Design of a List-structure Memory using Parallel Garbage Collection

Mark H. Foster
B.A., Willamette University, 1983

A thesis submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science & Engineering

September, 1985

The thesis "Design of a List-structure Memory using Parallel Garbage Collection" by Mark H. Foster has been examined and approved by the following Examination Committee:

Richard B. Kieburtz, Thesis Research Advisor
Professor and Chairman,
Department of Computer Science and Engineering

David Maier
Associate Professor,
Department of Computer Science and Engineering

Shreekant S. Thakkar
Assistant Professor,
Department of Computer Science and Engineering

Will Clinger
Tektronix, Inc.

Acknowledgements

I would like to express my gratitude to Dr. Richard B. Kieburtz for his guidance and encouragement during my study at the Oregon Graduate Center. It was the formal insight and expertise of Dr. Kieburtz that permitted the extension of Ashoke Deb's algorithm, and thus, the foundation of this project. He has been both a teacher and a friend. Thanks are due Dr. Ashoke Deb, for without his algorithm development and his advice, this project could not have commenced.

I wish to thank Dr. Shreekant Thakkar for his support and motivation during the early stages of this project. Much gratitude is due to my thesis committee members for their efforts, and especially to Dr. David Maier, an excellent teacher.

I gratefully acknowledge the Computer Research Lab, at Tektronix, for the design and provision of the Magnolia workstation, on which most of the project was initially developed. Thanks are due to Bruce Jerrick, for providing skillful technical advice and for maintaining the development environment. I want to thank my friend Richard Vireday for his support and assistance throughout my work at OGC.

Table of Contents

List of Figures

Abstract

Design of a List-structure Memory using
Parallel Garbage Collection

Mark H. Foster, M.S.
Oregon Graduate Center, 1985

Supervising Professor: Richard B. Kieburtz

Dynamic, list-structure memory provides direct hardware support for list-processing systems; it facilitates concurrent evaluation and memory management that requires little explicit cooperation between the evaluator and memory management processors.

This thesis reviews the suitability of present concurrent garbage-collection schemes, and introduces a new scheme that provides incremental, parallel collection. A memory architecture to support the new algorithm is also described. A multi-process simulation of the architecture and the new algorithm show that this model will support intensive list-structure operations of applicative language evaluators under development.

1. Introduction.

Dynamic, list-structure memory is an architectural primitive of all abstract machines that evaluate list-processing languages. Since the advent of list processing systems, there has been demand for memory organizations specific to lists. Evaluators for list processing systems make use of memory in a manner that emulates dynamic, object-oriented storage. This technique can be observed in a variety of systems. Lisp |McC60], Prolog |WaP77|, and Smalltalk-80™ [GoR83, KaK83| all use list- or object-structure memory for the basic store model. Typical list-structure memory consists of individually accessible nodes, each capable of storing two or more data, which may be basic values or the addresses of other nodes. During the evaluation of a list-processing program, new nodes are automatically supplied by the memory system, while nodes no longer used are simply ignored.

Thus, as the evaluator creates, modifies, and discards lists of nodes, it both consumes new nodes and produces garbage nodes. These so-called garbage nodes are the memory components that are no longer accessed or used by the evaluator. A simple example illustrates the creation of a garbage node.

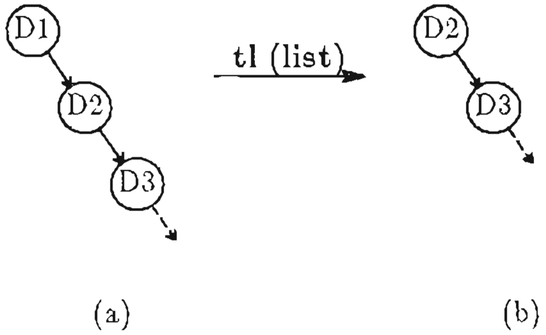


Figure 1.1. Example of garbage creation.

Smalltalk-80 is a trademark of Xerox Corporation

Given a list of data elements (Figure 1.1a), a typical operation on the list is to remove the "head" of the list, leaving the rest of the list intact (Figure 1.1b). Assuming a single reference to the list, once the head of the list (D1) is removed, the storage for the head is no longer needed, and can be re-used for other data elements. At this point, the storage for D1 is considered "garbage".

