may also involve parallel processing. But generally distributed processing refers to local-area networks (LANs) designed so that a single program can run simultaneously at various nodes. With distributed software applications, the process of garbage collection faces many new challenges, because the objects may be used by applications running across the different machine on the LAN or any other network. In this paper we have discussed the Rudalics Distributed Copying Collector; the major distributed processing garbage collection algorithm. A conceptual analysis of the collector is presented in this paper. In this paper the improvements to the Rudalics Collector are also suggested.

1. Introduction

Distributed software application has new challenges during its implementation and execution. One object can be used by many different programs running on many different computers. Now the browser such as Netscape Navigator has built-in CORBA support. But with the advent of the distributed processing the process of the garbage collection has become very important. The Garbage Collection is a technique of automatic reclamation of allocated program storage and was first proposed by McCarthy [1]. Other techniques for storage reclamation also exist and they are explicit programmer-controlled reuse of storage used in Pascal, C, and reference counting [2] and so on. Processing is distributed across two or more machines and the mutators are running at the same time on the two or more machines. Each process performs part of an application in a sequence. Distributed processing environment may also be distributed across different platforms. During the process of garbage collection the intergenerational pointers are also affecting the performance of the mutator. Since the inception of the distributed processing the process of garbage collection has challenged the modern compiler developers.

2. Literature Review

Garbage collection places large overhead on the execution of the program. The cost of the garbage collection is highly system dependent. Studies from 1970's and early 1980's found that large LISP programs are typically spending up to 40% of their execution time in garbage collection [6,7,8]. Clement R. Attanasio, David F. Bacon, Anthony Cocchi, and Stephen Smith [9] observed that when resources are abundant, there is no clear winner in application speed. However, 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.

Stephen M Blackburn, Perry Cheng, Kathryn S McKinley[10] observed that the overall performance of generational collectors as a function of heap size for each benchmark is mainly dictated by collector time. Semi Space is often the best in large heaps, but Mark Sweep does better in tight heaps. The overall results are not encouraging for constrained memory. Even with generational collectors, memory management costs are prohibitive. Garbage collection algorithms still trade for space and time which needs to be better balanced for achieving the high performance computing.

The principal goal of EVM [10,11,12] embedded in JAVA2 SDK, is to achieve high performance for multi-threaded programs executing on hardware with multiple CPUs. To achieve sequential efficiency, EVM uses a combination of interpretation and compilation to execute the byte code format defined by the Java platform ("Java byte code").

3. Rudalics Distributed Copying Collector

Rudalics suggested a copying algorithm for a distributed environment [3]. This algorithm is a combination of Cheney's copying collector and Baker's real time algorithm. Collection is incremental but each step may take an unbounded amount of time in a processor. The local memory of each processor is divided into three spaces: the root space, which stores global objects, and two semi spaces. Roots are invisible to the programmer, and serve as a second stage in the indirection of references between processors. Each root is an incoming external reference and contains a local

PG Department of Computer Sciences, University of Jammu at Bhaderwah, Jammu
E-mail: [1]jamwalsnj@gmail.com

This document was created using SOLID CONVERTER PDF To remove this message, purchase the product at www.SolidDocuments.com

pointer to the actual object and a tag bit for garbage collection. Roots are linked in either of three lists. The first two list acts as semi space for root, while the third is used to store roots temporarily while a remote object or root is being created. Semi spaces are used by collector for moving and compacting local objects. The upper parts of each semi space is reserved for storing remote pointers, which act as indirection to external references and also have tag bits for garage collection. Roots and remote pointers establish a two stage indirection concepts, and are similar to inter area links [4] and entry / exit points [13,14]. The algorithm assumes that all objects are reachable from one global root, from which collection stops. Collection consists of a scan phase followed by flip phase that eventually includes all processors. Rudalics suggests interleaving local collections the global ones in order to reclaim short lived objects more easily. This protocol is unable to terminate global objects, but is able to collect them after they exhausts there own resources [15].

4. Analysis

Rudalics algorithms suggested that the local memory of each processor shall be divided into three spaces: the root space, which stores global objects, and two semi spaces. But the ratio of the memory reserved for the root space and the two semi spaces varies from application to application. More over the roots are also invisible to the programmer. These roots serve as second stage in the indirection of references between processors. If the root space will remain visible to the programmer the programmer can take corrective measures for the effective memory management.

Rudalics algorithm assumes that all objects are reachable from one global root, from which collection stops. Collection consists of a scan phase followed by flip phase that eventually includes all processors. It is observed that performance of the train collector is almost equal to the performance of serial collector in terms of memory reclamation. But as the stack size increases, the performance of all garbage collectors becomes equal. In cyclic and object serialization applications, the incremental collector reclaims very high memory as compared to others. As the size of the stack increases to 64mb the performance of the incremental collector becomes better. Therefore it is suggested that the process of garbage collection should be started in the incremental manner. It is further suggested that approximate time of the garbage collection, if possible by the individual processor should be calculated in advance, so that the server can send the tasks to be performed to some other available processor. The task of the garbage collection should be preceded at the different processor level in an Incremental way. Therefore the time for which the mutator remains stopped for a particular processor, the job of the processing can be switched over to the other processing node.

