NAVAL POSTGRADUATE SCHOOL
Monterey, California

DEPARTMENT OF THE NAVY
NAVAL POSTGRADUATE SCHOOL

THESIS

THE FEASIBILITY OF AUTOMATIC
STORAGE RECLAMATION WITH CONCURRENT
PROGRAM EXECUTION IN A LISP ENVIRONMENT

by

Kevin G. Cassidy

December 1985

Thesis Advisor: Bruce J. MacLennan

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Feasibility of Automatic Storage Reclamation with Concurrent Program Execution in a LISP Environment		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis December 1985
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Kevin G. Cassidy		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93943-5100		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93943-5100		12. REPORT DATE December 1985
		13. NUMBER OF PAGES 107
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Garbage Collection, list processing, parallel & real-time collection, storage reclamation, parallel processing, LISP, compaction

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In "classical" LISP implementations, program execution/computation continues until there is no more memory available (i.e. the free list of available cells has become exhausted). When this happens, user program(s) HALT and then storage reclamation, in the form of garbage collection, takes over. This halting of programs in the midst of their computation is not only frustrating to programmers and researchers but can also be of crucial importance in other applications. This paper investigates (Continues)

ABSTRACT (Continued)

the feasibility of allowing concurrent program execution with garbage collection. Proof of correctness and performance issues are not discussed. Neither allocation of memory techniques/procedures nor garbage collection in virtual memory systems are thoroughly discussed. These issues are thoroughly described in the listed references. LISP has been selected because it has been estimated that typical LISP programs take 10%-30% of their processing time to perform garbage collection.

The Feasibility of Automatic Storage Reclamation with Concurrent Program Execution in a LISP Environment

by

Kevin G. Cassidy
Lieutenant Commander, United States Navy
B.S., U.S. Naval Academy, 1972

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

Naval Postgraduate School
December 1985

ABSTRACT

In "classical" LISP implementations, program execution/computation continues until there is no more memory available (i.e. the free list of available cells has become exhausted). When this happens, user program(s) HALT and then storage reclamation, in the form of garbage collection, takes over. This halting of programs in the midst of their computation is not only frustrating to programmers and researchers but can also be of crucial importance in other applications. This paper investigates the feasibility of allowing concurrent program execution with garbage collection. Proof of correctness and performance issues are not discussed. Neither allocation of memory techniques/procedures nor garbage collection in virtual memory systems are thoroughly discussed. These issues are thoroughly described in the listed references. LISP has been selected because it has been estimated that typical LISP programs take 10%-30% of their processing time to perform garbage collection.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I wish to acknowledge and thank three "key" people for their support and assistance on this thesis. First, I wish to thank Dr. B. J. MacLennan for his patience, guidance, and assistance. It was from Dr. MacLennan's course of instruction that I first became interested in the study of garbage collection. Secondly, I wish to acknowledge my father, John R. L. Cassidy, for his wisdom and direction, for whom I have everything to thank for. Lastly and most importantly, I wish to give formal and public appreciation to the number ONE person that totally and whole-heartedly supported me during the many long hours required in the research, writing, and editing of this thesis: that person is my loving wife, ORENA.

I. BACKGROUND

A. INTRODUCTION

Interest in storage reclamation methods, in particular garbage collection, has increased dramatically the past several years. Storage reclamation is the process of reclaiming discarded information and returning the memory space occupied by that discarded information to an available area or list in memory that is available to be utilized by programmers. This increased interest in storage reclamation is a result of the increasing use of list-processing environments. A list-processing environment is a system in which the language manipulates data structures called "lists." In other words, in a list-processing environment, the information to be reclaimed is in the form of list cells.

One of the better known and more popular languages that mainpulates lists is LISP, which is designed to facilitate programming of complex symbolic processes. It eases this burden by providing automatic storage allocation and reclamation [Ref. 1: p. 522].

During the period of time that storage reclamation is taking place, program execution comes to a halt. While this time period may appear to be insignificant to the programmer (say in terms of seconds), it can become frustrating and becomes increasingly larger as the number of programmers on the system increases, resulting in longer "wait" times for each user. Therefore, if it were possible to conduct storage reclamation without having to stop program execution, it would allow program execution to continue at the same time that the necessary storage reclamation operations are taking place.

Experience with large LISP programs indicates that 10% to 30% of their execution time is spent in garbage collection [Ref. 2: p. 341]. This paper investigates the feasibility of being able to perform garbage collection concurrently with program execution in LISP.

B. HISTORY

As programs that utilize extensive list-processing become more and more common and as they continue to grow and become more complex, it becomes essential that a method for efficient and real-time storage reclamation be used. Delays due to storage reclamation are a nuisance to programmers and researchers but they could be of critical importance in some applications. For example, a natural language interface to an emergency medical database, that is written in a list-processing language such as LISP, could be considered untrustworthy if garbage collection caused lengthy delays [Ref. 3: p. 1143]. A method for returning discarded information to memory (to a structure called the "free list") is an essential ingredient of any list-processing system. This free list, a space in memory that is available to be utilized by user programs, contains all cells that are not being used by any program (i.e. cells are system representations of memory space). Initially, the free list contains all storage not occupied by the programs; cells are removed from it and formed into list structures as the programs are executed [Ref. 4: p. 501]. The term "working list" is the set or list of cells that is currently being utilized by a user's program.

There are several storage reclamation methods. Each method is described in the following chapter. One of the classical methods, and the one that most LISP implementations use for reclaiming unused memory space, is called garbage collection. Basically it is the operation of first marking all cells in memory reachable from the main program and then sweeping or returning all unmarked cells to an available free list [Ref. 5: p. 491]. Garbage collection is usually invoked only after the main program has run out of memory. The time when garbage collection is invoked depends upon the implementation of the storage reclamation method on the computing system in question.

A requirement of any storage reclamation method, including garbage collection, is to return cells back to memory, or the free list, only when that cell is no longer needed by the program (i.e. it is discarded and no longer accessible from the program). Storage reclamation causes program interruption normally when the program has no more memory to use (i.e. no more cells in the free list), and thus program execution is suspended until the storage reclaimer has finished.

C. LISP LANGUAGE

It is assumed that the reader has a fundamental grasp and understanding of LISP. LISP was selected as the language to investigate the feasibility of CONCURRENT storage reclamation because it is used for highly interactive programming. The language has several properties, including program/data equivalence, that enable a certain style of programming to develop that is characterized by powerful interactive support for programmers, non-standard program structures and non-standard program development methods [Ref. 6: p. 35].

A LISP list is really nothing more than a linear list of elements called cells. These cells have fields and may contain pointers to other lists. The list is a "finite sequence of zero or more cells or other lists" [Ref. 7: p. 406]. A cell can be thought of as one or more continuous computer words, representing memory space, that can be made available to a user [Ref. 8: p. 534]. These cells are requested from the free list by the user's program. Because there is a finite number of available cells, there may come a time that there are no more cells still in memory for the program to use. When this happens, LISP uses an automatic method of reclaiming discarded cells of the user's program. This method is called garbage collection or "regular" garbage collection. A cell becomes discarded (commonly referred to as "garbage") when it can no longer be pointed at or accessed through the pointer fields of any reachable or accessible cell. It is then left to the garbage collector to reclaim this "garbage" and return these cells to the free list [Ref. 2: p. 342]. The garbage collector can be implemented using hardware methods, software methods, or a combination of both hardware and software.

A cell is called "accessible" if it is reachable from at least one root via any directed path. LISP terminology will be used throughout the remainder of this paper and the words "cell" and "node" will be used interchangeably.

Each node has a separate identity, which means that it can be identified independently of the structure of any directed graph. Finding this node from a "root" and finding any left or right successors of this node are called "primitive operations." The process of locating or finding the predecessors of this node involves a search through the entire list of nodes; i.e. through the entire area of memory allocated to the users' programs. This is the reason that identifying garbage is not such a simple or easy task. The task is delegated to the garbage collector, which maintains the free list. Again, the free list is just a collection of nodes that have been identified as "garbage" and are available to a user's program.

LISP is a high level language and as such contains primitives that automatically call the garbage collector. In LISP the function cons triggers the garbage collector when no more free cells are available.

LISP functions are called S-expressions (where S means symbolic). The basic elements of an S-expression are called atoms. These expressions are surrounded by parenthesis and these parenthesis must balance in a meaningful way. For example, the expression:

(add 8 9)) (add(add 2 1)

is not a proper S-expression. However, the following expression is a properly written S-expression:

(times(difference 8 6) (sum 3 1)).

In the latter expression the proper answer is: 8.

Although it is awkward to use functional notation for simple arithmetic, one of the good features of LISP is that everything in the language can be expressed as a function [Ref. 9: p. 140]. These functions are represented by the basic data structure of LISP, which is the list. When the components or cells of a list contain pointers in their cell fields, then the list data structure topology resembles a tree. The head of this tree is called the root and the elements of the tree are called nodes. For example, in Figure 1.1 the S-expression used above is expressed in a tree-like structure.

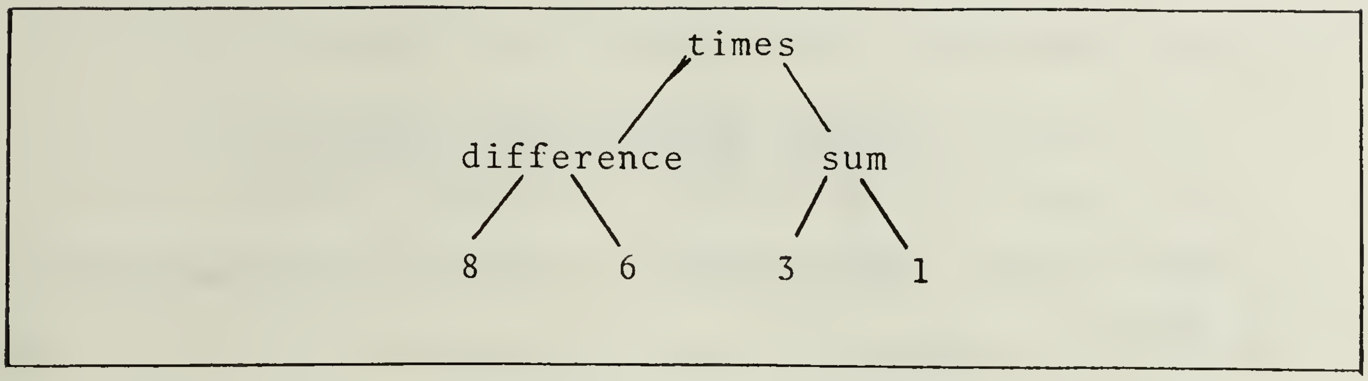


Figure 1.1 S-expression Tree.

Some common LISP functions are car and cdr, which respectively locate and provide the first element in a list and the remaining elements in the list (if there are any) [Ref. 9: p. 140]. Another feature of LISP which is invaluable in an interactive system is that LISP can manipulate symbols as well as numbers. For example, the function (SETQ K 95) serves a dual purpose in that it not only acts as an assignment statement but it also gives the variable "K" the value of 95. Whereas the function (SET K E) equates the variable that is the value of "K" to the value of "E" and takes-on whatever value that "E" has.

Every atom in LISP has a value. Numbers or literals are atoms whose values are themselves, but the value of an atom also can be another atom with a symbolic name. NIL is the terminator that indicates the end of a list.

The next chapter provides a look at the various methods of storage reclamation.

II. STORAGE RECLAMATION TECHNIQUES

There are two basic methods for Storage Reclamation:

(1) Manual --

  (a) the responsibility for reclamation lies with the programmer.

(2) Automatic -- Eliminates programmer responsibility and includes the following two subtypes:

  (a) Reference counts

  The list manipulation primitives maintain a reference count for each cell that indicates the number of other cells which point to it [Ref. 10: p. 495].

  (b) and garbage collection.

Each of these methods is discussed further in the subsequent sections of this chapter.

A. PROGRAMMER RESPONSIBILITY STORAGE RECLAMATION

The first method of storage reclamation is the manual or programmer responsibility method. There are several languages that provide the capability for the programmer to allocate storage and deallocate storage (i.e. storage reclamation). IPL-V includes instructions that cause lists and list structures to be erased and their cells to be returned to the free list [Ref. 4: p. 501]. These instructions are for use by the programmer; what this means is that the programmer has to keep track of the current status of all lists, sublists, and other data structures. One immediately recognized disadvantage of this method is that, in addition to maintaining a table of the status of each data structure, the programmer, when "de-allocating" storage, may accidentally erase a cell (or even a list of cells) that are being shared with other lists: i.e. being pointed at by cells in other lists. These other cells may still be required in the user's program and be accessible from/to the user's program. This will result in the "dangling reference" problem, which is discussed later in this chapter.

Other languages, such as Pascal, use "explicit erasure," which means that whenever a running program no longer requires a particular cell, then the programmer himself must explicitly return it to the free storage area [Ref. 11: p. 440]. Pascal uses two procedures to accomplish this task:

(1) The "dispose" procedure returns to the free storage area whatever cell was pointed to by the argument of "dispose." This is done by linking the discarded cell onto the free list. Later, the storage allocator will be able to reuse this cell from the free list to satisfy a request from a user's program.

(2) The "new" procedure obtains an available cell from the free list to satisfy a request for memory space from a user's program.

There are several other inherent disadvantages in this method of storage reclamation in which the programmer shoulders the burden of allocating and deallocating memory spaces. The programmer has to work harder. He has to remember additional items, such as the current status of each cell and whether that cell is still active (i.e. still required for his program) or if the cell is non-active (garbage). In a list-processing system, this also means that he has to remain aware of which lists, sublists, cells in the lists, etc. are active and which are non-active. This is a large task and one that few programmers can accomplish successfully and accurately, especially as the size and complexity of the user's program increases. The programmer's main task should be to concentrate on the more important issues-at-hand, such as program structure and organization.

As mentioned earlier, there may be times when a cell will be returned to the free list but that cell is still being pointed at by other cells in the same list or possibly a sub-list. In either situation, this results in what has been called the _dangling reference problem_. Figure 2.1 illustrates a simple example of this problem.

The dangling reference problem occurs when a cell is classified as "garbage" and returned to the free list but the cell still has pointers from allocated cells in the user's program. Why is this such a problem? In Figure 2.1, cells H, B, C, and E have been designated as "garbage" and are returned to the free list. When this happens, cells A, D, and F are said to have dangling references because they reference (i.e. have pointers to) non-existent cells. Now the problem is that whenever the storage allocator reuses cell B for the user's program, these dangling references will most likely cause undesired and unpredictable side effects (if pointers are not reinitialized).

