A Parallel Real-time Garbage Collection Scheme for Shared-memory Multiprocessors

Khayri A. M. Ali

Swedish Institute of Computer Science, SICS
Box 1263, S-164 28 Kista, Sweden
Phone: +46 8 752 15 00
Fax: +46 8 751 72 30
khayri@sics.se

Extended Abstract

A common problem for realizing many advanced programming environments, such as functional, logic, and object-oriented programming environments, is how to manage efficiently the heap storage with automatic garbage collection (GC). These languages are based on dynamic object creation and automatic reclamation of storage during computation. A garbage collector retains a program's data that is in use and reclaims data that is unreachable (*garbage*) by the program. In addition to reclaiming storage, a garbage collector can also restore *locality* to fragmented data by dynamically compacting it, thereby improving the performance of caches and virtual memory systems and reducing memory requirements.

With growing the number of commercial parallel computers, many researchers try to utilize this opportunity to speed up the execution of programs. The garbage collection problem for parallel computers has not yet intensively studied. Further more, not too many researchers have studied the real-time (or incremental) garbage collection problem for parallel computers. Here, we are interested in parallelization of Baker's algorithm [2] for shared-memory multiprocessors.

Our scheme parallelizes a general version of Baker's scheme where processors in the system cooperate to reclaim storage for a single program. It works for contiguous and non-contiguous memory blocks of the storage heap. The heap space is dynamically allocated to processors according to their demands for free space. The scheme supports dynamic load-balancing and improves locality of references. Dynamic load-balancing has contributed by (1) guaranteeing garbage collection progress in every allocation of a new object, (2) guaranteeing successful termination of every GC cycle, and (3) improving efficiency by making processors that have no computation work, due to lack of parallelism in the program, to perform most of GC work leaving the other processors to concentrate on computation work.

In our scheme, the space devoted to the heap is subdivided into two semispaces: OLD and NEW. The could be constructed from contiguous or non-contiguous memory blocks. The only condition is to have a given address that separates the two semispaces. The space of each semispace is dynamically allocated to processors according to their demands for free

space. During program execution new objects are allocated in the NEW semispace and each accessed object in the OLD semispace is moved to the NEW semispace. Each time a new object is allocated, an increment of scanning and copying is done. This increment is calculated at execution time. The scanning and copying work is balanced between processors to guarantee garbage collection progress in every allocation of a new object. Processors with shortage of memory space will give their work to processors with free space to guarantee completion of each collection cycle. Processors synchronize only at the beginning of a new collection cycle to reverse the roles of the two semispaces. The space requirement for the scheme is very small; a three-fields object per each piece of GC work (unscanned area). The scheme has no space overhead for locking objects in OLD semispace.

A number of atomic memory operations are defined and their implementations are specified. The operations are: *read* a value or pointer from an object cell, *write* a value or pointer into an object cell, *assign* a pointer to a cell in the root set, and *create* a new object given its arguments. The mutator is never allowed to see objects in OLD semispace. Whenever the mutator accesses a heap object in OLD semispace, it immediately copies it to NEW semispace if it is not already copied.

We discuss also how to extend the total heap space during program execution and how to dynamically balance GC work between idle processors leaving the other (busy) processors to concentrate on computation.

Our scheme is like Halsted [5] and Herlihy and Moss [6] schemes where all are parallelization of the sequential Baker's scheme [2]. The essential differences between our scheme and the other schemes are in the way of allocating the total heap space to processors and in the way of balancing the load between processors. In our scheme, the semispaces are dynamically allocated to processors according to their demands for free space whereas in the other two schemes the entire heap area is statically divided by the number of processors. In our scheme, the scanning and copying work is balanced between processors whereas the other schemes do not support load-balancing. Since the other schemes do not support load-balancing, they cannot guarantee successful termination of every garbage collection cycle. This is because processors allocate objects in their memory block with different rates. The result could be a situation where some processors may have GC work and have no free space available and other processors have no GC work and have free space. This problem can be solved only by moving GC work from the former processors to the latter processors, i.e., by load-balancing. The other advantages for supporting dynamic load-balancing have mentioned above. To the author knowledge, our scheme is the first parallel real-time garbage collection scheme for shared-memory multiprocessors with dynamic load-balancing and dynamic allocation of heap space to processors.

The other schemes that support dynamic load-balancing and dynamic allocation of heap space are proposed by Imai and Tick in [7] and by the author in [1]. None of them is a real-time scheme. The two schemes are parallelization of two variants of stop-and-copy garbage collection for shared-memory multiprocessors. Imai and Tick scheme [7] is parallelization of the basic sequential copying GC scheme [3] whereas Ali's scheme [1] is parallelization of a sequential copying GC scheme in which live data is traversed depth-first [4].

Keywords

Storage heap; real-time garbage collection; parallel algorithms; shared-memory multiprocessors.

References

[1] Khayri A. M. Ali. A Parallel Copying Garbage Collection Scheme for Shared-memory Multiprocessors. Submitted to *NGC Journal*, August 19, 1994.

[2] Henry G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4): 280–294, 1978.

[3] C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13(11): 677–678, November 1970.

[4] Björn Danielsson, Sverker Janson, Johan Montelius, and Seif Haridi. Design of a Sequential Prototype Implementation of the Andorra Kernel Language, Draft, May 1994.

[5] R. H. Halstead. Implementation of Multilisp: Lisp on a Multiprocessor. *ACM Symposium on LISP and Functional Programming*, Austin, Texas, pages 9–17, 1984.

[6] M. P. Herlihy and J. E. B. Moss. Lock-Free Garbage Collection for Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3): 304–311, May 1992.

[7] Akira Imai and Evan Tick. Evaluation of Parallel Copying Garbage Collection on a Shared-Memory Multiprocessor. *IEEE Transactions on Parallel and Distributed Computing*, 4(9): 1030–1040, September 1993.