Garbage Collection Algorithm for Ubiquitous Real-Time System

Sang-Young Lee[1] and Yoon-Seok Lee[2]

[1,2]*Department of Health Administration, Namseoul University, Cheonan, South Korea*
[1]*sylee@nsu.ac.kr,* [2] *yslee@nsu.ac.kr*

Abstract

Most parallel garbage collection algorithms are based on the mark-and-collect technique. A mark-and-collect technique an effective asynchronous marking algorithm. There are two basic marking techniques: coloring and stacking. The coloring technique is asynchronous but its time complexity is O(MN) where M and N are the total number of nodes in the list memory and the total number of active nodes, respectively. The stacking technique offers effective marking process having only O(N) time complexity but requires extra stack space which can be as large as the size of entire active nodes(N). A new parallel garbage collection algorithm in ubiquitous environment has been devised which takes advantage of the asynchronous processing of coloring algorithms and the time efficiency of stacking algorithms. The algorithm requires no synchronization between the collectors and the mutators. and its tome complexity is close to O(N) with a small fixed-size stack in ubiquitous real-time system.

Keywords: *Parallel, Garbage Collection, Real-Time*

1. Introduction

Recent interest in using artificial intelligence for time-critical ubiquitous real-time systems controlling physical devices such as dynamically unstable airplanes, nuclear reactors, and robots demands a large search space operating in the manner such that it never runs out of space [1] because these applications are required to create and to release large amounts of data continuously without interruption.

Lists are the fundamental data structure in artificial intelligence programs. List processing systems free the programmer from managing the memory storage. Instead, the list processing facility maintains a set of free memory cells (called nodes) and dynamically collects garbage memory nodes which are no longer accessible by the program into the set of free memory nodes. Thus, in time-critical real-time list systems, a real-time garbage collection system is a must.

With the emergence of multiple processor systems, parallel garbage collection algorithms have been proposed where more than one processor, called the collectors, exclusively collect garbage concurrently with the activity of other processors, called the mutators, which are dedicated only to the application processes [2, 3, 4]. The mutators proceed with the list processing activity while the collectors reclaim the garbage nodes concurrently. Thus, the mutator and the collector should operate independently of each other [5, 6].

Recently, Shin [7] developed a parallel garbage collection algorithm using associative tags. His algorithm requires no synchronization between the collectors and the mutators. The algorithm is based on the mark-and-collect technique which consists of three phases. In the initializing phase, every node except free nodes is initialized as a garbage node. Every reachable node is marked during the marking phase as in active node, starting form a set of root nodes. Nodes which remain as garbage nodes are reclaimed in the reclaim phase. The time complexity of Shin's marking process is O(N) and is does not require extra stack

memory. Although Shin's algorithm can be used for implementing a massively parallel garbage collection system, his algorithm requires associative tags which are expensive to construct. We have developed a parallel garbage collection algorithm in ubiquitous real-time system which is linearly scalable and operates asynchronously; thus it is possible to construct a massively parallel garbage collection system for a large time-critical real-time system. In section 2, we introduce general parallel garbage collection algorithms. We present a new parallel garbage collection algorithm suitable for implementing the autonomous memory in Section 3. In Section 4, The proof of correctness of the algorithm is given and the time complexity of the algorithm is presented in Section 5. Conclusions are drawn in section6.

This document is a template. An electronic copy can be downloaded from the conference website. For questions on paper guidelines, please contact the conference publications committee as indicated on the conference website. Information about final paper submission is available from the conference website.

2. Garbage Collection in Ubiquitous Real-time System

There are two basic techniques for parallel garbage collection: copying [6] and mark-and-collect techniques [5, 7]. The copying technique employs the principle of freezing the memory at the instant in time at which the copying begins and preserving the list structure for the collectors. There are two basic methods in the copying technique based on how the copying process is performed: duplication methods [8] and evacuating methods [6]. The duplication method is required to make a copy of a part or of the entire list memory before beginning the garbage collection process. The mutator proceeds with the list processing activity while the collector reclaims the garbage nodes from the unchanging second copy of the list structure.

The evacuation method divides the available memory into two logical space: newspace and oldspace. A garbage collection cycle starts with a flip, in which newspace is converted to oldspace, and vice-versa. The mutator proceeds with the list processing activity in newspace while the collector copies (evacuates) all accessible nodes in oldspace into newspace by tracing them from a set of roof nodes. After completion of the evacuation process, oldspace contains only garbage nodes and may be transferred to free nodes.