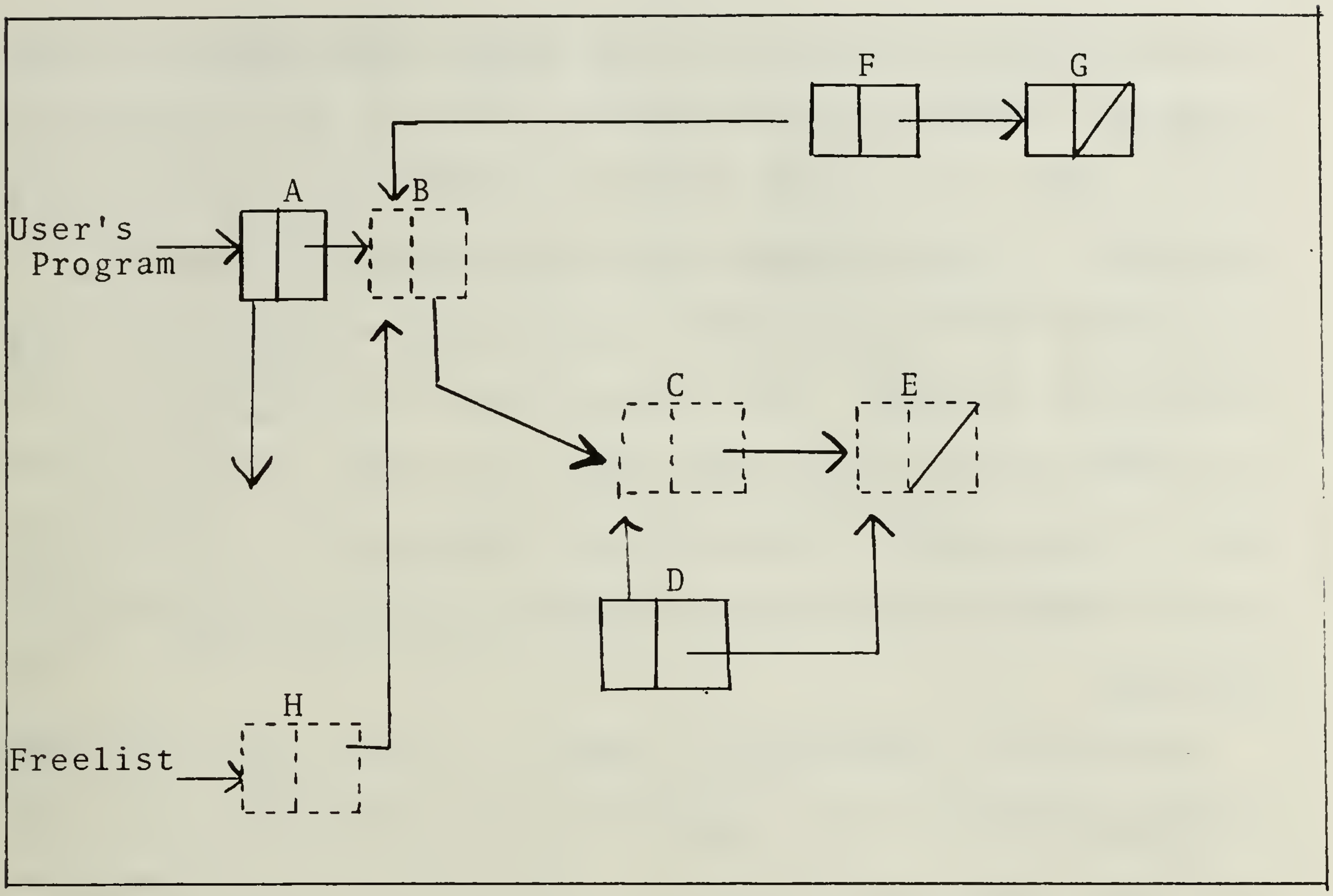


Figure 2.1 Dangling Reference Example.

Some LISP processors provide a feature allowing the user to invoke the storage collector. Normally, the storage reclamation method used is a type of garbage collection. This can be a very helpful feature if the programmer has a good idea of when would be the best time to invoke the collector.

In computing systems where the user has the capability to invoke the garbage collector, the programmer is concerned that there is sufficient memory for his program to run without "crashing." He does this by knowing how much memory his program has used and how much more memory is still available to his program. This is a difficult assignment in an interactive system where the number of users fluctuates and each user is trying to figure out the memory requirements for his program. Forgetting just one pointer to a cell that has been sent to the free list will result in the dangling reference problem. In LISP systems where the programmer determines allocation and deallocation, the function "<u>return</u>" can also cause this problem. The only way that a programmer can stop this problem from occurring is to take extreme care NOT to reuse any cell that he may think has a pointer to it. Again, this is just another thing for the programmer to remember.

Because of the dangling reference problem and the additional programmer burden, the MANUAL reclamation method in LISP or any other language that has such features is NOT considered an adequate selection for concurrent storage reclamation.

B. REFERENCE COUNT STORAGE RECLAMATION

The second method of automatic storage reclamation is called the "<u>reference</u> <u>count</u>" method. This means that the programmer's responsibility for both storage allocation and deallocation is eliminated and is automatically taken care of by the system. The reference count method of storage reclamation requires an additional field in each cell, which contains a <u>count</u> of how many other cells point to it. The count is a positive integer number, with a lower limit of zero. Whenever the numerical count in the cell's field becomes zero, then it is available to be used for users' programs. Cells are returned to the free list as soon as possible after they become inaccessible; consequently, "garbage" cells that are inaccessible and unavailable to be reused again will NEVER accumulate [Ref. 11: p. 443].

Do not confuse available with accessible! In LISP, a cell is accessible if it is pointed at or referenced by other accessible cells. The only cells that are directly accessible are those cells that are used by the system's interpreter, such as those cells that contain a user's program that is being interpreted and its corresponding association lists that represent the environments still in use [Ref. 11: p. 441]. A cell is "available" if it is on the free list, ready and available to be utilized by user programs.

Reference counts maintain track of the number of accessible references to each cell. Figure 2.2 illustrates an example of a simple LISP structure that contains cells with three fields each: the reference count field, and the left and right pointer fields.

In the reference count method whenever a cell is returned to the free list, it means that the cell's reference count field became zero and thus the cell is inaccessible. There is an additional operation required in reference counting in that whatever cells were pointed at by the cell that has been termed "garbage" and returned to the free list must also have their reference count fields decremented by one. This is because there is one less accessible cell reference to each.

This decrement operation may in turn cause any of these cell's reference count fields to also become zero and thus become inaccessible and available for return to the free list. In other words, decrementing a reference count could be a recursive operation [Ref. 11: p. 442].

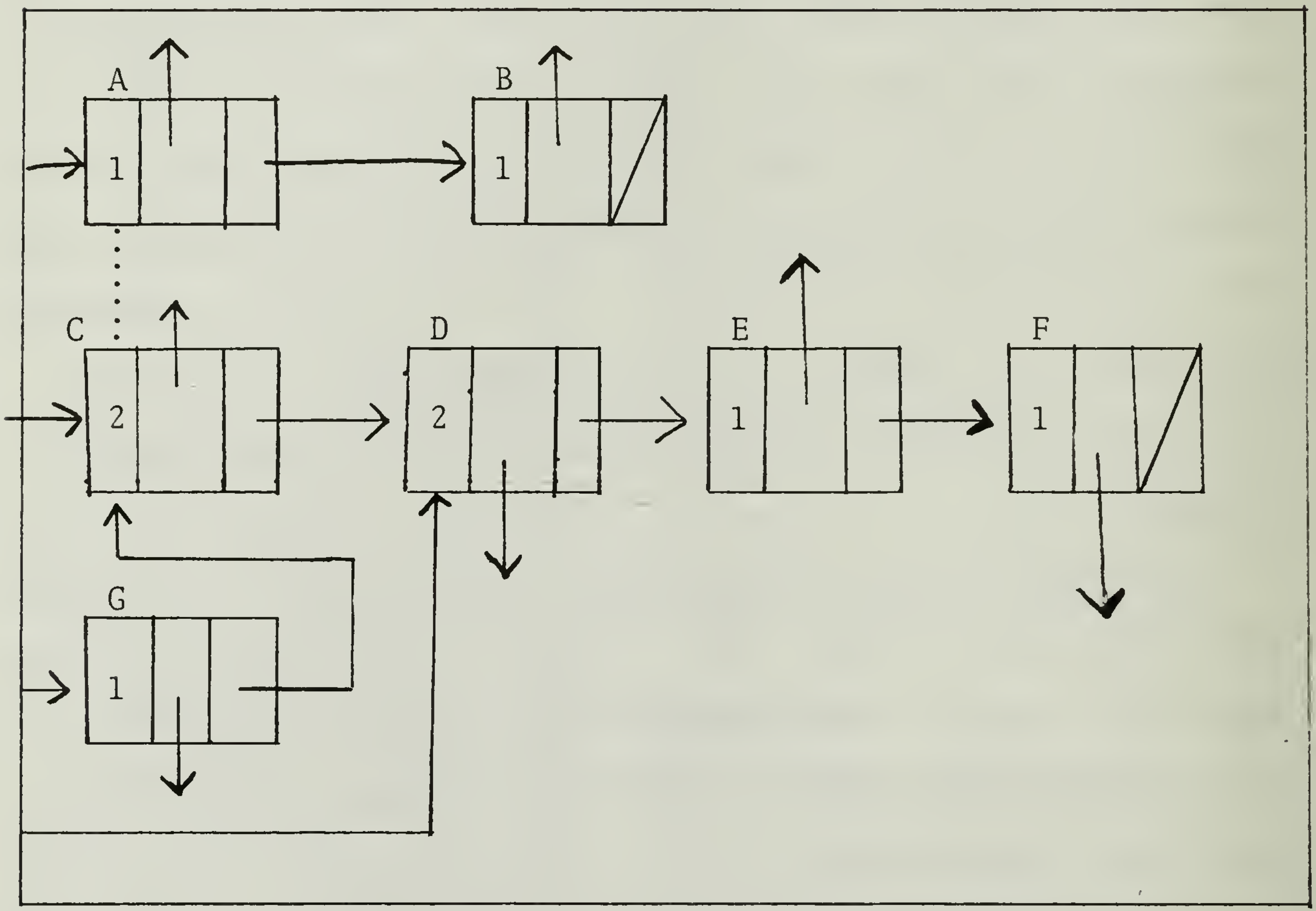


Figure 2.2 Reference Count Example.

There is a modification of the reference count technique suggested by Weizenbaum, that uses "doubly linked" list structures. In this method, a reference count is placed only in the header of each list (this means that there are no reference count fields in individual cells). Doubly linked lists were examined because they provide more efficient operations (e.g. transversing back and forth at will in examining doubly linked lists) than singly linked lists. Also in this method, the programmer plays a more active role in both the allocation and deallocation of memory. The programmer must remember the rules for maintaining reference counts for a list of cells so that he can avoid performing any operations that may refer to a particular list whose reference count has reached zero. Additionally, the programmer can explicitly override the reference count and return a specific list to the free list even before the reference count becomes zero. These are features that have to be utilized with caution because, in the hands of inexperienced programmers, they can cause side effects that are neither wanted nor constructive (e.g. dangling references). Weizenbaum's approach is excellent for the handling of list structures: whenever a list's reference count has become zero, then it is appended at the end of the free list. [Ref. 7: p. 412].

However, the time required to find the head may be extravagant and not worth the unnecessary delay. Besides being time consuming and increasing the programmer's burden, this reference count technique may prevent part of the list from being returned to the free list. Why? A part of this list may still be required by other lists in the user's program while the remaining part of the list is "garbage"; the problem lies with the part of the list that is still required, because this list must now be treated as a separate list with a new reference counter [Ref. 4: p. 501]. These are operations that are normally not provided in the doubly linked garbage collection method.

In LISP, as in other languages, reference counts must be maintained accurately and the status of all cells' reference counts being kept current. Whenever an additional reference to an accessible cell is made, that cell's reference count field must be increased by the number of additional references pointing to the cell. Similarly, whenever a reference to a particular cell is removed then that cell's reference count field must be decremented by the number of references removed. In LISP, there are two ways that a cell can be destroyed:

(1) A pointer can be overwritten by using an assignment operation, or by using the functions rplaca or rplacd [Ref. 11: p. 441].

(2) and the cell containing the pointer itself can become inaccessible [Ref. 11: p. 441].

In Figure 2.2 (reference count example), there is a possibility that no cells could be returned to the free list even if the cells were all inaccessible. This can happen whenever there is a cycle somewhere in the list structure, which means that there is a path from a cell back to itself. In a reference count system, cyclic data structures are NOT reclaimed, and these cells will be lost forever [Ref. 11: p. 442].

Reference counts provide a different approach to the problem of storage reclamation than the programmer responsibility technique. While it removes the programmer's burden, it causes several additional problems. The reference count method does not work at all in the case of a circular list, a list that is a sublist of itself. Figure 2.3 illustrates an example of a circular list structure. Reference counts do not work because the reference count field can never be reduced to zero, even when the entire list becomes inaccessible [Ref. 4: p. 501].

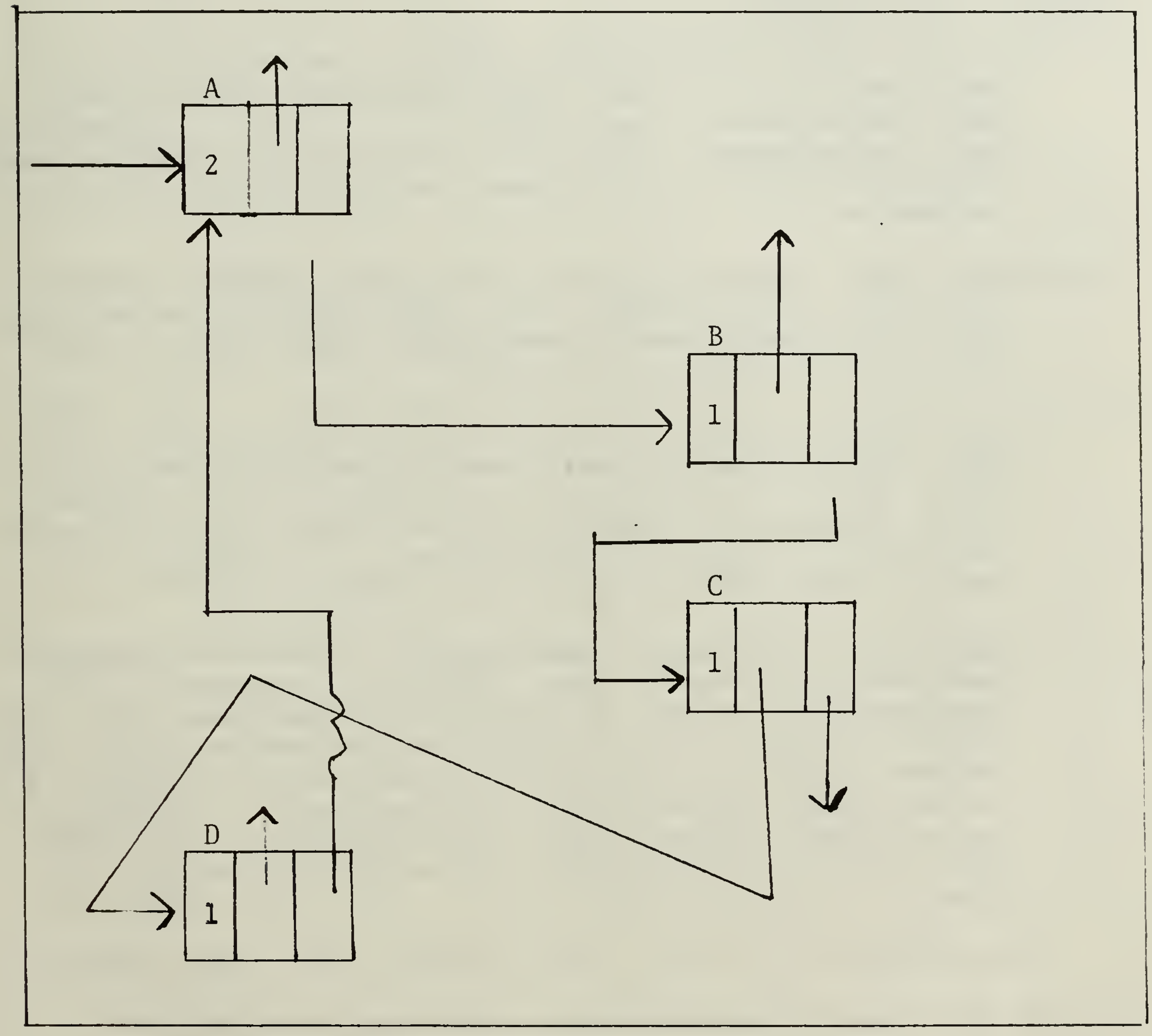


Figure 2.3 Circular List Structure.

In Figure 2.3 (circular list structure), each cell has again the same three fields as earlier discussed. This structure was created by using the rplacd function where the right pointer of cell D points to cell A. There is one and only one accessible path and that is from the pointer that points to cell A from the left direction.

There are other disadvantages in the reference count method. These are:

(1) It requires an additional field in each cell to serve as a counter. In small cells this may cost 25% or more extra memory space [Ref. 10: p. 495]. Theoretically, the reference count field in each cell must be large enough to handle the maximum number of cells that are in memory.

(2) The basic list processing primitives which create objects and copy pointers must spend their time updating the cells' reference count fields [Ref. 10: p. 495]. This is an expensive overhead.

(3) Reference counting does not always free all the cells that are available. Circular lists (i.e. lists that refer back to themselves) will NEVER have a reference count that will reach zero. This holds true even when no other list that is accessible to a "running" program points to them [Ref. 7: p. 412].

(4) Reference counting is unacceptable to use as a memory management scheme because there are "unbounded" delays whenever a cell is returned to the free list; this occurs because all successors of the returned cell may become "garbage" and should be returned to the free list at the same time when that original cell is being returned [Ref. 12: p. 112]. This again is a vast overhead to pay and may result in non-uniform execution times.