The detection and reclamation of unused store are the primary functions of list-structure memory management. Once unused store is reclaimed, it can subsequently be re-used. Several fundamental approaches for handling unused store have emerged. We outline them here, and discuss the nature of the techniques in more detail later.

manual management: This method is programmer enforced. Some still consider it useful for special-purpose applications, but it tends to be error prone and is not adequate for most applications. We will not return to this approach.

reference counting: This technique tallies the number of references to a cell, and determines immediately when the cell is no longer referenced. It cannot collect cyclic structures, however.

copying, compacting: Memory space is divided into two or more workspaces, where some of the workspaces are active and some are free. Periodically, cells are copied from an active space into a free space; the active space is freed, and the free space is considered active. Fragmentation is reduced during the copying process.

mark-sweep: Cells in the active store are marked, then the entire store is swept. This two-cycle operation determines which cells can be accessed, and collects the rest.

specific adaptations: These involve noting behavior of a particular computing environment and tailoring the memory management to specific circumstances.

All of these techniques suffer from one or more significant drawbacks when extended for parallel collection. As demand for large scale list processing and efficient functional program evaluators increases, these drawbacks become major performance issues in a functional language environment. By adapting a new, incremental garbage collection scheme for execution in a parallel collection system, we establish an approach that avoids these problems and shows promise for high-performance, multi-processor, functional-language systems.

1.1. Outline of remainder.

In the following section, we overview the main features and drawbacks of established techniques outlined above, thereby setting the stage for Section 3. Section 3 consists of a presentation of a new collection algorithm and an extension of the basic algorithm for concurrent execution. Section 4 describes an application of the concurrent algorithm and desirable changes for a graph-reduction machine. The architecture for a relatively general-purpose dual-ported list-structure memory is provided in Section 5. The simulation of the presented architecture, in terms of the graph-reduction application, is detailed in Section 6. Section 7 reports the results of the simulation. Conclusions and suggestions for further work are given in Section 8.

2. History -- known and used techniques.

Several techniques for managing list-structure memory have been developed. Many popular techniques are termed *garbage collection*, since unused storage is allowed to accumulate, becoming garbage, before it is returned to the available store pool. Garbage collection has received considerable attention in the development of many large list processing systems [Moo74, Tei78]. Another approach, based on *reference counting*, immediately deallocates storage once it is no longer needed. Most of the current Smalltalk-80 systems use some form of reference counting [KaK83]. In this section, we discuss the general[1] features of storage management techniques that have been developed for list-structured memories and explain why we feel none is completely satisfactory.

2.1. Mark-Sweep.

Two modes of mark-sweep garbage collection have been established. The stop-and-collect approach allows the evaluation to proceed until there are no nodes available in the free-node pool. The parallel approach uses two processors: one for the evaluation and one for concurrent garbage collection.

For the stop-and-collect mark-sweep technique, the evaluation is suspended once the free pool is exhausted. Subsequently, the garbage collector is activated and reclaims all unused, but not free, storage. The basic algorithm consists of two phases:

1) Mark. Determine which nodes are still accessible (in the active graph).

2) Sweep. Create the free pool from nodes that are not accessible.

[1] A rather thorough survey of classical and relatively recent storage management techniques is presented by Cohen [Coh81].

Once the collector completes the sweep phase, evaluation restarts. The primary disadvantage is that list processing must be suspended periodically, since the same processor is used for both evaluation and for collection. Both the frequency and the length of the suspension can be arbitrary.

In an attempt to eliminate the inherent wait cycle, a parallel, multi-processor, algorithm can be developed from the stop-and-collect approach. This algorithm reduces the in-line overhead of a stop-and-collect implementation, and reduces the amount of time an evaluation must be suspended. Algorithms for multi-processor collectors have been proposed and proven correct [DLM78, Ste75]. The extension requires particular exclusion and synchronization constraints between the evaluator processor and the collector processor, and is defined by the combination of

(a) Low-level, hardware-supported synchronization of access to physical memory.

(b) Producer-consumer synchronization for allocation requests made by the evaluation process.