The copying garbage collection technique provides several advantages including the ability to reclaim circular structures and to compact the storage which reduces the storage fragmentation and improves locality of reference. The copying garbage collectors have been widely used in a virtual memory environment. However, this technique not only requires extra memory space which can be as large as the requires that list processing activity by the mutator be suspended while the mutator makes a copy of the list structures (duplication method) or the mutator be suspended while the collector must be tightly synchronized for a flip operation (evacuation method). Thus, the copying technique is not suitable for implementing a massively parallel garbage collection system.

Most parallel garbage collection algorithms are based on the mark-and-collect technique. The mark-and-collect technique requires tag bits. In general, a mark-and-collect algorithm consists of there phases: initialization, marking, and collection, In the initialization phase, non-free nodes are initialized ad garbage nodes for the subsequent marking process. In the marking phase, all active nodes are marked b tracing form a set of root nodes, All the nodes that have been neither marked nor already declared as free nodes are transformed into free nodes in the collection phase. There are two basic methods for the mark-and-collect technique: coloring [7] methods and stacking [5] methods.

The coloring method starts by initializing all non-free nodes as garbage nodes in the initialization phase. Marking process begins by marking all root nodes. Then, the collectior

finds a marked node and marks all its immediate descendants. The procedure continues until there is no marked node with unmarked immediate descendants. Only nodes remaining in garbage status after the marking process are reclaimed in the collection phase. Since the mutator never changes the status of nodes to garbage status, this method does not require the suspension of the mutator. Thus, the coloring method can be used to implement a parallel garbage collection system. However, the time complexity off the marking process is O(MN) where M and N are the total number of nodes in the list memory and the total number of active nodes, respectively[1]. The stacking method offers efficient marking requiring only O(N) time at the expense of extra memory space for the stack which can be ad large as the size of the set of entire active nodes(N). Also, the access to the stack must be done through the critical section since the mutator and the collector both need to update the stack. These algorithms are based on the following explicit or implicit assumptions.

1. The marking process is terminated before the mutator exhausts free nodes. Wadler presented sufficient conditions for this assumption in terms of the maximum rate at which free nodes are used, the maximum number of nodes in use at one time, and the total number of nodes in the system, which may be hard to determine[1]. However, in general, the average time needed for the creation of a list element is required to be greater than the average time interval between two consecutive instances of marking a node to guarantee proper termination of the marking process.

2. A set to root nodes for all active lists is provided for the collectors. In general, foot nodes are maintained by the mutator in special memory blocks such as internal registers or stack buffers. All accessible nodes, only those active nodes, are referencable via some paths from the nodes in these special memory blocks. The requirement that the average time needed to create a list element be greater than the average time interval between two consentive instances of marking a node is particularly important because it limits the speed of the mutator relative to the speed of the collector. The most time-consuming process of the coloring method is the marking process, which is O(MN), because the worst time interval between two consecutive instances of marking a node is the time required to search the whole memory. Shin introduced an algorithm using associative tags which provides a constant one-unit time for searching a marked node[7]. thus, the marking algorithm has time complexity O(N), which is optimal. However, it requires associative tags which are expensive.

3. Algorithm Model in Ubiquitous Real-time System

All We present a marking algorithm whose time complexity is close to O(N) by combining the coloring and stacking methods. The algorithm is very similar to the Shin's algorithm[7]. Before proceeding with the algorithm statement, let's present the system model. In our system model, there is a collector for each memory block. The system maintains a separate set of rot nodes for each memory block which is an array of special pointers, ROOT(1), ROOT(2),....,ROOT(R). They contain the pointer to the root nodes of lists residing in the same memory block.

The system also maintains a separate free list for each memory block. There are two pointers to the free list: one to the head of the free list (F-HEAD) from which the mutator accesses the free list and the other to the tail of the free list (F-TALK) to which the collector appends free nodes. The advantage of this organization is that it gives better locality of reference; thus it minimizes the communication between collectors. Identification of garbage nodes is achieved by marking all active nodes. However, when a mutator redirects an existing pointer which the collector has already marked to another active node which has not yet been

marked, problems may occur. The conventional solution of these problems in to let the mutator mark the redirected node which has not yet been marked. This is one of the reasons that the conventional stacking method is required to maintain the backtracking pointers of more than one list which makes the stack size as big as the size of the set of entire active nodes. These problems are alleviated in the proposed system by maintaining a special list, called the replaced node list which contains the pointers of redirected nodes. Unlike a conventional list, the replaced node list is only accessible from the tail of the list. However, the first node of the list is always created at a fixed location. The system maintains a special record called R-POINT for the replaced node list. There are two fields in R-POINT and R-TAIL. The R-FLAG (value 1) indicates the recreation of the replaced node list; thus the first node will be created at the fixed location (R-NODE). A node of the replaced node list contains two fields: R-ROOT and R-PREV. R-ROOT contains the pointer of the redirected node. R-PREV contains the pointer of the previous replaced node except in the first node where the value is NIL.