Measurements of actual LISP programs show that about 97% of all list cells have just one reference to them [Ref. 2: p. 351]. But because of its inability to handle cyclic structures, reference counting is NOT considered an adequate method for concurrent storage reclamation.

C. GARBAGE COLLECTION STORAGE RECLAMATION

The third and last method of storage reclamation is "garbage collection" and like reference counting is an automatic reclamation method. As previously stated, garbage collection is the process of reclaiming unused storage space. It is an automatic storage reclamation method that can handle circular data structures [Ref. 11: p. 443]. The basic garbage collection method requires an additional field in each cell, like that required for the reference count method. However, unlike the reference count method, the additional field is only a one-bit field. This one-bit is called the "mark" bit. The general idea of garbage collection is that a program continues to run without returning any cells to the free list until no more storage is available (this is called REGULAR garbage collection). When this happens, the program halts and a "recycling" algorithm uses the mark bits to first determine or mark which cells are "garbage" and then to sweep (return to the free list) all inaccessible or "garbage" cells. This creates memory space through the reuse of previously used cells to allow continuation of the user's program.

Regular garbage collection postpones the problem of storage reclamation until the free list cells is exhausted

[Ref. 10: p. 495]. When this occurs, the user's program is temporarily halted during which a garbage collector routine determines which cells are no longer accessible to the user's program and returns these cells to the free list where they can be reused again by the user's program. A simple way of viewing garbage collection is that storage reclamation is NOT a problem until there is no more memory for users' programs to utilize; and then with all available memory exhausted, the garbage collector becomes involved. Garbage collection, unlike reference counts, does reclaim cyclic structures. Normally, the garbage collector is an independent routine, relatively disjoint from the rest of the list-processing system [Ref. 10: p. 496].

Although garbage collection has several advantages, there are some problems with using garbage collection in a list-processing environment. The garbage collector has to scan all of memory in order to identify "garbage" cells and non-garbage cells. This scanning requires significant processing time. With LISP programs spending 10% to 30% of their time doing garbage collection, it is NOT unusual for a large LISP program to take 3-6 seconds to perform garbage collection [Ref. 5: p. 491]. If these run times were to be multiplied by a factor of 2-5 (or more) in order to obtain the corresponding "real" run times in a time-shared and interactive system, it can be seen that performing garbage collection can lead to delays that are inconvenient and excessive to users in an interactive system; and as programs continue to grow and become more complex and as memory sizes continue to expand, delays due to garbage collection may reach the state of becoming intolerable.

Just as the two previous storage reclamation methods had their disadvantages, there are some disadvantages in utilizing garbage collection. The most obvious one is that an additional "marking" bit is required in each cell. Additionally, garbage collection traditionally runs very slow when all of memory is in use. This is because the garbage collector must scan the entire memory area that is occupied by the user's program in order to identify "garbage" cells. In some instances, the number of reclaimed cells that are returned to the free list may not be worth the effort [Ref. 7: p. 412]. The <u>unpredictability</u> of when garbage collection occurs is a difficult design concept in a real-time system. In other words, when does garbage collection occur? It may not occur and normally does not occur at the same time every time it is invoked. Delaying program execution is another disadvantage, and combined with the unpredictability of garbage collection can be frustrating and dangerous. This results from the fact that the garbage collector is relatively disjoint from the rest of the list-processing functions [Ref. 10: p. 495].

As list-processing databases continue to grow, garbage collection problems will also continue to surface more frequently and become more noticeable to programmers (i.e. garbage collection will take longer and longer to complete). Depending on the size of memory, the delay caused by garbage collection is proportional to the amount of memory being used [Ref. 1: p. 522].

However even with the above disadvantages, garbage collection is a worthwhile and necessary activity, especially in a LISP environment. Garbage collection has always been needed because the amount of addressable space in memory has always been much less than the total space both required and used during execution of a list-processing program. Thus, garbage collection enables a "reusing" of the system's finite amount of addressable memory. Additionally, garbage collection reclaims cyclic or circular data structures. Garbage collection frees not only the programmer but also most of the list-processing primitives and functions from the concern about storage reclamation; now only those primitives that create new cells from the free list can invoke the garbage collector [Ref. 10: p. 495]. In LISP, the function <u>cons</u> requests new cells from the free list [Ref. 2: p. 351].

D. CONCLUSION

The major problems that arises when attempting to reclaim a part of a list structure (i.e. an individual cell or a list of cells) is knowing which part of the structure is "garbage" and which part is still required for the user's program [Ref. 10: p. 496]. Each of the above three methods of storage reclamation are responsible for this reclamation. Of the three methods, the remainder of this paper is devoted to garbage collection. Garbage collection removes the responsibility of reclaiming "garbage" from the programmer and allows him to concentrate on his program. Garbage collection also reclaims circular data structures which reference counts could not handle. Circular recursive structures are common features in a list-processing system, such as LISP.

Additionally, it has been convenient to classify garbage collection according to the size of the cells that are used by the users' program and reclaimed by the garbage collector [Ref. 2: p. 343]. LISP cells illustrate the problems that are involved in marking and sweeping "single-size" cells or cells that are of the same size (i.e. all cells have the same number of fields and each field has the same number of bits per cell). This paper will discuss the LISP garbage collection methods; consequently, single-sized cells are assumed to be the "norm."

The next chapter examines the basic process and techniques used in garbage collection.

III. GARBAGE COLLECTION PROCESS

As discussed in the previous chapter, garbage collection is a method of reclaiming unused areas of the memory that are being used and allotted to the users' programs. Garbage collection will be referred to as "GC" throughout the remainder of this paper.

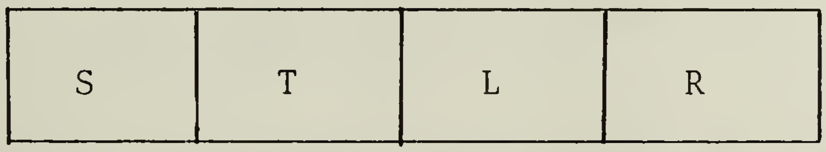
There are two basic phases that constitute GC. These have been termed the "marking" and the "sweeping" phases. Marking is the process of transiting through memory and identifying any cells that may be reclaimed. Sweeping is the process of incorporating these cells into the memory area that is available to the user. This memory area is called the free list.

A. PHASE ONE: MARKING

This phase is usually performed with the garbage collector maintaining a list of immediately accessible cells; when the garbage collector is activated, the links (or pointers) that connect one cell to another are traced (followed) and every accessible cell is then marked.

In the marking phase, all mark bits are assumed set to zero (usually by the preceding sweep phase). The phase starts from the beginning of memory and from these cells that are accessible to the user's program. Cells that are no longer required by the program are called "garbage." To save time, a list of immediately accessible cells is maintained and the garbage collector simply traces the links of those cells that are on this list. This ensures that every single accessible cell is marked and unmarked cells are thus ready to be reclaimed.

For illustration purposes, each LISP cell has two fields that contain pointers to other cells. Other LISP implementations may have other cell formats. The left cell field can be found by using the function car and the right cell field can be found by using the function cdr. Each cell additionally has two boolean (one-bit) fields; one that is used to differentiate between atomic and non-atomic cells and the other field is used for marking [Ref. 2: p. 343]. Figure 3.1 illustrates such a LISP cell.



Legend

S(sign): "mark" bit used in GC
0 = garbage
1 = non-garbage

T(type):
T = 0 indicates atomic
T = 1 indicates non-atomic

Left, Right: pointers to other cells/lists or NIL

Note: Full word of data associated with this cell (by pointers) when T = 0

Figure 3.1 LISP Cell Format.

The "marking" algorithm is recursive and is able to reclaim circular list structures. But being recursive also means that memory space is required (i.e. a stack) in order to place and thus save the accessible cells. This sounds like a "catch-22" in that the garbage collector was originally called because there was no more memory area for the users' programs. How then can there be memory available to handle the recursive marking algorithm when the garbage collector was called? This situation becomes a problem because the marking algorithm is operating entirely in main memory and not secondary storage. To solve this problem, several algorithms have been proposed. All these algorithms reduce the required storage by trading it for longer time needed to perform the marking phase (whereas most recursive marking algorithms reserve some memory space for a fixed-length stack, which is used for the marking). [Ref. 2: p. 345].

One algorithm is the <u>Deutsch-Schorr-Waite</u> algorithm, in which the cells of a list structure are traced and inspected without having to use a stack. This algorithm reverses successive links until either the leaves (i.e. atoms) in the structure or cells that have already been visited are found. The <u>link reversal</u> is then undone by reconstructing the original list structure. All cells are visited three times. The additional visit and the overhead to restore all pointers and for inspecting and marking the bits render this method less efficient. Additionally, this algorithm requires the use of another bit field in each cell. This bit is called the "tag bit", and indicates the direction in which the restoration of the reversed links should proceed (i.e. whether to follow the left or right pointer field in each cell) [Ref. 2: p. 344].

Another marking algorithm that solves the above problem is that proposed by Kurokawa. His algorithm uses a fixed-length stack and a tag bit (similar to Deutsch-Schorr-Waite's tag bit). When the stack overflows, some of the pointers from the stack are deleted but the information is preserved by turning on the tag bit of the unstacked cells: these cells form a chain (the pointer to this chain is left on the stack). Removal of stack cells makes space available to continue the marking phase. Whenever a pointer is removed from the stack, it is examined to determine whether the cell it points to is tagged. If it is, then the linked tagged cells are retraced [Ref. 2: p. 345].

Figure 3.2 illustrates the marking phase of garbage collection. The dashed line reflects the sequential trace through the cells in memory. For additional information, see [Ref. 11].

By the end of the marking phase, ALL accessible cells have been marked and are ready to be reclaimed. The mark phase basically determines which cells remain accessible to the users' programs and which cells are not needed by the user's

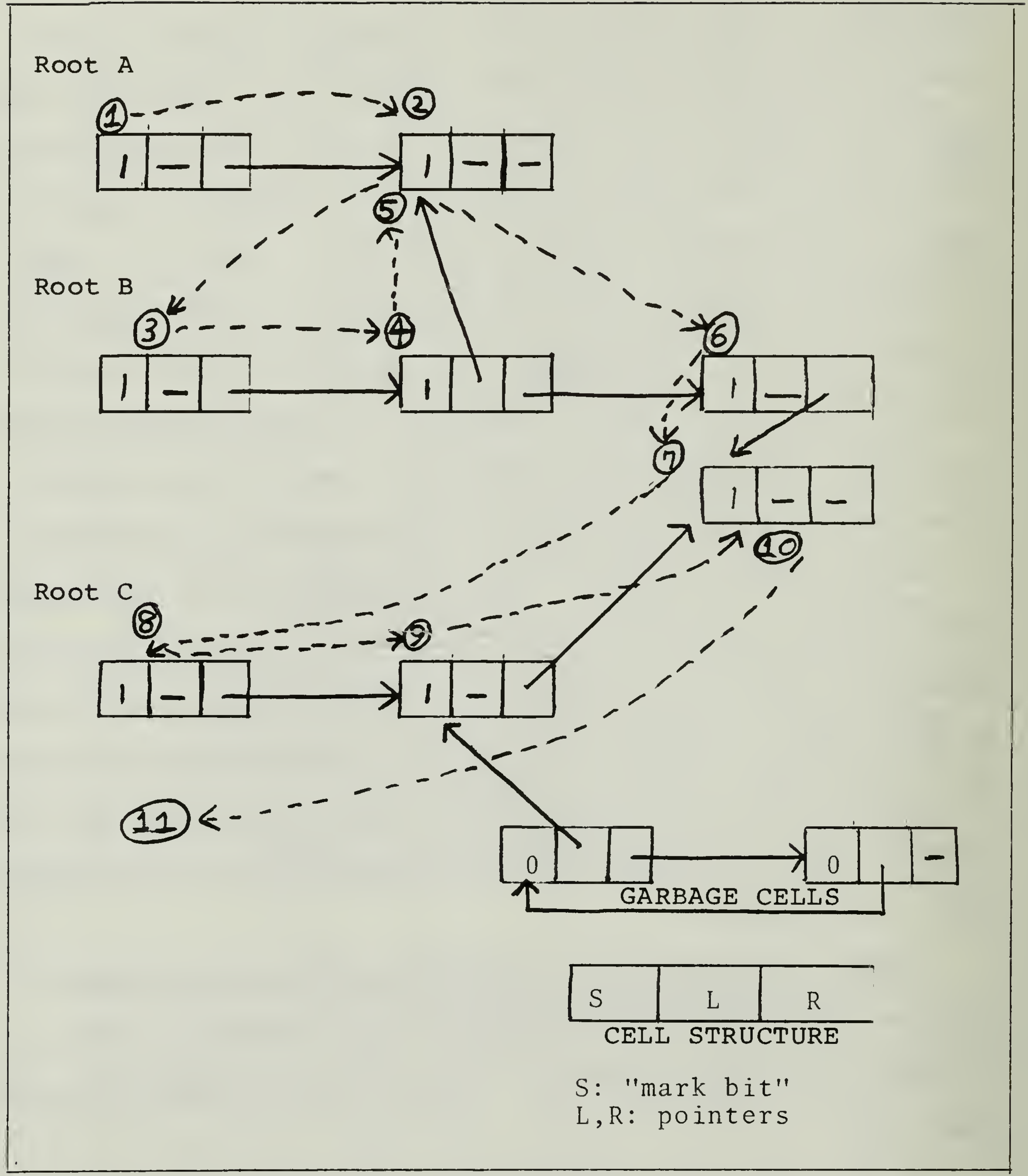


Figure 3.2 Marking Phase Example.

program and thus are "garbage." This reclamation is the function of the second phase of garbage collection: sweeping.

B. PHASE TWO: SWEEPING

The marking phase traced and identified ALL non-garbage cells, after having started from those cells that were immediately accessible to the user's program. The sweeping phase now makes a sequential pass in memory of all the cells that were traced by the marking phase and incorporates ALL unmarked cells onto the free list in memory. In other words, ALL unmarked cells are concluded to be "garbage" and are appended to the free list, in which available cells are linked by pointers [Ref. 11: p. 445].

The simplest method for reclaiming the marked cells consists of linearly sweeping through memory. Each cell is visited in order (i.e. sweeping through memory). If a cell is unmarked then it is inaccessible to the user and can now be linked onto the free list. However, if the cell is marked, then it is accessible and its mark bit is reset back to zero in preparation for the next marking phase of garbage collection. The garbage collector moves on to the next cell in order, repeating this process until there are no more cells to "sweep." Thus, there are two sequential passes through memory: one for the marking phase and the other for the sweeping phase.

The sweeping phase is usually separated into 2 distinct subphases. These are:

(1) Incorporation of all available cells, that are linked or connected to other cells by pointers into a free list.

(2) Compaction, which will be explained later, is where all unused cells are "moved" to one end of memory while the other end of memory contains those cells that are currently accessible and being utilized by users' programs.

The incorporation of cells being returned (sweeping phase) to the free list can be subdivided into specific function areas, depending on whether compaction is required or not. If compaction is not required, then the sweep phase just consists of removing all cells identified as garbage and placing them anywhere on the free list. However, if compaction is used then ALL available cells are compacted into one contiguous area in memory; compaction also necessitates the updating of all pointer references to the cells that have been relocated. Compaction prevents "thrashing" and performs better (significant time gains) than a non-compacting garbage collector. This paper will be concerned with garbage collection with compaction.

C. COMPACTION

As mentioned earlier, when compaction is considered a desirable feature of the garbage collector, memory is basically divided into two areas; compaction moves all available cells to one end of memory (containing the free list) and the other end of memory contains accessible cells that are being utilized by users' programs. There are several areas of concern that a "compacting" GC scheme has to be concerned with.

(1) It must know exactly where each cell is to be moved to.

(2) All pointers in use must be adjusted to the "planned" address of the cells that they reference.

(3) and all cells in use must be moved to its "planned" address [Ref. 13: p. 204].

Compacting garbage collectors are more complicated than non-compacting garbage collectors. Most garbage collectors today are non-compacting [Ref. 13: p. 204], because of the increased overhead in using "compaction." Compaction is really an option in GC. It can be invoked only when needed. Although speed of the garbage collector with compaction is not crucial, it must be efficient and should make a minimum of demands on the storage area itself. If compaction is an option and is not invoked a great many times, why then should we be concerned with compaction in the first place?