Table 1
Average Memory Reclaimed by all Garbage Collectors in Threaded Applications

	SR GC	PR GC	TR GC	INC GC
4mb	42573.71	2325640	1459622.29	X
8mb	42573.71	2325640	1459638	1853650
16mb	159860.57	2325640	1909024	1853635
32mb	637451.86	2323299.43	1909028.7	1853632
64mb	1595133.14	1878496	1909024	1853635

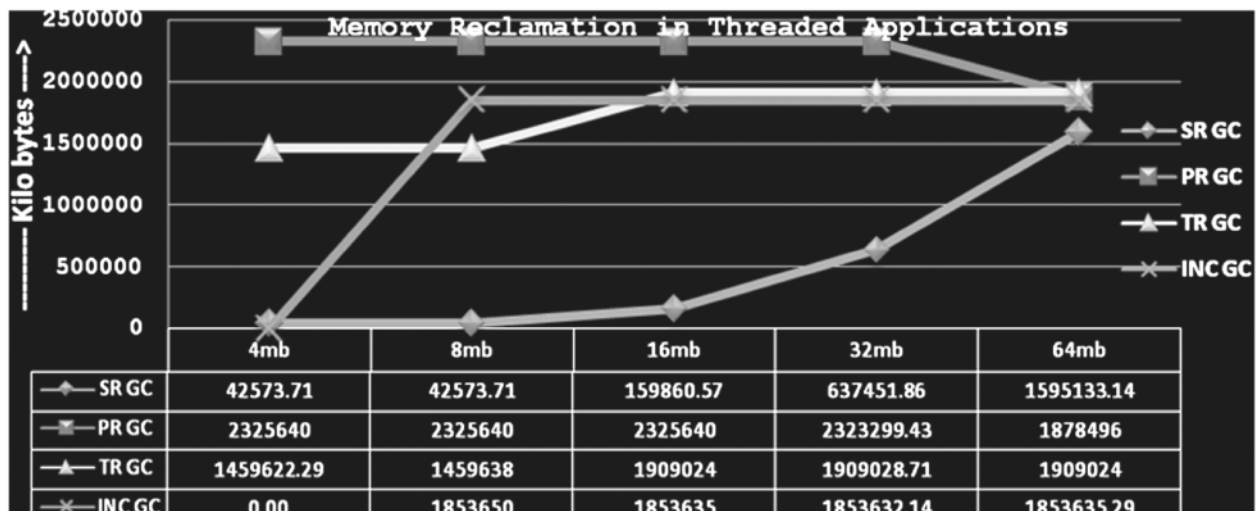

Fig. 1: Performance of the Garbage Collectors in Memory Reclamation

This document was created using SOLID CONVERTER PDF. To remove this message, purchase the product at www.SolidDocuments.com

5. Conclusion and Future Scope

The ratio of the memory reserved for the root space and the two semi spaces varies from application to application. Therefore the space management should be varied from application to application. The roots should also be made visible to the programmer so that the programmer can take some corrective measures and manage the memory accordingly. The possible improvements suggested in the paper can be simulated over different local area networks and different programming languages which are provided with implicit garbage collection procedures.

References

[1] John McCarthy, "Recursive Functions of Symbolic Expressions and their Computations by Machine", Communications of the ACM, 3(4):184-195, April 1960.

[2] George E. Collins, "A Method for Overlapping and Erasure of Lists", Communications of the ACM, 2(12):655-657, December 1960.

[3] M. Rudalics, Distributed Garbage Collection. In [LFP, 1986], page 364-372.

[4] Peter B. Bishop, Computer System with Very Large Address Space and Garbage Collection, Phd. Thesis MIT Laboratory for Computer Science, May 77, Technical Report MIT/LCS/TR-178.

[5] M. Schelvis and E. Bledoeg, "The Implementation of a Distributed Smalltalk", Lecture Notes in Computer Science. 322 :212-232, 1988.

[6] Guy L Steele, "Multiprocessing Compactfying Garbage Collection", Communications of the ACM, 19(6):354, June 1976.

[7] John K. Foderaro nad Richard J. Fateman, "Characterization of VAX Macsyma." ACM symposium on Symbolic and Algebraic Computation, pp 14-19, Berkeley, CA, 1981, ACM Press.

[8] Richard P. Gabriel, "Performance and Evaluation of Lisp Systems", MIT Press, Cambridge MA-1985.

[9] Clement R. Attanasio, David F. Bacon, Anthony Cocchi, and Stephen Smith, "A Comparative Evaluation of Parallel Garbage Collector and Implementations", IBM T.J. Watson Research Center, LNCS 2624, pp. 177–192, 2003.

[10] Ole Agesen and David L. Detlefs. "Finding References in Java Stacks", OOPSLA'97 Garbage Collection and Memory Management Workshop, Atlanta, GA, October 1997. http://www.dcs.gla.ac.uk/~huw/oopsla97/gc/papers.html.

[11] Ole Agesen, David Detlefs, and J. Eliot B. Moss, "Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines", Proceedings of the ACM SIGPLAN '98 Conference Programming Language Design and Implementation (PLDI), pages. 269-279, Montreal, Canada, June 1998.

[12] Derek White and Alex Garthwaite, "The GC Interface in the EVM." Technical Report, SMLI TR-98-67, Sun Microsystems Laboratories, Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA, December 1998.

[13] Henry Lieberman and Carl E. Hewitt, A real Time Garbage Collector Based on Life time of the Objects, Communications of the ACM, 26(6):419-29, 1983. Also report TM-184, Laboratory for computer science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[14] David Plainfosse and Marc Shapiro, Experience with fault Tolerant Garbage Collection in a Distributed LISP System, IWMM, 1992.

[15] Richard Jones and Rafael Lins, Garbage Collection: Algorithm for Dynamic Memory Management, John Wiley and Sons, Page 312-313, 1999.

This document was created using SOLID CONVERTER PDF. To remove this message, purchase the product at www.SolidDocuments.com