There is a foxed-size small stack. The stack is organized as a last-in first-out circular queue. Thus, the stack maintains the most recent backtracking pointers up to the stack size(S). The algorithm in ubiquitous real-time system is based on tagged memory. There are two tag bits. Thus, there are four possible states for a node

1. (0,0) - F-state: The nodes in the F-state are free nodes (in the free list) which are available for the mutator to create a new list element.

2. (1,0) - G-state: The nodes in the G-state are garbage nodes. Nodes which are not free nodes (not in the F-state) are initialized as garbage nodes (the G-state). The nodes remaining in the G-state after the marking process are garbage nodes which can be transferred to the free list.

3. (1,0) - A-state: The nodes in the A-state are active nodes.

4. (1,1) - N-state: The nodes in the N-state are new nodes

which are created during the current garbage collection cycle. A new node may or may not be a garbage node. If a new node is create and released during the same garbage collection cycle, it is called a floating node. The algorithm collects floating nodes in the next garbage collection cycle. Thus, garbage nodes are guaranteed to be collected in two cycles.

The algorithm starts by setting the value of R-FLAG to 1 and then initializing all non-free nodes to garbage nodes (G-state) including the root nodes. After completion of the initialization, the marking process begins. The nodes are traced starting from ROOT(i) for i = 1 to R. If the status of ROOT(i) is the G-state, the collector marks the nodes from ROOT(i) in a depth-first order. If any of its descendants nodes are in the G-state, is converts its state to the A-state. The algorithm uses the stack for storing backtracking pointers. Although the stack is small and fixed, the collector operates as if there is unlimited stack memory because the stack is organized as a last-in first-out circular queue. However, the stack maintains only the most recent backtracking pointers up to the size of the stack (S). After the stack is empty, the collector re-examines the states of the immediate descendants of ROOT(i). If any of them are still in the G-state, the collector retraces the list starting from ROOT(i).

This procedure is repeated until none of the immediate descendants of ROOT(i) is in the G-state, at which time the collector converts the state of ROOT(i) to the A-state and proceeds to the next ROOT(i). after the marking of the last ROOT(R), the collector traces the replaced node list backward from the tail (R-TAIL) using R-PRVE until the value of R-PREV is NIL. The collector marks the redirected nodes by tracing them from R-ROOT using the same

procedures as above. The marking process terminates. When the collector has examined all replaced nodes.

A: Initialization Phase in ubiquitous real-time system

Set R-FLAG to 1.
For i = 1 to M, of NODE(i) is not in the F-stage,
initialize it to the G-state.

B: Marking Phase in ubiquitous real-time system

M1: *For all i = 1 to R, if ROOT(i) is in G-state, let ROOT(i) be all parent node,*

1. *If LP = NIL of the state of NODE(LP) is cot G-state, go to step 3.*
2. *Push the parent node onto the stack and let NODE(LP) be the new parent node. Repeat step 1.*
3. *If RP = NIL or the state of NODE(LP) is not G-state, go to step 5.*
4. *Push the parent node onto the stank and let NODE(NODE(RP)) be the new parent node. Repeat step 1.*
5. *Mark (convert the modes from G-state to A-state) the parent node. Pop the stack. If the stack is empty, go to step 6. otherwise let the popped node be the new parent node. Repeat step 1.*
6. *Let ROOT(i) be a parent node. If (LP = NIL or NODE(LP) is not in G-state) and (RP = NIL or NODE(LP) is not in G-state), mark the ROOT(i); otherwise, repeat step 1.*

M2: *If R-FLAG = 1, skip the M2 procedure. Let NODE(R-TALK) be a working node (R-N ODE) and NODE(R-ROOT) be a parent node.*

1. *If LP = NIL or the state of NODE(LP) is not G-state. go to step 3.*
2. *Push the parent node onto the stack and let NODE(LP) be a new parent node. Repeat step 1.*
3. *If RP = NIL of the state of NODE(RP) is not G-state, go to step 5.*
4. *Push the parent node onto the stack and let NODE(NODE(RP)) be a new parent node. Repeat step 1.*
5. *Mark (convert the nodes from G-state to A-state) the parent node. pop the stack. If the stack is empty, go to step 6, otherwise let the popped node be a new parent node. Repeat step 1.*
6. *Let NODE(R-ROOT) be a parent node. If (LP = NIL of NODE(LP) is not in G-state) and (RP = NIL or NODE(RP) is not in G-state), mark the NODE(R-ROOT), otherwise repeat step 1.*