In its bare and simplest version (no compaction), GC reclaims unused or discarded cells while leaving the areas where the cells are located exactly where they are physically located in memory. Eventually, a snapshot of memory after GC has occurred will look like a piece of swiss cheese in that the memory area will be fragmented. Consequently, there may be a situation where there is no single space or area in memory large enough to accommodate or satisfy a user's request even though the total amount of available memory space is large enough. Therefore, one solution to this potential problem, is compaction.

Compacting memory areas basically means that all unused areas of memory are moved (compacted) to one end of memory and the other end of memory is reserved for the areas that are currently being utilized by users' programs. The resulting list structure has the identical topological structure as its old structure [Ref. 13: p. 204].

Compacting GC has been shown to have significant time gains in terms of performance in LISP programs [Ref. 2: p. 345]. Even though GC with compaction is more costly than GC without compaction, its advantage is that it indirectly helps to reduce the number of transfers from secondary storage in virtual memory systems. In other words, "thrashing" is greatly reduced.

The decision whether or not to use GC with compaction should be based upon the ratio between the total amount of computation time and the amount of time the processor spends in GC. If the ratio is small, then compaction is unnecessary [Ref. 14: p. 26].

With the garbage collector maintaining the free list, it is possible to incorporate the returned garbage cells in one contiguous area in memory through compaction. Compaction becomes a concern of the programmer ONLY if program execution terminates due to insufficient memory, even though garbage collection has taken place.

There are several types of compaction, classified by the relative positions in which cells are left after compaction. These are:

(1) Arbitrary -- cells that originally point to each other do not necessarily occupy contiguous memory positions after compaction. Arbitrary compaction has also been called the two-pointer scheme because the garbage collector uses two pointers: one pointer sweeps from the top address and the other pointer sweeps from the bottom address in memory. When the top pointer reaches a garbage cell and the bottom pointer reaches an accessible cell, then the accessible cell is moved to the garbage cell's address. The process ends when the two pointers meet. This has an additional overhead involved in the updating of the pointers but it is a simple technique to describe and implement [Ref. 2: p. 343].

(2) Linearizing -- Cells that originally point to each other have adjacent memory positions after compaction.

(3) Sliding -- Cells are moved toward one end of memory without altering their linear or relative order. This is usually applicable for heaps of odd size cells; but this technique also requires the destruction of occupied cells before all pointers to them have been updated [Ref. 2: p. 343].

(4) Copying -- This is a compaction method in which the garbage collector creates a second storage area for each cell that is to be compacted and then copies these cells from the old area to this second area. This method is normally reserved for virtual memory systems because of the extra memory areas required for copying.

D. CONCLUSION

The compaction method that is normally used and one that will be assumed to be the type of GC Compaction technique in the remainder of this paper is the two-pointer or arbitrary method. Memory is scanned two times. In the first scan, two pointers are used as previously discussed. When an accessible cell is moved to its new location in memory, its mark-bit is turned off to await the next GC cycle. The second scan of memory is needed for pointer readjustment: this is because some cells have been moved and now have new memory addresses. It is critical to update all pointers to any obsolete cell locations. If this was not done then the <u>dangling</u> <u>reference</u> problem can and usually will occur. The second scan also scans only the compacted area. Pointers are readjusted whenever they point to cells that have been moved to new memory locations.

For the remainder of this paper, whenever garbage collection is mentioned, compaction is assumed to be one of its features. This assumption takes into account the fact that in systems that utilize single-size cells, garbage collection compaction is not really a problem except on VM systems.

Whether or not compaction is used with GC, the same original problem of suspension of program execution remains. HOW this problem can be eliminated will be discussed in the following chapter, where parallel (concurrent) GC methods and techniques are examined.

IV. PARALLEL/CONCURRENT GARBAGE COLLECTION

Parallel garbage collection is one of the more desirable approaches for eliminating the periodic and unpredictable suspension of program execution in a list processing system; it is also desirable in an interactive and real-time system [Ref. 15: p. 1]. For example, being able to implement parallel garbage collection in LISP will enable simultaneous list processing and execution of user programs, thus eliminating the user having to be concerned about his program halting because of lack of memory space. The garbage collection process will be taking place without the system's users being aware of it.

It is difficult for an interactive and real-time list processing environment to provide satisfactory service when operations must be halted to allow GC to occur. As programs become larger, as list processing systems become larger and more complex, as list processing databases become larger, and as the GC takes longer and longer to perform, this difficulty becomes worse and more noticeable to the users [Ref. 16: p. 113].

A. INTRODUCTION

A GC system that ensures users' programs are never suspended due to lack of available memory has been termed a real-time garbage collection system [Ref. 5: p. 491]. This is because it provides real-time responses to each user's request.

A real-time GC system avoids suspension of list processing operations and does NOT halt program execution. Clarification between real-time garbage collection and regular garbage collection is needed to avoid any confusion in readings about garbage collection. The only real difference between these two types of garbage collection is WHEN the GC process occurs. In "real-time" GC, garbage collection and program execution run concurrently (i.e. at the same time); while in "regular" GC, the user's program executes until it runs out of memory when it halts and waits for the GC process to reclaim some memory.

Attention in this paper is devoted to "real-time" garbage collection and NOT "regular" garbage collection. The terms parallel or concurrent garbage collection will be used interchangeably throughout this paper.

In a computing system that has a significantly large memory, which is not an uncommon occurrence today, GC can become very expensive in both time and money due to its requirements to scan and sweep through this entire large memory to identify (mark) and return (sweep) all garbage cells. In an interactive LISP system, WHEN garbage collection is actually occurring becomes very apparent to the user, especially when his program halts and the system then informs him that there is no more memory available for him to use. This can be very frustrating to the users on that system. How then can this programmer frustration be eliminated? As previously stated, one of the possible solutions is through _parallel_ garbage collection.

With this parallel approach, garbage collection occurs simultaneously with program execution. One method to eliminate the problem of the temporarily halting of users' programs (so as to allow GC to occur) would be to construct faster hardware (i.e. processors, buses, disk drives, etc.); this would allow faster and larger transactions between main memory and secondary store. In other words, one solution is a fast system where paging occurs and where the garbage collector is speeded up by implementing it in _microcode_ as a primitive operation of the list-processing computer. However, this hardware method does have some limitations. While the current trend of memory sizes continues to increase, it is very doubtful that it is possible to increase processor speed to the point of eliminating GC time, so that it is unnoticeable to the users of the system [Ref. 5: p. 492].

But even if the garbage collector were to be microprogrammed in the hardware of the computing system, GC would still be necessary; and the GC process may even still necessitate some halting of program execution. Additional methods and techniques to allow GC to occur simultaneously with program execution are presently available. If the selected GC method for a particular system (with or without compaction) is still desired to have the goal of eliminating the halting of user programs due to lack of available memory, then the only satisfactory way to ensure that program execution is not halted while garbage collection is taking place is through parallelism or concurrency [Ref. 10: p. 496].

The efficiency of parallel GC will play a major role in the widespread use of list-processing systems [Ref. 3: p. 1143]. There are several techniques to implement concurrent GC, each of which are subsequently explained. Because the focus of this paper is on non-virtual memory systems, the Virtual Memory (VM) approach is only briefly described. These concurrent GC methods are:

(1) two processor method
(2) time slice method
(3) dynamic processor allocation method
(4) Virtual Memory (VM) method

B. TWO PROCESSOR METHOD

In this method, two separate and distinct processors are used. The first processor, termed the "collector", is responsible only for garbage collection. The second processor, termed the "mutator", is responsible for program execution [Ref. 2: p. 354]. Having two processors available allows parallel operations to take place. In other words, the collector can be conducting GC at the same time that the mutator is performing list processing tasks. The collector operates in such a manner that the free list never becomes empty, resulting in there not being any noticeable delay caused by garbage collection [Ref. 5: p. 492]. Both processors share a common memory.

(1) The "Collector" -- The collector performs the previously described tasks of marking and incorporating (sweeping) all unmarked cells into the free list. It collects the list structures that the mutator has discarded [Ref. 5: p. 493]. While these GC tasks are occurring, the mutator remains active and continues its own operations of computation proper.

(2) The "Mutator" -- The mutator provides all storage required by users' programs but the mutator can not request cells from the free list until the collector makes them available. The mutator "mutates" (processes) list structures in memory [Ref. 5: p. 492].

C. TIME SLICE METHOD

Probably the simplest method to eliminate the halting of program execution while GC is occuring is to <u>time-share</u> or <u>time-slice</u> one processor between list processing operations and garbage collection tasks. The GC could be done at times that would not be inconvenient or noticeable to the user; such as during typing (keyboard inputting) in a interactive system or at other times that could be setup or scheduled by a system clock. GC could be conducted at regularly scheduled intervals and the duration of GC could be determined by the system's characteristics (i.e. number of users, time of the day, job priorities, etc.).

If the GC is to be conducted on a regular basis, then the duration of each GC cycle could be programmed to be relatively small; this would of course depend on several criteria of which one of the more important is the size of memory. The larger the size of memory, the less the number of times necessary to be set aside for GC (i.e. the garbage collector would be called less frequently). Alternatively, the system could be designed so as to reduce the length of each time slice. But no matter how often GC is conducted or how long a time slice is, it is important to remember that GC could be temporarily suspended at any time and later resumed [Ref. 5: p. 492]. This suspension of garbage collection would be controlled by the system operators and not the system users. Of course, this suspension of GC is dependent on the fact that sufficient memory always remains to allow program continuation. Although this would alleviate but not necessarily eliminate the aggravating problem of users waiting for GC, it would introduce still another problem, that of increased overhead in <u>context switching</u>, and still may not result in any net gain of system speedup [Ref. 10: p. 496].

There currently exist algorithms for both <u>time sharing</u> the operations of both the collector and mutator processors and the <u>two processor</u> scheme mentioned in the previous section. These algorithms demand a greater percentage of the processing time than does <u>regular</u> garbage collection. Additionally, the results from these algorithms have shown that the garbage collector with one processor being time-shared wastes (i.e. does not use) more processing time than if there were two processors dedicated to separate operations and functions. The main reason for this is that the time allotted for the garbage collection in a time-sharing environment must proceed even if there was no demand for it. A system that time-shares one processor between list-processing tasks and garbage collection tasks has specific times set aside for both these tasks. These times are created by the system designers at design time. For example, if a one processor system were designed to conduct garbage collection every 100 milliseconds and there was no necessity for GC during one or more of these 100 millisecond allotted GC times, then there is an apparent waste of processing time; because the processor could be doing productive work (list-processing) rather than idly waiting for the allotted garbage collection time to expire so it could conduct list-processing [Ref. 2: p. 355].

If the design decision for a particular computing system was made to have only one processor to SHARE the list processing and the garbage collection duties, then the dynamic processor allocation method might be more appropriate.

D. DYNAMIC PROCESSOR ALLOCATION METHOD

This technique involves dynamically allocating one processor to both garbage collection tasks and to list processing operations as real-time needs dictate [Ref. 10: p. 496]. However, if the trend of decreasing hardware costs including processor costs continues, it would be more practical to devote one processor strictly to GC requirements and devote another processor to list-processing operations, even if there were times that there would be no need for GC (resulting in one processor being idle part of the time). But having ONE processor, being used either in a time-shared or dynamically allocated method, would reduce the overall system cost and eliminate synchronization problems between the two processors. In addition to the previously mentioned "time-slice" method, the other method of using only one processor to conduct garbage collection would be to dynamically share this processor between GC tasks and list-processing tasks. Steele has stated that dynamic processor allocation could be utilized to achieve a fair performance level [Ref. 10: p. 496].

Dynamic processor allocation is similar to the time-shared scheme in that there is only one processor being utilized for both GC and list-processing tasks; however there is no conception of time with the dynamic processor allocation method (i.e. the time at which the garbage collector takes control of the processor operations is NOT pre-established or predesigned). Rather, the garbage collector is only invoked WHEN absolutely necessary. The big implementation question with this method is this "WHEN."

The dynamically allocated processor method is different from the regular GC methods found in most LISP processors in that the algorithm that decides this "WHEN" is more complex and is developed with special considerations in mind: how much memory remains available to the users, job priorities, job sizes, etc. One implementation method for the dynamic method, could be that the garbage collector would be invoked at times that the users would not find it harmful to their program's execution. In an interactive system, these times (that GC could occur) may be based on when users are utilizing a "read-only" file or when users first check into the system. The point is that there is no set or established time for the garbage collector to take control of the system in the dynamically allocated processor method.

E. VIRTUAL MEMORY (VM) METHOD

This is a GC scheme, first proposed by Baker, that operates and collects garbage in a virtual memory system. In this method, secondary storage is utilized through paging. The basic concept is that available memory is divided into two areas called semispaces. These two areas are allowed to grow from opposite ends. The sizes of the two semispaces vary at execution time and the moving of accessible cells is done whenever a new cell is requested [Ref. 2: p. 363]. One area (resembling a stack that uses contiguous locations for the users' programs) is reserved for the list processor. The other area is available for providing new cells (also from contiguous locations). These two areas respectively resemble the previously mentioned working list and the free list.

Bobrow and Murphy have shown that the use of a selective cons, which is LISP's function that requests a cell from the allocator, can improve the efficiency of subsequent list-processing and GC operations [Ref. 2: p. 351]. In a VM system, each time that a cell is requested (i.e. a cons is executed) a fixed number of cells are moved from one semi-space to the other.

Compaction is necessary for garbage collection in this method in order to avoid thrashing. For example, it has been advocated to keep one free list per page. In a paging environment, the extra memory that is needed is of less importance than the size of the working list. Since the moved cells are <u>compacted</u>, page faults are more likely to be minimized. The moving of cells during a <u>cons</u> execution corresponds to the cell tracing in regular garbage collection. This method distributes some of the GC tasks during list-processing operations; this guarantees that actual garbage collection CANNOT last more than a fixed or tolerable time: i.e. the time necessary to flip semispaces and readjust a fixed number of pointers declared in the user's program [Ref. 2: p. 355].

For the purpose of this paper, the virtual memory method is NOT reviewed in depth nor is it considered a worthwhile task to pursue to determine if it is an adequate method for concurrent garbage collection. This decision is based on the fact that systems with very large memories currently abound and are becoming more commonplace as time passes on.

F. CONCLUSION

The successful implementation of a list-processing system with parallel garbage collection provides a strong foothold for parallel list evaluation [Ref. 15: p. 8]. List processing and list evaluation is exactly what LISP provides.

Typical large LISP jobs may spend 10%-30% of their time in garbage collection tasks. Running GC in <u>parallel</u> could cut the total real time for a given task by close to this amount, without requiring the user to plan explicitly for parallelism. Since the cost of CPUs has been steadily dropping, it would be practical to devote one processor to GC even if it would be idle part of the time [Ref. 10: p. 496].

Although <u>dynamic</u> processor allocation and <u>time-sharing</u> processor allocation have some advantages and disadvantages, the <u>Two Processor</u> implementation method is the one that the remainder of this paper is devoted to. The complexity of the algorithm for the dynamic processor method and the likely possibility that the garbage collector may at times not be performing any useful operations in the time-sharing method preclude either of these two methods from providing the <u>optimum</u> solution for a parallel garbage collection scheme. Additionally, the VM method is not pursued because of the ready availability of computing systems with large memories.

With the current trend of decreasing hardware costs, it is practical and cost-effective to utilize the Two-Processor method: one processor devoted to list-processing tasks and the other processor strictly devoted to garbage collection tasks. This decision takes into account the advantages and disadvantages offered by the other methods. Both the two processor method and the time-shared method require at most twice as much processing power as regular garbage collection, but, since the list processor no longer has to perform garbage collection, there is a net speedup in the list-processing operations [Ref. 5: p. 492]. As the cost of processors is steadily decreasing, the feasibility of conducting GC with two processors should increase with time. As the cost of hardware continues to fall, one more processor in a system would not be infeasible. In fact, having this processor devoted entirely to GC would in the long haul provide more advantages and would alleviate user frustration; but more importantly, it would eliminate the halting of execution of user programs while GC is taking place because of lack of available memory.

The specific operation of the mutator processor in the succeeding chapters will NOT be discussed in detail because basically it is just a typical, currently existing type processor and studying its operation would reveal no significant or new techniques for the operation of the collector.

Instead the collector processor is examined because it has unique and different characteristics and is devoted entirely to the task of garbage collection. Synchronization between the two processors will be looked at.