7. If R-PREV is not NIL, let NODE(R-PREV) be a working node (R-NODE). Let NODE (R-ROOT) be a parent node. Repeat step 1.

C: Collection Phase in ubiquitous real-time system

C1: *For all i = 1 to M, if the stare of NODE(i) is G-state, append it to the free list after converting its state to F-state*

C2: *Set R-FLAG to 1.*
The only function required in the mutator for garbage collection is to create the replaced node list whenever the mutator redirected a node which is still in the G-state to a node which is in the A-state.
1. If R-FLAG = 1, then reset it to 0 and create the first R-NODE at the special location, otherwise create an R-NODE at any location. Update the R-TAIL with the pointer of the newly created R-NODE.
2. If R-NODE is a frist node, move the NIL value to R-PREV, otherwise move R-TALK to R-PREV
3. Move the pointer to the replaced node to R-ROOT

After completion of the marking process, nodes still remaining in G-state are garbage nodes. The collector converts these nodes to free nodes (F-state) and appends them to the tail of the free lost. The only function required for the mutator related to garbage collection is the creation of the replaced node list. Whenever the mutator redirects a node which is still in G-state to a node which is in A-state, it creates an R-NODE and places the pointer of the redirected node into R-ROOT and updates R-POINT. However, if the value of R-FLAG is 1, the mutator resets R-FLAG to 0 and creates the first node at the fixed location (R-NODE). The algorithm assumes that each node contains two pointers (LP and RP) to other nodes in the list structure. However, the algorithm can be modified easily to handle list structure having more than two pointers. NODE(LP) and NODE(RP) represent the child nodes pointed to by the pointers LP and RP of the parent node, respectively.

4. Correctness of the Algorithm in Ubiquitous Real-time System

For brevity, we present a brief summary of the proof of correctness of the algorithm. To prove the correctness of the garbage collection algorithm, we need to show:

C1. Invariance: The active list structures should not be modified by the garbage colector
C2. Termination: The garbage collection processes terminate properly.
C3. Validity: No active nodes should be mistaken for garbage nodes.
The following theorems prove the correctness of the algorithm.
Theorem 1: The algorithm satisfies the Invariance condition.
Proof: The collector does not alter any pointer of lists except in the collection phase in which it only alters the pointer of inactive nodes (garbage nodes).
Lemma 2: The Initialization Phase terminates properly.
Lemma 3: The Collection Phase terminates properly.

Proof: The initialization and collection phases involve only the sequential scanning of the memory form top th bottom; they will terminate properly.

Lemma 4: The procedure M1 terminates properly.

Proof: after the initialization phase, all nodes are either in F-state, G-state, or N-state. The nodes in N-state are newly created nodes after the initialization process began. How-

ever, after the initialization process, neither the collector nor the mutator changes the state of nodes to G-state. Thus, there are a fixed number of nodes in G-state. The procedure M1 converts the state of nodes from G-state to A-state; thus, it must terminate properly.

Lemma 5: The procedure M2 terminates properly.

Proof: The problem of redirecting nodes occurs only when the mutator redirects nodes which have not been marked to a node which already has been marked, Thus, after completion of the procedure M1, the redirecting of nodes will not cause a problem. Therefore, redirected nodes pointed to by R-NODE which is created after R-TAIL need not be examined. Since the collector traces the replaced node list backward from R-TAIL to the first R-NODE, it will terminate properly if the redirected lists are not expanded. After the completion of the procedure M1, all active nodes are either marked to A-state or in G-state. If the active node still is in G-state, it must be a member of a redirected list. However, since there is only a finite number of nodes in G-state, the procedure M2 will terminate properly

Theorem 6: The marking phase terminates properly.

Proof: The Theorem is true by Lemma 4 and Lemma 5.

Theorem 7: The garbage collection processes terminate properly.

Proof: It follow from Lemma 2, Lemma 3, and Theorem6

Theorem 8: NO active nodes should be mistaken for garbage nodes.

Proof: The mutator only converts the state of nodes from F-state to N-state. The collector only initializes non-free nodes in the initialization phase. The active nodes in G-state must be marked by either the procedure M1 or M2. The collector only collects the nodes still remaining in G-state. Thus, no active nodes are mistaken for garbage nodes.