Because of the numerous algorithms and implementations that currently exist for concurrent garbage collection, only one hardware and one software method of the Two Processor method will be examined. However, the selected hardware and software methods are both considered to be the ones that are used in comparison with other garbage collection methods.

These two methods for two processor allocation with concurrent garbage collection are examined in the next chapter.

V. "TWO PROCESSOR" GARBAGE COLLECTION IMPLEMENTATION

A garbage collection cycle is defined as the total execution time required for both the marking and sweeping (reclaiming or scanning) phases. For the concurrent GC system, there exists a requirement (primarily system efficiency) to reduce the length of the garbage collection cycle [Ref. 16: p. 113].

Implementing concurrent GC in a LISP environment with the Two Processor Method raises an important question: Is it feasible to utilize hardware or software techniques or possibly a hardware and software combination to achieve concurrent garbage collection? This question will be examined in this chapter.

There are numerous methods and algorithms proposed for implementing a parallel garbage collection with two processors. Some methods are: Ben-Ari's two-color scheme [Ref. 17], Dijkstra's three-color schme [Ref. 18], Kung and Song's four-color scheme [Ref. 3], and Steele's special coded or micro-coded scheme [Ref. 10]. However, this paper will only concentrate on reviewing two classic approaches to parallel garbage collection:

(1) Dijkstra's Software Implementation Method
(2) and Steele's Hardware Method.

These two methods were chosen for review not only because they represent some of the earliest ideas on the problem of garbage collection, but also because each proposed a different type of solution.

A. DIJKSTRA'S METHOD (SOFTWARE APPROACH)

Dijkstra's algorithm for parallel garbage collection requires two distinct and separate processors, called the <u>mutator</u> and the <u>collector</u>. Both processors operate from and on a common memory from which cells required for a user's program are allocated and deallocated. The collector continually executes a two-stage cycle while maintaining the free list of available cells for the mutator [Ref. 3: p. 1144]. The mutator is dedicated to executing the user's list-processing program and the collector is responsible for garbage collection tasks.

With two processors operating on the same memory there are sure to be coordination and synchronization problems whenever the collector is marking the in-use cells on the working list at the same time that the mutator is modifying cells on the same working list. One solution to this problem (and the one that Dijkstra's approach proposes) is to utilize <u>colors</u>, which indicate the status of each cell; in particular, a third color (gray) is used during the mark stage to ensure all in-use cells are marked.

Dijkstra's software-oriented method for concurrent GC with two processors has sometimes been called "Dijkstra's three-color" garbage collection algorithm [Ref. 3: p. 1143]. In other literature, it has been referred to as an "on-the-fly" garbage collector. On-the-fly garbage collection is a system that allows concurrent execution of both the mutator and the collector [Ref. 17: p. 334].

The three colors for cell marking proposed by Dijkstra are:

(1) "white": indicates unmarked or unused

(2) "black": indicates marked or used

(3) and "gray": indicates that the cell has been requested to be used by the user's program [Ref. 2: p. 354].

Garbage collection is performed by a two-stage cycle (i.e. a two-phase algorithm): the collector marks all cells accessible from the root and then appends all unmarked (and hence inaccessible) cells to the free list. The basic algorithm continually executes mark and sweep phases that are similar to the phases discussed in Chapter 3. The algorithm that is used applies a simple scan and mark method for mark propagation. However, the marking phase is a little more complex than the mark phase in regular GC systems because two mark bits are now required instead of one mark bit. Two mark bits are used because a cell can be in any of three states, which correspond to the above three colors.

A description and illustration of the collector's marking phase and collecting/appending phase is provided later in this chapter.

Each cell in memory contains the following:

(1) A data field.

(2) Two fields that contain pointers to other cells.

(3) and A two-bit color field that is used by the collector to mark cells either black, white, or gray [Ref. 3: p. 1144].

Figure 5.1 illustrates a typical cell. Notice that this is a different cell configuration than that in Chapter 3, and is not necessarily the only configuration of a typical LISP cell.

All cells contain the two extra bits for marking the cell as black, white or gray. During the mark phase, the collector blackens all cells that are accessible to the mutator. During the scan phase, all white cells are returned to the free list and all black cells are whitened.

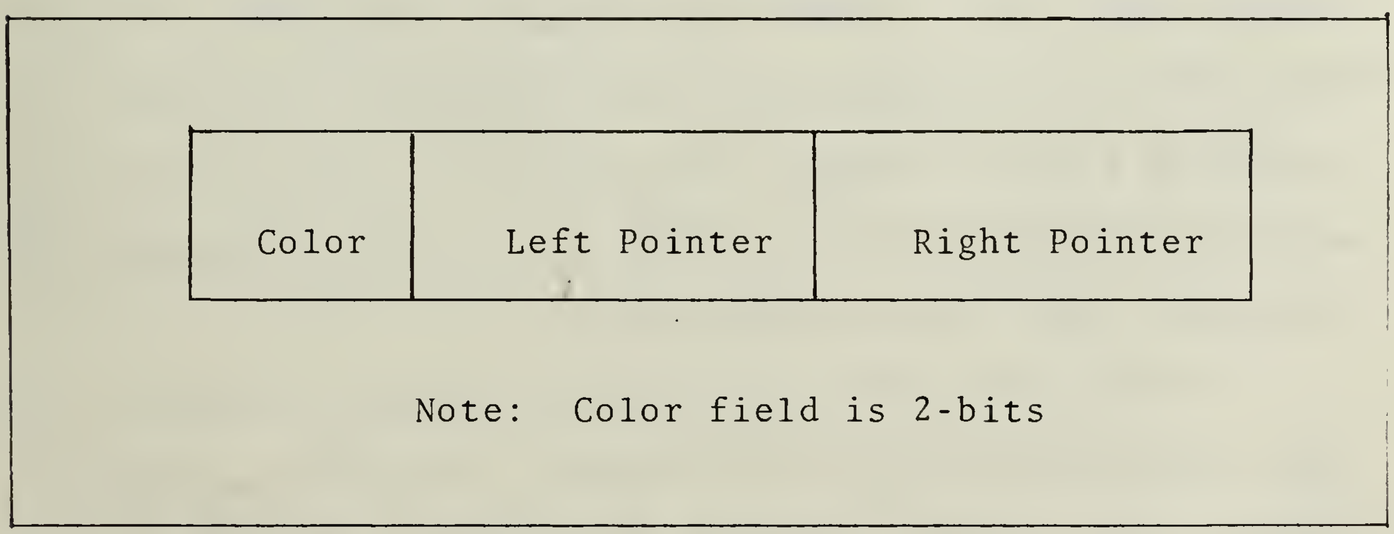


Figure 5.1 Three-Color Cell Format.

Located within main memory, which is common to both processors, are two distinguished cells:

(1) One cell points to the beginning and end (i.e. head and tail) of a linked list of cells that are available for users' programs. This linked list is similar to the previously discussed free list.

(2) The second cell also points to a list: a working list of cells that are currently being used by the mutator. A cell is accessible if it is located within the working list; and a cell is in-use if it is either accessible or located on the free list [Ref. 3: p. 1144].

In a compacting GC system, these two lists are located at opposite ends of memory. The mutator and the collector tasks and responsibilities are subsequently discussed.

1. Mutator Tasks/Responsibilities

The mutator uses primitive operations that allow it to change a pointer of any accessible cell to point to any other accessible cell, including NIL. The mutator operations include pointing to a new cell that is removed from the head of the free list after that cell was requested by a user's program [Ref. 3: p. 1144]. In essence, the mutator is responsible for conducting the list-processing operations required by a user's program. The mutator is doing useful work while the collector is collecting and recycling garbage cells to permit their reuse by the mutator.

The color gray is used during the mark stage to ensure that all accessible cells are marked. The mutator helps the marking phase of the collector by changing a "white" cell to "gray" when that cell is requested and used by a user's program [Ref. 2: p. 354]. Cells that are black are used by the mutator to perform list-processing in the user's program.

All other cells are white (i.e. cells that are not being used or requested from the free list by the mutator are white). Cells, that are no longer being used by the mutator and have not been already reclaimed by the collector, will eventually be returned to the free list.

For synchronization purposes (between the mutator and the collector), there is a restriction placed on the mutator's operations. During the marking phase, no cell can ever become "lighter" [Ref. 18: p. 969]. In other words, whenever the mutator changes a pointer in the pointer field of a cell, that cell must be shaded. Shading is a primitive operation that turns a white cell to gray. It has no effect on black cells or other gray cells [Ref. 3: p. 1144].

The mutator will initiate an interruption and will halt only when the free list is reduced to one cell. It resumes list-processing when the collector returns additional cells to the free list (i.e. the free list is now greater than one cell) [Ref. 2: p. 354]. It has been determined that the probability that the number of available cells on the free list will ever contain only one cell is very low [Ref. 17: p. 336]. This is because the algorithms for the mutator and the collector are designed to avoid this situation through the use of the two "distinguished" cells that exist in memory.

Removal of cells is done at one end of the free list and appending of garbage cells occurs at the other end of the free list. The algorithms for the two processors are designed to take into account the required synchronization between the mutator and the collector so as to avoid having only one cell on the free list. But if the free list were ever reduced to ONE cell, then the mutator would have to wait until the collector returns more cells to the free list. Again, this situation will occur very infrequently.

The mutator's execution can result in cells that are neither on the working list (pointers are now removed) or are NOT yet on the free list. These cells are the "garbage" cells which will be reclaimed by the collector.

2. Collector Tasks/Responsibilities

The basic function of the collector is to identify "garbage" cells and collect them at the end of the free list in memory by repeatedly executing the two-stage cycle. This cycle first marks (blackens) all cells that are in-use. Then it appends all the unmarked cells (white cells), during a linear scan of memory, to the free list; next it unmarks or whitens the marked cells (black cells) in preparation for the beginning of the next GC cycle [Ref. 3: p. 1144]. Figure 5.2 and Figure 5.3 illustrate the above two-stage cycle of the collector. Appending a garbage cell to the free list is the collector's ONLY modification of the existing topology (i.e. the shape of the data structure) [Ref. 18: p. 969].

How are the collector and mutator synchronized? In other words, how is it possible to ensure that the collector or mutator does not change a nongarbage cell (black) to white or does not change a garbage cell (white) to nongarbage (black)? As previously mentioned, the algorithms are designed to ensure this necessary synchronization. Additionally, the use of the third color ("gray") also helps in providing synchronization between the two processors.

There are some constraints placed on the mutator. The mutator can only shade cells (i.e. white cells to gray cells) and it can only change pointers in the cells' fields to point ONLY to other already accessible cells.

At the end of the marking phase, there are no gray cells. The absence of white reachable cells prevents the mutator from introducing gray cells while the absence of gray cells prevents the collector from doing so [Ref. 18: p. 971]. The marking and the appending phases of the collector are reviewed next.

a. Marking Phase of the Collector

The collector marks or colors: the used cells of a user's program black, the cells in the free list white, and all other cells gray.

Initially, before any cells are allocated to a user's program, all cells are white [Ref. 18: p. 969]. The collector accomplishes the marking task by initially "graying" the first used cell on both the working list and the first cell on the free list (i.e. the roots of both the free list and the working list). The collector then selects a gray cell and tracing proceeds by the collector graying any other cells that are linked to this gray cell (this selection of the initial gray cell is dependent on the particular algorithm implementation, but the simplest method to implement is to have the collector utilize the first cell it encounters during its sequential scan of memory). Those cells that are linked to this gray cell, including the initial gray cell, are then "blackened." All cells that are still white after the marking phase is completed will be "garbage" [Ref. 18: p. 969]. Figure 5.2 provides an illustration of the above operations.

Tracing through memory is complete when all the white cells are incorporated into the free list and all black cells are whitened in preparation for the next mark phase. During the next cycle of the collector, these white cells are swept to the tail of the free list [Ref. 3: p. 1144].

Every garbage cell is eventually appended to the free list. In other words, no garbage cell will ever remain unreclaimed or uncollected for more than two consecutive garbage collection cycles. With a GC cycle consisting of a marking phase followed by a sweeping/appending phase, it is impossible to guarantee that each sweeping phase will collect and append all garbage cells that existed at its start. This is because new garbage could have been created between the current sweeping phase and the preceding marking phase [Ref. 18: p. 968].

Figure 5.2 displays memory with respect to time as the collector performs the marking phase. The only cell fields illustrated in Figure 5.2 (and Figure 5.3) are the left and right pointer fields. Cells are either initially on the free list or the working list or are "garbage" that has not yet been appended to the free list. Cells 1 and 2 are marked garbage cells ("marked" garbage is explained later in this chapter). Cells 3-5 constitute the initial working list, cells 6-9 constitute the initial free list, and cells 10 and 11 are unmarked garbage cells not yet appended to the free list ("unmarked" garbage is explained later in this chapter). Cell 5 is the starting cell (initial cell) for the mutator and cell 6 is the starting cell of the free list.

The various subphases of the collector's marking phase (as illustrated in Figure 5.2) are:

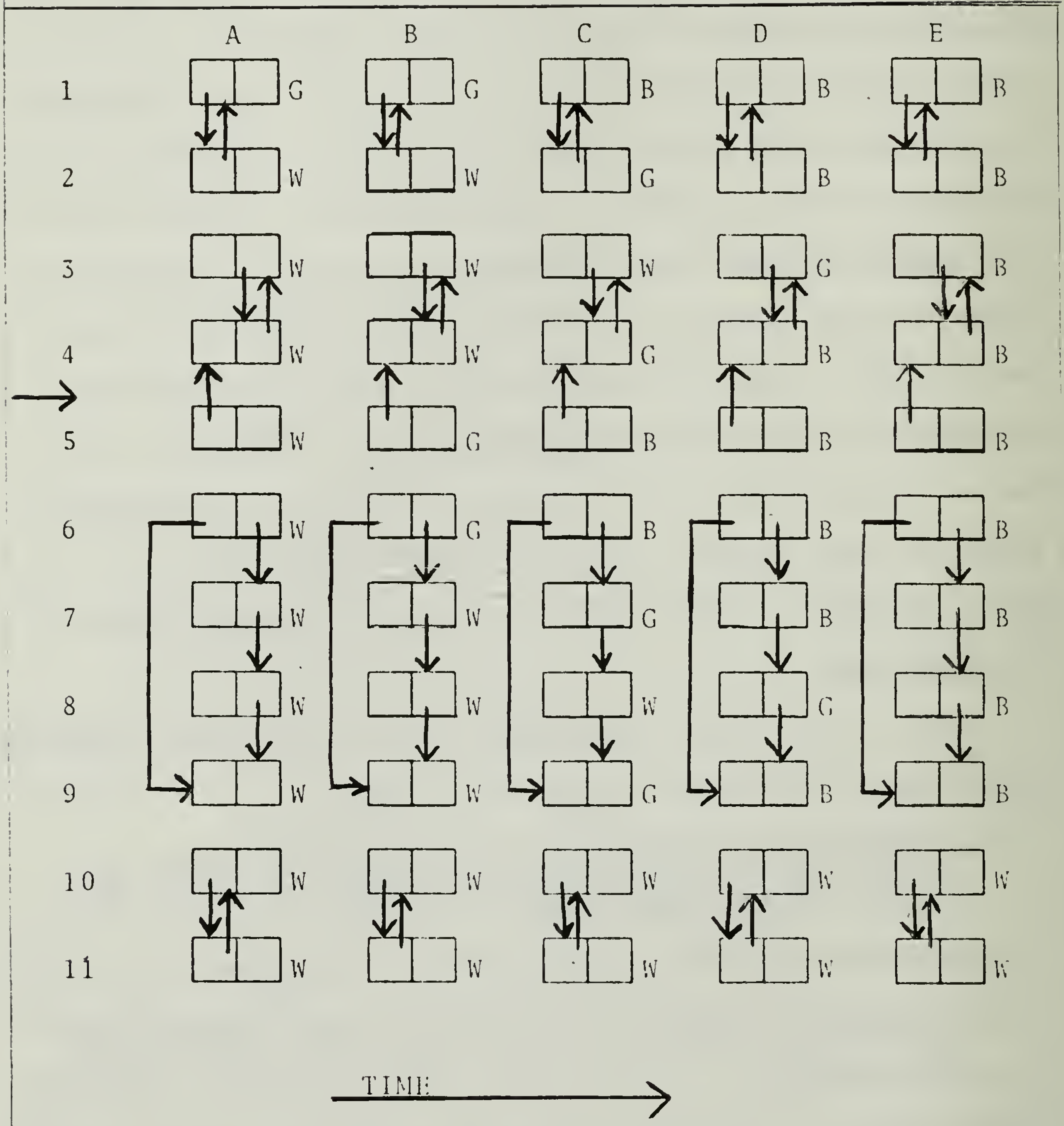
(1) Subphase A is the beginning of the mark stage (see Subphases G and H in Figure 5.3 for the reason why cell 1 is initially gray).

(2) Subphase B grays the roots (cells 5 and 6).

(3) Subphase C indicates the halfway point through the mark stage (grays the sons of the root and blackens all grays).

(4) Subphase D darkens cells (blackens gray cells and grays white cells) through the use of primitives. This subphase operation is performed to ensure that all used cells and all requested cells are marked in preparation for the collector's sweeping phase.

(5) Subphase E is the end of the mark stage. Cells 10 and 11 are "unmarked" garbage cells which will be collected during the scan phase of the current GC cycle.



Legend:
W-white
G-gray
B-black

Cell 5 = Initial cell of the mutator
Cell 6 = Initial cell of the free list

Figure 5.2 Memory Snapshot During Mark Phase.

The operations of the collector and the mutator preserve Dijkstra's invariance property:

"Every white in-use cell can be reached from a gray cell along a path passing through white cells exclusively."

No gray cells present implies that all in-use cells are marked black; and when this happens, it signifies the end of the mark phase [Ref. 3: p. 1144].

b. Collecting/Sweeping Phase of the Collector

This is Dijkstra's scanning phase. It is also the collecting, sweeping, and the appending phase. This phase returns garbage cells to the free list. Although it was not originally designed to include the compaction feature, it can also be implemented to relocate cells for compaction, with all pointers being updated as necessary. At the end of this phase, all unmarked cells are eventually returned to the free list; and compaction of cells can occur as they returned to the free list.

When the marking phase is completed, there exist two different types of garbage cells:

(1) Unmarked garbage cells which are collected during the scan phase of the current GC cycle. These cells are called "quick" garbage.

(2) and Marked garbage cells which are not collected until the scan phase of the next collection cycle. These cells are called "slow" or "floating" garbage [Ref. 3: p. 1144].

Differentiation is made between the two types of garbage cells to illustrate that there may be occasions that some marked cells become garbage while the garbage collector is still in its marking phase: these garbage cells will remain uncollected until the next garbage collection cycle.

Memory is examined in a linear or sequential fashion. White cells are now appended to the free list and black cells are whitened in preparation for the next mark phase.

Figure 5.3 illustrates the collector's scanning phase in memory with respect to time. It shows memory from where the last mark phase ended (see Figure 5.2) to the beginning of the next mark phase.

The various subphases of the collector's collecting/appending phase (as illustrated in Figure 5.3) are:

(1) Subphase F is the end of the mark phase and the beginning of the scan phase.

(2) Subphase G is the halfway point through the scan phase. The right pointer from cell 3 to cell 4 was dropped and cell 3's left pointer adjoined while cell 4 was still black (hence, the mandatory shading operation had no effect on cell 4). If this pointer manipulation occurred after cell 4 was scanned and whitened, cell 4 would now be gray rather than white [Ref. 3: p. 1145]. This situation occurs because of some primitive operations in the user's program that are taking place at the same time as garbage collection is occurring.

(3) Subphase H shows memory immediately after the scan phase. Cell 3 is gray because the right pointer from cell 4 to cell 3 has been dropped and cell 4's left pointer has been adjoined; this is the same reason why cell 1 in Figure 5.2 (Subphase A) is initially gray. Cells 7 and 8 are now being used by the mutator (i.e. the user's program). This subphase is also the beginning of the NEXT mark phase if compaction is not a feature of the garbage collection process. White cells 4 and 5 ("marked" garbage cells) will be appended to the free list during the next GC cycle.

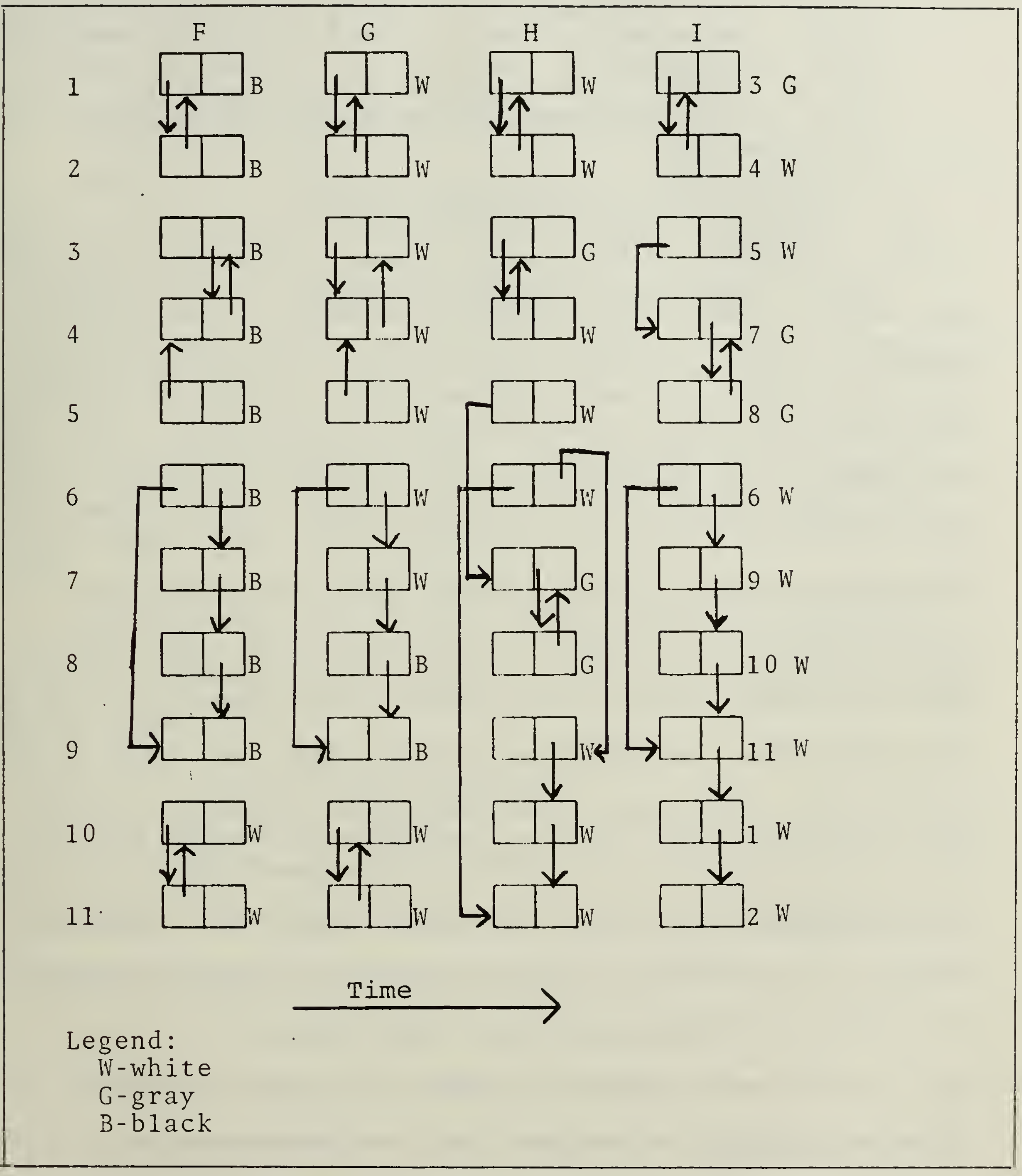


Figure 5.3 Memory Snapshot During the Scan Phase.

(4) Subphase I illustrates memory if compaction were a feature of the system. Cells 6, 9, 10, 11, 1, and 2 represent the free list with cell 6 being the head of the free list and cell 2 being the tail of the free list. White cells 4 and 5 ("marked" garbage cells) will be appended to the free list during the next GC cycle. Cells 7 and 8 are on the working list. Cell 3 is in the same situation as cell 1 in Figure 5.2 (Subphase A).

3. Conclusion of Dijkstra's Approach

Dijkstra's method uses software techniques: two bits per cell are used for coloring or marking of the cells in memory. No use of semaphores or other interprocessor communication is needed with Dijkstra's method [Ref. 3: p. 1144]. The mutator and the collector are separate processors, each with specific tasks and responsibilities. The synchronization and coordination features between these two processors are incorporated in the design of the respective algorithms, which are more complex than that of regular garbage collection. The algorithms' descriptions in this paper have been greatly simplified. The purposes of discussing Dijkstra's method are to provide a view into one classical method of conducting concurrent garbage collection and to show that concurrent garbage collection is possible by using software techniques. For additional information, see [Ref. 18].

The next section examines a different approach: that of using hardware techniques for the implementation of concurrent garbage collection with two processors. It is not the objective of this paper to compare the two methods of using two processors to accomplish concurrent garbage collection; instead, it is the intention to determine the <u>feasibility</u> of using software or hardware methods to accomplish this goal.

B. STEELE'S METHOD (HARDWARE APPROACH)

Steele's algorithm for parallel garbage collection, like Dijkstra's approach, uses two processors, called the <u>list processor</u> and the <u>garbage collection processor</u>. While Dijkstra's method does not explicitly provide for compaction, although it could be implemented, Steele's method does explicitly allow for compaction. <u>Compaction</u> is done using the two-pointer technique previously discussed [Ref. 2: p. 354].

Each processor has its own stack, which are used for temporary storage and recursion. The list processor's stack is used for:

(1) List structure manipulation.

(2) and Temporary variables, such as local variables, for LISP functions.

The garbage collection processor's stack is used for:

(1) Marking of the accessible list structure.

(2) and recursive tracing [Ref. 10: p. 497].

One of the goals with Steele's approach was to keep the list processor's overhead to a minimum by using necessary synchronization features between the two processors [Ref. 19: p. 50]. <u>Semaphores</u> are used for this interprocessor communication and synchronization.

Both processors operate from and on a common memory from which cells required for users' programs are allocated and deallocated. The cells that are located in memory are similar to the cell's configuration previously discussed: the cells' fields may contain pointers to other cells.

Steele introduced a new concept in the internal organization and representation within memory. Cells are organized into sets called spaces. A space is nothing more than an ordered sequence of cells [Ref. 10: p. 497]. The cells are of the same size and of the same format; in other words, the cells are homogenous and therefore the spaces are also homogeneous. There is no consideration of cells of various sizes.

Steele's garbage collector makes exclusive use of semaphores and requires two bits per cell: the mark bit and the flag bit. These two bits are used NOT only for the required marking of cells but also for compaction and readjustment of pointers in the cells' fields [Ref. 2: p. 355].

A "space" has two pointers associated with it. These pointers are called freelist and lastfree, which respectively point to the first and last cells in a linked list of available cells within that particular space. These two pointers resemble Dijkstra's two distinguished cells located in memory; except thate there may be many freelist and lastfree pointers depending on the number of spaces; while in Dijkstra's approach, there are only two such distinguished cells. These two pointers are actually structures that each have two components:

(1) A space indicator.

(2) and an integer which is a valid index within that space's sequence of cells. In other words, this integer identifies a particular cell within a space [Ref. 10: p. 497].

The size of the spaces is predetermined by the system designers: it is a system design decision. The concept of these combined spaces resembles the notion of the free list and the working list previously discussed.

The synchronization between the two processors and the tasks of the two processors is subsequently discussed.

1. Synchronization of the Two Processors

As it was with Dijkstra's software approach, the synchronization of the two processors is necessary because the GC processor may be relocating cells within a space (which is required for garbage collection and compaction) at the same time that the list processor is trying to operate on these same cells. The way that Steele's approach handles this problem is through the use of the flag bit and semaphores. This flag bit is the key principle that allows the garbage collector processor to relocate cells with the list processor being aware of it [Ref. 10: p. 498]. Before the list processor performs any operation on a cell, it first must check the flag bit of that cell. If that bit is set to true then it means that the cell has been relocated and that the new address of that cell is in the first component field of that cell

[Ref. 10: p. 498]. Steele calls this process "normalization" (to assist in the processor synchronization, Steele incorporated the normalization process into a single function). The garbage collector processor sets the flag bit after relocating the cell.

Additionally, the "P" and "V" semaphore primitives defined earlier by Dijkstra are used for synchronization [Ref. 10: p. 498]. Steele carried the idea of the above semaphores to a higher degree in that more complex synchronization operators were defined (in terms of the "P" and "V" primitives). These semaphore primitives permit exclusive access to a single cell for only one processor at a time but in doing so the entire space where that cell is located is NOT locked out to the other processor.

Semaphores are used to interlock access to the freelists. For efficiency and to provide a greater degree of synchronization, it is necessary to have a freelist interlocked as little as necessary. Synchronization is accomplished by having a pointer to the last cell of the free list and stipulating that the garbage collector processor can only append cells to the end of the free list and the list processor can only remove cells from the beginning of the free list. With this method a free list itself is interlocked with a semaphore ONLY when the free list is reduced to one cell. This situation will occur very seldom in practice [Ref. 10: p. 499].

Each processor also has a global register which contains the address of the particular cell that a processor wants exclusive access to. If the first processor tries to use that cell before the second processor has released it, then the second processor will loop (busy wait) until it can have access to it [Ref. 10: p. 498]. Furthermore, it is necessary to keep the garbage collector from shifting from one phase to another phase while the list processor is trying to determine what phase it is in. To accomplish this goal, Steele uses two global "variables", _gcstatesem_ and _gcstate_ (the former is a semaphore controlling access to the latter, which in turn has as its value: mark, relocate, update, or reclaim). If "gcstatesem" were executed by the list processor, then the garbage collector is prevented from changing phases [Ref. 10: p. 499].

All spaces, cells, pointers, and semaphores can be operated on by either processor. However for synchronization purposes, some constraints and restrictions must be placed on the processor operations. Neither processor can push or pop cells from a stack at the same time that the other processor is doing so because this could result in erroneous stack indexes. The solution to this problem is to use another semaphore with each stack. Additionally, the list processor can access and modify ONLY accessible cells. This is accomplished by having the list processor follow the following rule:

> If the list processor modifies a marked cell during the garbage collector's mark phase, it must ensure that the garbage collector reexamines that cell and any other cells that may be affected by the original cell modification [Ref. 10: p. 499].

Each processor has its own internal registers that can be used as "temporaries." However, neither processor can examine the other processor's internal registers [Ref. 10: p. 498].

2. The Garbage Collector Processor

The two bits in each cell (the "mark" and the "flag" bits) determine the current status of a particular cell with respect to the current phase of the garbage collector. These two bits and the use of these bits by the two processors provide one of the biggest reasons why concurrent garbage collection is possible.

Table I provides a summary of the meanings of these two bits. For additional information, see [Ref. 10].

The garbage collector repeatedly executes in sequence the following four phases that comprise garbage collection with compaction:

(1) Mark Phase -- A simple recursive and trace method for locating all accessible cells. Through the use of one of the global variables, the garbage collector processor locks out the list processor during this phase.

(2) Relocate Phase -- Uses the two-pointer scheme previously discussed (see Table I).

TABLE I MEANINGS OF THE MARK AND FLAG BITS

Mark bit Flag bit	false false	false true	true false	true true
Mark phase	Cell not yet seen by mark and trace routine.	(Does not occur during mark phase)	Cell seen by mark and trace routine Cell is therefore accessible.	Cell on freelist. Should not be seen by mark and trace routine.
Relocate phase	Discarded cell.May be used to relocate an accessible object into if necessary.	Relocated cell. First pointer component indicates new location.	Accessible cell. May be relocated into new place if necessary.	Cell on freelist. Ignored by relocate phase.
Update phase	Discarded cell. Ignored by update phase.	Relocated cell. Ignored by update phase.	Accessible cell. Pointer components may need to be normalized.	Cell on freelist. Ignored by update phase.
Reclaim phase	Discarded cell. May be returned to freelist.	Relocated cell, now discarded. May be returned to freelist.	Accessible cell. Ignored by reclaim phase.	Cell on freelist. Ignored by reclaim phase.

(3) Update Phase -- Sweeps over each space and normalizes all pointers in all accessible cells and also on the list processor's stack. Combined with the relocate phase, this constitutes the required compaction.

(4) Reclaim Phase -- Sweeps over all spaces and looks for cells with their mark bit set to false (see Table I).