From the above theorems, We conclude that the algorithm is correct. However, the algorithm does not collect all garbage nodes in one cycle. Nodes created and released during the cycle(floating nodes) are collected in the next cycle. Thus, the algorithm guarantees that garbage nodes are collected within two cycles. All manuscripts must contain an informative 150 to 300 words abstract explaining the essential contents of the work, key ideas and results.

5. Time Complexity in the Ubiquitous Real-time System

The time complexity of the algorithm cam be estimated as follows. Assume that the distribution of the depth of lists is a normal distribution function with the mean μ and the variance. Since the time complexity of the initialization phase and the collection phase is O and the variance. The time complexity of the algorithm cam be estimated as follows. Assume that the distribution of the depth of lists is a normal distribution function. If we assume that the density function is normal distribution. This is the typical time and space trade-off. However, the time penalty (Equation 2) is considerably smaller than the spatial gain. If the stack size is, the time complexity of the algorithm is just 16% more than the optimum. If the stack size is, the time complexity of the algorithm is almost optimum (1.001 times of the optimum). The algorithm is not dependent on M or N; thus the time complexity of the algorithm is O(1).

Table 1 : A Performance Comparison of Marking Algorithms for Garbage Collection

Algorithm	Stacking	Coloring	shin[7]	New Algorithm
Time	O(N)	O(MN)	O(N)	O(N)
Tag Bits	1	2	2	2
Assoc. Memory	no	no	no	no
Critical Section	stack	none	none	none
Extra Space	O(N)	none	none	O(1)

This is a substantial improvement over stacking algorithms which require the stack space as big as the entire available memory (M) or coloring algorithms whose time complexity is O(MN).

6. Conclusions

We have presented a parallel garbage collection algorithm in ubiquitous real-time system which can be used for artificial intelligent systems for time-critical real-time applications. The algorithm takes advantage of the time efficiency of stacking algorithms and the space efficiency of coloring algorithms. The algorithm requires no critical section and the time complexity of its marking process is close to O(N) with a small fixed-size stack. Thus, a massively parallel garbage collection system can be effectively constructed.

In this paper, we didn't discuss the case when a list is expanded form one memory bank to other memory bank for brevity. In this case, an address fault would occur and the collector needs to send the address of the children to other collectors. A receiving collector is required t create a temporary list similar to the replaced node list. At the end of the tracing of the replaced node list, the collector marks the active nodes whose roots are located in other memory blocks by tracing the temporary list. To construct a massively parallel garbage collection system. we need to design a communication network to interconnect collectors. A communication system called a has been developed to interconnect multiple processors to from a massively parallel computer at the Aerospace Technology Censer of the allied-Signal Aerospace company[10]. The communication network exhibits a high degree of connectivity, modularity, extensibility, and scalability. The same module would be used for constructing a massively parallel garbage collection in ubiquitous real-time system.

References

[1] J. Richard, “Garbage Collection”, Wiley and Sons, **(2009)**.

[2] V. Bill, “Java's Garbage Collection Heap”, Javaworld, **(2007)** August.

[3] N. I. A. Woodward, “Alternative Approaches to Multiprocessor Garbage Collection”, Proc. Int. Conf. Parallel processing, **(2006),** pp. 205-210.

[4] P. Amsaleg, M. Freerira and M. Shapiro, “Evaluating Garbage Collection for Large Persistent Stores”, Proceedings of the OOPSLA Workshop on Object Database Behavior, Benchmarks, and Performance, Austin, Texas, **(2006)** October.

[5] Y. Hibino, “A Practical Parallel Garbage Collection Algorithm and Its Implementation”, Proc. Annual Symp. on Computer Architecture, **(2011)**, pp. 113-120.

[6] N. Podhorszki, “Analysis of the multi-phase Copying Garbage Collection Algorithm”, International Journal of Computational Science and Engineering, vol. 4, no. 3, **(2009)**, pp. 243-246.

[7] H. Shin, "A Boolean Content Addressable Memory and Its Applications", Univ. of Texas, Austin, **(2000)** May.
[8] R. J. P. Ingria and M. Cohen, "LAMBDA release 3.0 Notes", LISP Machine Inc., **(1986)** October.
[9] D. W. Clark and C. C. Green, "An Empirical Study of List Structure in Lisp", ACM 20, **(1999)**.
[10] C. Zhang, C. Wu and L. Zhao, "Research on Algorithm of Parallel Garbage Collection Based on LISP 2 for Multi-core System", Communication in Cpmputer and Information Science, vol. 93, **(2010)**.

Authors

Sang-Young Lee

Professor, Dept. of Health Administration, Namseoul University, South Korea.

Yoon-Seok Lee

Professor, Dept. of Health Administration, Namseoul University, South Korea.