3. The List Processor

The list processor accomplishes its task of being able to continue list-processing at the same time that garbage collection is taking place through the creation and use of list manipulation primitives. Some of the more common of these primitives are:

(1) Creation of new cells from the list available cells. In LISP, this is accomplished by cons.

(2) Selection of cells' components. In LISP, this is accomplished by car and cdr.

(3) Determination which space a cell belongs to. In LISP, this is atom.

(4) Comparison of pointers,where pointers identify a particular cell within a space. In LISP, this is eq [Ref. 10: p. 502].

C. CONCLUSION

Steele's approach provides a method to achieve concurrent garbage collection using specialized hardware (i.e. microcoded processors). His use of semaphores to assist in synchronization is in itself not a new concept. But his use of the flag bit that basically serves as an indirection marker that permits the garbage collector processor to relocate cells within the same space as the list processor is operating is a new idea.

Two different approaches to achieve parallel or concurrent garbage collection have been briefly discussed. Dijkstra's method proposes software techniques while Steele's approach proposes hardware techniques. Which method is better depends on many factors, such as the size of the system, especially main memory and secondary store; the speed of the processor(s); the number of users on the system (average and worst-case); which high level language is being used (for example, LISP); the cost of the system, etc.

Table II illustrates the major characteristics of the two methods. Steele's algorithm, which is more complicated than Dijkstra's algorithm (mainly because of its compacting capability), requires all active cells to be bound in a stack. Dijkstra's algorithm uses multicolored marking (white, gray, and black). The gray nodes inform the garbage collector of list modification occurrences. The garbage collector counts not-gray nodes during the mark propagating phase. The existence of gray nodes, when the garbage collector examines the count, indicates that the list processor has performed some list modifications. Dijkstra's algorithm is obliged to apply a simple scan and mark method for mark propagation (because it must correctly count not-gray nodes) [Ref. 15: p. 6]. For additional information, see [Ref. 2].

It is recommended that a technique using hardware technology be utilized to implement <u>concurrent</u> garbage collection because hardware costs are continuing to decrease, while software costs are continually increasing.

TABLE II PARALLEL AND REAL-TIME GARBAGE COLLECTION

ALGORITHM	STORAGE REQUIRED	COMMENTS
Dijkstra	No stack; 2 bits/cell.	Uses a free list; Main objective is to prove correctness.
Steele	Stack; 2 bits/cell, several semaphores.	Designed to be microcoded; does compaction.

It has been shown with the two previously reviewed garbage collection methods of Dijkstra and Steele that concurrent GC is feasible, including having the feature of compaction. There are numerous other methods and techniques to implement concurrent GC. The next chapter briefly looks at some of these other methods and the future possibilities of concurrent garbage collection.

VI. OTHER ALGORITHMS/FUTURE POSSIBILITIES

A. INTRODUCTION

Algorithms for concurrent garbage collection are too numerous to list, let alone describe. Instead, this chapter will briefly introduce several other concurrent garbage collection methods and some future possibilities in the area of garbage collection. These methods are not necessarily restricted to two processors nor do they have compaction as a feature. Additionally, the configurations of the LISP cells for these algorithms differ from those presented in this paper.

B. OTHER ALGORITHMS

The first algorithm is that proposed by Lamport. Although efficiency of garbage collection systems has been basically ignored in this paper, it remains an important consideration in the design of any computing system. Lamport informally proposed that possibly the best way to improve the efficiency of a garbage collection system is to use several processors for garbage collection. Lamport's algorithm allows any number of processors to be used in a GC system, both as mutators and as collectors [Ref. 3: p. 1153].

Lamport's proposal of using multiple list processors for concurrent garbage collection solved two problems:

(1) Concurrent execution of multiple mutator processes.

(2) Increasing the speed of the garbage collector.

These two problems were solved through the use of constraints placed on the processors and through parallelizing the garbage collection phases. Lamport's goal was to keep the overhead of the mutator to a minimum; consequently, no unnecessary synchronization between the collector and the mutators was introduced [Ref. 19: p. 50].

Lamport's algorithm guarantees to mark any structure. It requires a two-bit field in each node to allow marking. In a multiple garbage collector system, Lamport's algorithm effectively requires each garbage collector to access a physically separate portion of the node space. Each garbage collector examines nodes until ONE collector finds and marks an accessible (shaded) node and then it shades its successors. All garbage collectors are then reset, restarting the search of their assigned node space. Consequently, with most list structures, only one garbage collector does any useful work between "resets" [Ref. 20: p. 367].

A substantially improved version of Lamport's algorithm is obtained by allowing each garbage collector to complete its sequential pass through its assigned section of the node space before resetting, instead of resetting as soon as one garbage collector has found and "colored" a node. This version has two advantages:

(1) Several garbage collectors may find shaded nodes on each pass, be able to mark them, and shade their successors.

(2) Successors to a marked node may themselves be marked in the same pass through the node space [Ref. 20: p. 368].

Kung and Song's algorithm is introduced next.

Kung and Song's algorithm, which is a variant of Dijkstra's three-color collector, uses FOUR colors (white, off-white, gray, and black) and has two independent processors that share a common memory; one processor for list-processing and the other for GC. This algorithm requires that the nodes of the free list always be off-white; this requirement shortens the marking stage because the nodes on the free list do not have to be marked (i.e. the free list does not need to be traced) [Ref. 3: p. 1145]. Each cell contains three fields: a left and right pointer field and a color field.

Kung and Song's algorithm also introduces a deque, which permits application of a recursive trace and mark method (this is different from Dijkstra's simple scan and mark method). The insert operations to the deque increases the overhead of the list processor, creating a new problem of how to allocate the deque [Ref. 15: p. 1]. Although usage of a deque for the marking phase increases the complexity of proving the correctness of the system, a deque is used in order to avoid possible access conflicts, since both processors may manipulate the deque at the same time [Ref. 15: p. 1].

As previously stated, efficiency has not been specifically addressed in this paper, and neither has system performance.

Kung and Song's algorithm satisfies the following properties, which are important to the system's performance:

(1) The time to perform the marking phase by the garbage collector is independent of the size of memory, and depends only on the number of active nodes.

(2) The nodes on the free list do not have to be marked during the garbage collector's marking phase.

(3) Only two active bits for encoding four colors are required for each node.

(4) Minimum overheads for the list processor are introduced.

(Dijkstra's method does not satisfy properties one and two above) [Ref. 21: p. 120].

Kung and Song's system is designed such that the marking phase has an execution time proportional to the number of active nodes and independent of size of the memory. The results of their parallel garbage collection system show that a parallel garbage collector is usually significantly more efficient in terms of storage and time than a regular garbage collector [Ref. 21: p. 120].

The next method of concurrent GC is that introduced by Bonar and Levitan.

Bonar and Levitan's method uses specialized hardware, Content Addressable Memory (CAM), that creates a very fast real-time LISP system (they define a real-time list-processing system as having the property that the time required by each of the elementary operations is bounded by a constant independent of the number of cells in use). Each word contains one "simplified" LISP cell with only three fields: a left (car) and right (cdr)field, which can both point to another LISP cell; and a garbage one-bit field (that indicates if the cell is free). The key observation about GC with such a cell is that pointers to any given cell can be located with only two CAM operations:

(1) A CAM search of the left fields of all cells in memory.

(2) A CAM search of the right fields of all cells in memory.

When a free cell is needed, a CAM search is executed (using constructed routines or algorithms), for a cell whose garbage bit is set. Their algorithm requires that all cells be initialized with the garbage bits set and the left and right pointer fields set to nil. A part of the list structure becomes potential garbage when any one of may pointers to it is deleted: this can occur by using the functions replaca and replacd, that respectively replace pointers in the left (car) and the right (cdr) fields of LISP cells. CAMS are used to examine all memory cells in parallel and are constructed in such a manner that each word (a "word" is an inherent regular sub-structure of CAMS) can compare its contents, rather than its address, with a value broadcast by the CPU. This comparison process is done by all CAM words simultaneously. The CPU then interrogates the CAM to discover which words, if any, match the broadcast value [Ref. 12: p. 112].

Some advantages of this implementation are:

(1) It performs all elementary operations in real-time.

(2) All cells are available for use.

(3) Even though the cost of a CAM has been estimated to be 1.5-3.0 times the cost of an equivalent size RAM, it is well suited for sufficiently inexpensive implementation with VLSI technology.

(4) It retrieves the correct value for a name associatively, requiring only two CAM operations.

(5) Strings and other dynamic data types (in addition to the simplified LISP cells discussed above) can be elegantly and efficiently integrated into the basic scheme without partitioning memory.

Some disadvantages of this system are:

(1) Circular lists cannot be easily collected.

(2) Does not support a virtual memory environment [Ref. 12: p. 116].

Hibino's method is the next and last additional method of concurrent garbage collection introduced in this paper.

Hibino proposed a special processor for a parallel GC system, which consists of two independent processes sharing a common database: the list process and the garbage collector, which cooperate with each other to perform garbage collection. Hibino's algorithm is designed on the condition that all active cells are bound in a linear stack: this condition is necessary for a practical list-processing system. Additionally, it needs one mark bit per cell and has a scan-request-flag (SREQ) for the system (the SREQ is used by the garbage collector for deciding whether to go into the reclaiming phase or to again perform the mark propagating phase) [Ref. 16: p. 113].

The SREQ informs the garbage collector of list modification occurrences, thus requiring only one mark bit for each cell. It applies a recursive trace and track method for the marking phase (Dijkstra's algorithm applies a simple scan and mark method) [Ref. 15: p. 6].

Hibino's special processor, which performs concurrent garbage collection, was actually implemented using two TOSBAC-40L processors (whose architecture is similar to the INTERDATA 6/16). The processor cycle time is less than 200 nanoseconds. The processor is built using standard TTL circuit technology. The microinstructions for this processor are specifically tailored for garbage collection [Ref. 16: p. 119].

Hibino's method requires very little overhead due to parallelism [Ref. 15: p. 8].

The next section provides an insight into future possibilities in the area of garbage collection.

C. FUTURE POSSIBILITIES

Thus far the only application of parallel garbage collection has involved one or two processors. With the advent of VLSI technology, having multiple processors on a single chip is not only possible by potentially cost-effective. Lamport first explored the possibility of concurrent garbage collection in a <u>multiprocessor</u> environment. There are numerous other possible multiprocessor configurations, which can involve any number and any combination of collectors and list processors. Two simple examples are:

(1) A system with only one garbage collection processor and having a finite number of list processors.

(2) A system with several garbage collection processors and at least one list processor.

The former example could be implemented by using a variaion of Steele's method. Steele's synchronization interlocks would require redesign but may not be as complex as in his original method (e.g. require one primitive to lock out all the list processors, except the one list processor that is manipulating the free list) [Ref. 10: p. 506].

The latter example could be implemented by having all the garbage collectors in the same phase at the same time and then divide the work in each phase among them; or it could be implemented by assigning specific "spaces" (as in Steele's method) semi-permanently to the processors and allowing each processor to collect garbage <u>asynchroneously</u> [Ref. 10: p. 506]. One implementation of this method (multiple garbage collectors) could be to have all the garbage collectors active simultaneously. The garbage collector would still have four basic phases (only three if <u>compaction</u> not desired):

(1) Set-up
All nodes are marked as garbage by setting a mark bit (or bits). This is similar to Dijkstra's coloring (white cells). This phase is typically executed as a by-product of the collection phase.

(2) Marking
All nodes accessible from the roots of the list structure are marked as accessible. This is similar to Dijkstra's coloring black. The marking phase may have several stages in which the mark stage (color) of the node changes. This will necessitate having more than one marked bit.

(3) Collection
All unmarked (white) nodes are added to the free list.

(4) Compaction. [Ref. 20: p. 367]

While concurrent garbage collection, using either hardware or software techniques, is being pursued as a desired feature in list-processing systems, future possibilities in the variations of other garbage collection methods are also being studied.

For example, an extension of the previously discussed concurrent GC method (with multiple processors) could be _dynamic allocation_ of several processors to list-processing or garbage collection (i.e. perform GC or list-processing as necessary). Clever heuristics would be needed to describe whether switching a processor at time "t" is desirable to forestall having to wait on an empty free list at time "t + n" [Ref. 19: p. 50].

Another example is in the case of one processor being _time-shared_ between list-processing and GC; a special micro-coded processor could be designed to switch to garbage collection ONLY on completion of a list-processing primitive, which would eliminate many of Steele's synchronization interlocks [Ref. 10: p. 506].

In the last example, it is known that garbage collection with one processor typically consumes a substantial percentage of the total computation time used by list-processing systems. Shifting storage reclamation costs, away from a program's run time overhead, to compile time would reduce the system cost of operating list-processing systems. This can be accomplished using a Deutsch-Bobrow scheme (which uses a combination of garbage collection and reference count methods). This scheme maintains reference counts in a way that can be expected to require less space than usual. It has the property that the counts need to be updated far less often than by traditional methods. Moreover their method is incremental: consequently, unlike regular garbage collection, it is not disruptive of real-time computation [Ref. 22: p. 514].

Automatic storage reclamation, when viewed in the Deutsch-Bobrow model, can benefit from compile time optimization [Ref. 22: p. 518].

These last three examples were only provided to illustrate that studies in the variations of garbage collection methods for list-processing systems are still receiving attention and will continue to do so as the popularity of list-processing systems grows.

D. CONCLUSION

With the advent of VLSI technology, many limitations on restricting the number of processors (collectors or list processors) because of cost or feasibility, are eliminated. Of course, more processors in a system means more coordination and control among the processors is necessary. This results in an increase in system complexity and in the synchronization algorithms. If implemented in software, this will mean an increase in the cost of maintaining these software algorithms.

This chapter has shown that there exists a wide range of algorithms and techniques for concurrent garbage collection. VLSI technology has pushed the continuation of future studies in this area, as has the popularity of list-processing systems.

The last chapter of this paper provides conclusions about the feasibility of conducting concurrent garbage collection.

VII. CONCLUSION

Storage reclamation in most list-processing systems, including LISP, is a necessity. Most LISP implementations utilize garbage collection to reclaim storage (memory occupied by unused cells) as the data structures of a program grow and shrink. LISP is a highly interactive list-processing system, that is useful in many applications and is growing in popularity. The flexibility and expressibility of LISP have made it the "work-horse" of the AI community [Ref. 12: p. 112].

The use of garbage collection for reclaiming unused storage should always be considered when implementing a list-processing system [Ref. 4: p. 506]. Is it better to utilize regular GC or concurrent GC in a list-processing system? This is a design decision that must be made before implementation. Both garbage collection methods (regular and concurrent) have advantages and disadvantages.

Regular garbage collection has the problem of unpredictable suspension of users' programs while garbage collection is being conducted. This occurs when the free list has been exhausted. As previously illustrated (and depending on the size of memory), regular GC can cause serious list-processing delays, which can occur any time the program needs a new cell [Ref. 12: p. 112].

Concurrent garbage collection eliminates these list-processing delays, which are caused by insufficient memory being available. Concurrent GC also allows program execution to continue without interruption even during periods when garbage collection is occurring. The difficulties in parallel garbage collection are caused by the fact that the list processor is modifying the list data (i.e. creating new cells or changing cell pointers) while the garbage collector determines which cells are garbage (non-accessible) [Ref. 15: p. 1]. Which implementation is better? In the comparison between regular GC and concurrent GC, the productivity of a parallel GC system can be as much as 150% that of regular garbage collection [Ref. 3: p. 1151].

The idea of performing concurrent garbage collection has been around for some time. Knuth credits this idea to M. Minsky. Though concurrent garbage collection is an appealing idea for real-time processing applications, no papers were published until about 1975. Both Steele and Dijkstra were among the first to investigate such a system [Ref. 21: p. 127]. Steele in 1975 presented the first concrete algorithm for the problem of garbage collection, which involved a high degree of interleaving of the processors' actions [Ref. 15: p. 1].

Garbage collection can be implemented with <u>compaction</u> or without compaction. As previously noted, GC with compaction has been shown to have significant time gains in terms of performance in LISP programs. Garbage collection without compaction is typically performed in a two phase process of first tracing and marking all active cells and then sweeping all unmarked cells back to the free list. Garbage collection with compaction prevents "thrashing." While compaction is not a necessity even in a systems that utilize single-size cells, compaction in list-processing systems is recommended.

> Garbage collection with compaction may be the only way to prevent intolerably slow processing of lists having elements which are widely scattered in memory. In these cases, automatic GC could be triggered when the number of transfers from slow memory, per unit of time, becomes larger than a specified maximum [Ref. 14: p. 26].

This paper has attempted to show that concurrent storage reclamation in the form of garbage collection, with compaction and without compaction, is feasible today using software or hardware techniques; and if implemented will provide better services to the system users (i.e. less program execution interruptions, increased speed-up of list-processing, etc.) than regular garbage collection can provide.

Many possible methods of concurrent garbage collection systems have been presented, and study of the behavior and design (including implementation) of <u>concurrent garbage collection</u> systems is an increasing field of investigation as list-processing systems becomre more commonplace. These studies are occurring because regular garbage collection is unsatisfactory in many situations. For example, regular garbage collection delays may cause more serious problems than frustration and discomfort among interactive users. For example, in list-processing programs designed to control physical devices, such as robotics applications, suspending processing operations for the time garbage collection normally requires may be intolerable: consider the plight of a future tennis playing robot forced to halt in mid-swing while it performs garbage collection [Ref. 5: p. 491].

While this is a trivial example, imagine the consequences of a regular GC list-processing system, which is designed to provide time-critical decisions, such as the system whose design and purpose is centered around "space defense." Now imagine if program execution had to halt for garbage collection at a most crucial time (e.g. unknown air targets approaching Washington, D.C. from the NORTH at a high rate of speed): this would not only be an unsatisfactory and unreliable system but the results could prove to have unpredictable, disasterous, and extremely sensitive consequences.

If concurrent GC is being used at its maximum capacity, it requires twice as much processing power as regular GC. A system with two processors (one dedicated to list-processing and the other dedicated to garbage collection) will provide an increase in the execution speed and possibly a decrease in the total amount of memory required as well [Ref. 5: p. 500]. With the decreasing costs of processors, the drawback of requiring twice as much processing power is offset by the advantage of avoiding garbage collection interruptions.

Wadler developed an idea that compares regular garbage collection and concurrent garbage collection using a term called the power drain of the collector, which is the amount of time that is used by the collector divided by the amount of time required by the mutator in a single cycle. Since list-processing systems typically spend 10-30% of their time in garbage collection, the power drain of regular GC is typically 11-43%. For concurrent GC systems, this means that the full capacity of a parallel collector will not normally be used [Ref. 5: p. 498].

> The ratio of the power drain between parallel and regular garbage collection can reach infinity. For example, in the situation where NO cells are used or released (i.e. the collector is never invoked), the "power drain" of a regular drain" of a regular GC system is zero. However, in parallel garbage collection, the power drain is one because the collector is still being kept busy even though it does not return any cells. Wadler has shown that when two processors operate at maximum capacity, the power drain is 2. This means that parallel GC requires twice as much processing power as regular GC," [Ref. 2: p. 358]

The power drain ratio between concurrent GC and regular GC will have its minimum value whenever the concurrent garbage collector is being used to its maximum capacity [Ref. 5: p. 498].

A. ASSUMPTIONS

While this paper has shown the feasibility of concurrent garbage collection, it must be remembered that several assumptions were made in order to reach this conclusion.

Some of the major assumptions in this paper are:

(1) Economics is not considered an issue: the cost of adding additional hardware (i.e. processors) is considered negligible, as is the assumption of having large memories. Memory space is not at a premium: large memory systems are becoming more commonplace.

(2) Processing time in interactive systems is not taken into consideration. In other words, the only requirement emphasized in program processing is that program execution will not halt due to GC activities. The length of time for list-processing is not considered.

(3) Performance issues and proof of correctness are not discussed.

(4) Efficiency issues (e.g. productivity, system cost effectiveness, throughput, and system viability), important considerations in any system, are also not discussed. This includes the problem that most garbage collection algorithms seem to have in common: they spend a lot of time "coloring" or marking garbage. This can be inefficient if the application contains large data structures that are modified only occasionally [Ref. 17: p. 340].

(5) Memory allocation, while related to memory deallocation (i.e. storage reclamation), is only briefly mentioned.

(6) All cells are of the same size. Cells of various sizes, which present a different problem and solution, are not discussed.

(7) Future hardware costs will continue to fall; thus making it advantageous to pursue concurrent garbage collection with at least two independent processors.

(8) The operation of the list processor is a typical existing processor; consequently, its operation was not studied.

(9) Garbage collection in virtual memory systems was only briefly discussed. Garbage collection in a large VM system might be bypassed altogether, although this would degrade system performance by scattering the active cells sparsely throughout memory; thereby requiring frequent paging [Ref. 3: p. 1153].

(10) A basic understanding of list-processing languages, in particular LISP, is required to grasp the discussion of the various storage reclamation methods.

(11) Overhead for the list processor is ignored: the definition of overhead (in parallel garbage collection) is whenever the mutator performs any list data modification, it must then perform some additional operations to inform the garbage collector [Ref. 15: p. 2]. For example, consider how frequently list modification occurs in list processing systems. If this occurred extremely often, it would be impossible to introduce parallelism into garbage collection. Fortunately in the case of LISP, this list data modification occurs explicitly only when cons and replace operations are performed [Ref. 15: p. 2].

While it is not the purpose nor intention of this paper to prove any of the above assumptions, it still must be noted that many of the above assumptions are critical to the design, implementation, and maintenance of a list-processing system that uses garbage collection (in particular concurrent garbage collection). The above assumptions should first be considered separately and then jointly when deciding on a particular scheme to take care of "garbage cells" in a list-processing system.

B. FINAL REMARKS

The simplest example of a parallel garbage collection system consists of two independent processes (the list processor and the garbage collector) that share a common memory. Both processors must "cooperate" with each other to allow garbage collection.

Successful implementation of list-processing systems, with parallel garbage collection, gives a foothold for parallel list evaluation [Ref. 15: p. 8].

The problem of parallel GC, which involves a high degree of interleaving of the processors' actions, is one of the most challenging problems in parallel programming [Ref. 16: p. 113]. The latest trend in the computing industry shows that software costs are increasing annually. This is due largely to increased maintenance costs of this software and the associated personnel costs related to developing and maintaining this software. On the other hand, hardware costs have been declining, and with the advent of VLSI technology have made it possible to have numerous processors on a single chip at a reasonable cost. Consequently, concurrent garbage collection using software is not considered the optimum solution. Addition of a second processor for garbage collection requires very little overhead [Ref. 19: p. 50].

As prices of processors continue to fall and as micro-coded processors become more common, the design of these special processors becomes more feasible. If this is done, then the list processor can operate unimpeded; since processor interlocking is necessary, it could be achieved through special hardware in such a way that the list processor almost never has to wait on the garbage collector and then only for very brief periods of time [Ref. 10: p. 506].

Synchronization, which must be done when the mutator attempts to remove a node from an empty free list, can be accomplished if cell removal and cell appending operations are conducted at opposite ends of the free list. An empty free list should happen infrequently and any convenient synchronization primitive can be used to prevent this [Ref. 17: p. 335].

It is the conclusion of this paper that concurrent garbage collection is not only possible, but that it has already been implemented. The simplest (and least costly) method of parallel garbage collection with list-processing is to use one processor strictly for list-processing operations and another processor strictly for garbage collection tasks. Parallel garbage collection with more than two independent processors (easily attainable with VLSI technology at a reasonable cost) is possible but is not considered <u>optimum</u> because of algorithm complexity and increased synchronization problems.

Two independent processors designed with separate tasks (list-processing and garbage collection) will mean that the list processor will not have to wait while garbage collection is being conducted as in regular garbage collection systems. There are some immediate advantages to concurrent GC:

(1) A net speedup in list processing operations.

(1) An increase in the efficiency of memory usage [Ref. 5: p. 492].

Consequently, it is recommended that concurrent garbage collection (with compaction) be implemented in a list-processing system with hardware techniques. The least costly and simplest method is to have one processor dedicated to list-processing and another processor dedicated to GC.

LIST OF REFERENCES

1. Deutsch, L.P. and Bobrow, D.G., "An Efficient, Incremental, Garbage Collector", Communications of the ACM, v. 19 # 9, pp. 522-526, September 1976.

2. Cohen, J., "Garbage Collection of Linked Data Structures", ACM Computing Surveys, v. 13 #3, pp. 341-367, September 1981.

3. Cohen, J. and Hickey, T., "Performance Analysis of On-the-Fly Garbage Collection", Communications of the ACM, v. 27 #11, pp. 1143-1154, November 1984.

4. Schorr, H. and Waite, W.M., "An Efficient Machine-Independent Procedures for Garbage Collection in Various List Structures", Communications of the ACM, v. 10 #8, pp. 501-506, August 1967.

5. Wadler, P.L., "Analysis of an Algorithm for Real Time Garbage Collection", Communications of the ACM, v. 19, #9, pp. 491-500, September 1976.

6. Sandewall, E., "Programming in an Interactive Environment: the LISP Experience", Computing Surveys, v. 10 #1, pp. 35-71, March 1978.

7. Knuth, D.E., The Art of Computer Programming, 2nd Ed., v. 1, pp. 406-420, Addison-Wesley, 1973.

8. Cohen, J. and Nicolau, A., "Comparison of Compacting Algorithms for Garbage Collection", ACM Transactions on Programming Languages and Systems, v. 5 #4, pp. 532-553, October 1983.

9. Bates, W., The Computer Cookbook, pp. 139-140, Garden City, 1984.

10. Steele, G.L. Jr., "Multiprocessing Compactifying Garbage Collection", Communications of the ACM, v. 18 #9, pp. 495-508, September 1975.

11. MacLennan, B.J., Principles of Programming Languages: Design, Evaluation and Implementation, pp. 439-449, CBS College Publishing, 1983.

12. Bonar, J.G. and Levitan, S.P., "Real Time LISP Using Content Addressing Memory", 1981 Conference on Parallel Processing, pp. 112-119, 1981.

13. Wegbreit, B., "A Generalized Compactifying Garbage Collector", Computer Journal, v. 15 #5, pp. 204-208, August 1972.

14. Cohen, J.A. and Trilling, L., "Remarks on Garbage Collection Using a Two Level Storage", Bit, v. 7 #1, pp. 22-30, 1967.

15. Hibino, Y., "A Parallel Garbage Collection and its Application in LISP", The Transactions of the IECE of Japan, v. E63 #1, pp. 1-8, January 1980.

16. Hibino, Y., "A Practical Parallel Garbage Collection Algorithm and its Implementation", Computer Architecture 1980, pp. 113-120, 1980.

17. Ben-Ari, M., "Algorithms for On-the-Fly Garbage Collection", ACM Transactions on Programming Languages and Systems, v. 6 #3, pp. 333-344, July 1984.

18. Dijkstra, E.W., Lamport, L., Martin, A.J., and Scholten, C.S., "On-the-Fly Garbage Collection: An Exercise in Cooperation", Communications of the ACM, v. 21 #11, pp. 966-975, November 1978.

19. Lamport, L., "Garbage Collection with Multiple Processing: An Exercise in Parallelism", Proc. IEEE Conf. Parallel Processing, pp. 50-54, August 1976.

20. Newman, I.A., Stallard, R.W., and Woodward, M.C., "Improved Multiprocessor Garbage Collection Algorithms", 1983 Parallel Processing Conf., pp. 367-368, 1983.

21. Kung, H.T. and Song, S.W., "An Efficient Parallel Garbage Collection and its Correctness Proof", 1977 18th Annual Symp. on Foundations of Comp. Sci., pp. 120-131, November 1977.

22. Barth, J.M., "Shifting Garbage Collection Overhead to Compile Time", Communications of the ACM, v. 20 #7, pp. 513-518, July 1977.

BIBLIOGRAPHY

Arnborg, S., "Optimal Memory Management in a System with Garbage Collection", Bit, v. 14 # 4, 1974.

Baecker, H.D., "Garbage Collection for Virtual Memory Computer Systems", Communications of the ACM, v. 15, # 11, November, 1972.

Beck, L.L., "A Dynamic Storage Allocation Technique Based on Memory Residence Time", Communications of the ACM, v. 25, # 10, October, 1982.

Berry, D.M. and Sorkin, A., "Time Required for Garbage Collection in Retention Block-Structured Languages", Int. J. Comput. Information Sci., v. 7, # 4, 1978.

Caluwaerts, L.J., Debacker, J., and Peperstrate, J.A., "A Data Flow Architecture With a Paged Memory System", 1982 Computer Architecture, 1982.

Campbell, J.A., "Optimal Use of Storage in a Simple Model of Garbage Collection", Information Processing Letters, v. 3 # 2, November, 1974.

Christopher, T.W., "Reference Count Garbage Collection", Software-Practice and Experience, v. 14 # 6, June, 1984.

Dewar, B.K., Sharir, M., and Weixelbaum, E., "Transformational Derivation of a Garbage Collection Algorithm", ACM Transactions on Programming Languages and Systems, v. 4 # 4, October, 1982.

Fenichel, R.R. and Yochelson, J., "A LISP Garbage Collector for Virtual Memory Computer Systems", Communications of the ACM, v 12 # 11, November, 1969.

Fitch, J.P. and Norman, A.C., "A Note on Compacting Garbage Collection", Computer Journal, v. 21 # 1, February, 1978.

Hansen, W.J., "Compact List Representation: Definition, Garbage Collection and System Implementation", Communications of the ACM, v. 12 # 9, September, 1969.

Hoare, C.A.R., "Optimization of Store Size for Garbage Collection", Information Processing Letters, v. 2 #6, April, 1974.

Jonkers, H.B.M., "A Fast Garbage Collection Algorithm", Information Processing Letters, v. 9 #1, July, 1979.

Kain, R.Y., "Block Structures, Indirect Addressing, and Garbage Collection", Communications of the ACM, v. 12 #7, July, 1969.

Kucera, L., "Parallel Computation and Conflicts in Memory Access", Information Processing Letters, v. 14 #2, April, 1982.

Larson, R.G., "Minimizing Garbage Collection as a Function of Region Size", Siam J. Computing, v. 6 #4, December 1977.

McCarthy, J., "Recursive Functions of Symbolic Expressions and the Computations by Machine, Part I", Communications of the ACM, v. 3 #4, April, 1960.

Martin, J.J., "An Efficient Garbage Compaction Algorithm", Communications of the ACM, v. 25 #8, August, 1982.

Morris, F.L., "A Time and Space Efficient Garbage Collection Algorithm", Communications of the ACM, v. 21 #8, August, 1978.

Morris, F.L., "A Comparison of Garbage Collection Techniques", Communications of the ACM, v. 22 #10, October, 1979.

Morris, F.L., "Another Compacting Garbage Collector", Information Processing Letters, v. 15 #4, October, 1982.

Owicki, S., "Making the World Safe for Garbage Collection", ACM Symposium on Principles of Programming Languages, January, 1981.

Ramesh, S. and Mehndiratta, S.L., "The Liveness Property of On-the-Fly Garbage Collector-A Proof", Information Processing Letters, v. 17 #4, November, 1983.

Terashima, M. and Goto, E., "Genetic Order and Compactifying Garbage Collectors", Information Processing Letters, v. 7 #1, January, 1978.

Thorelli, Lars-Erik, "A Fast Compactifying Garbage Collector", Bit, v. 16 #4, 1976.

Vermeulan, F.L., "On the Combined Problem of Compaction and Sorting", IEEE Transactions on Software Engineering, v. 8 #4, July, 1982.

Weizenbaum, J., "Knotted List Structures", Communications of the ACM, v. 5 #3, March, 1982.

Wise, D.S., "Morris Garbage Compaction Algorithm Restores Reference Counts", ACM Transactions on Programming Languages and Systems, v. 1 #1, July, 1979.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Va 22304-6145	2
2.	Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	1
3.	Dr. Bruce J. MacLennan Code 52M1 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	1
4.	CDR Ronald E. Rautenberg, USNR Code 52Rt Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	1
5.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
6.	Computer Technology Programs Code 37 Naval Postgraduate School Monterey, California 93943-5100	1
7.	CDR J.R.L. Cassidy, USN (Ret.) 7409 Forrester Lane Manassas, Virginia 22110	1
8.	LCDR K. G. Cassidy, USN 3801 Forrester Lane Virginia Beach, Virginia 23452	3

602-197

216181

Thesis
C283 Cassidy
c.1 The feasibility of automatic storage reclamation with concurrent program execution in a LISP environment.