(c) The state of an evaluation process is periodically passed to the collector to reveal the current roots of the active graph.

Another disadvantage is realized when the memory occupancy approaches 100 percent. In this situation, the collector is forced to make nearly two full passes through the store in order to collect a small number of garbage nodes. Considerable time may be expended just to discover there is not enough memory available. Furthermore, the cost of the sweep phase increases as the total store increases. The problem becomes acute as very large memory spaces become necessary.

Hickey and Cohen have uncovered some specific limitations in a recent analysis of on-the-fly garbage collection [HiC84]. First, the freelist size is cyclical. This is because many nodes may become garbage during a given cycle, but will have to wait until the following cycle to be collected; the end result is that temporary storage is the most expensive to collect! Second, this approach yields a maximum achievable speedup of only about 1.5 over the sequential approach.

2.2. Copying, Compaction.

An alternative to mark-sweep collection is *scavenging* [Bak78, LiH83, Ung84]. That is, memory is partitioned into two or more spaces. New nodes are allocated from an active space until the space is full. A breadth-first traversal of the graph in an active workspace copies reachable nodes into an available (free) workspace, coalescing the active nodes into a contiguous block. After a scavenge, the former workspace is freed. This approach avoids the costly sweep phase of mark-sweep, and periodically eliminates fragmentation. Copying-compaction also has the advantage that allocations of new nodes can be made in a very cheap fashion by simply incrementing an address pointer. However, only half the total address space is available at a time: this effectively "wastes" 50 percent of usable store[1].

As with mark-sweep collection, copying-compaction degrades as the memory occupancy approaches capacity. This degradation is due to increased frequency of copying nearly the entire active store to reclaim (perhaps) just a few nodes.

In terms of store accesses, the copying process incurs about twice the expense of the marking phase of mark-sweep collection, and it is less amenable to parallel execution [Kie85a]. Node-level exclusion and synchronization is required while a node is being copied. Furthermore, indirection overhead will be introduced whenever the evaluation process accesses a node that has already been copied. This mechanism

[1] Admittedly, if virtual memory is used, this is not a problem

requires a "fault" whenever an indirection pointer is encountered, and a retry on the node that is pointed to.

2.3. Reference Counting.

The reference counting technique makes use of an additional field in each node or equivalent space in an auxiliary table. A count of the number of inward arcs (references to the node) is stored in this field. Operations on a pointer may either increment the reference count when a pointer is copied, or decrement the reference count when a pointer is overwritten or released. Whenever the reference count reaches zero for a node, the node is returned to the pool of available nodes. This technique benefits from the simplicity of implementation and the immediate availability of a node for reuse, once it is no longer used. There is also no reduction in efficiency of memory management as the size of the active graph gets large with respect to the available store. However, reference counting suffers from several disadvantages:

- Cyclic graph structures are never reclaimed.
- Additional bits in each node must be devoted to memory management. The number of bits is generally quite a few more that those needed for a marking or copying scheme; plus, the count can overflow when the number of bits is fixed.
- The average cost of a memory reference is increased by the in-line overhead of maintaining reference counts.

Modified reference counting schemes have addressed some of these burdens. In particular, an approach developed by Ashoke Deb uses reference counts to trigger traversal of (possibly) cyclic structures [Deb84]. When the overhead of maintaining reference counts is made negligible, there may be far more efficiency in this approach than the marking or copying schemes. This topic will be expanded upon shortly.

2.4. Adaptations.

Many recently developed list-processing systems have employed specific adaptations of general memory management schemes. These schemes typically are designed around particular attributes of the evaluation system being used. Some examples include:

Automatic memory reclamation, based on scavenging, was first introduced by Baker. Baker's semispace algorithm [Bak78] divides memory into two partitions and scavenges all reachable objects from one partition to the other. Lieberman and Hewitt [LiH83] generalize Baker's approach by separating objects according to age; young objects often become trash quickly while older objects are less likely to be discarded.

A hybrid of reference counting and garbage collection is given by Deutsch and Bobrow [DeB76]. Their notion is to primarily use a reference-counting approach. They avoid in-line overhead by delaying the determination of the reference count until just prior to collection. In addition, they rely on an eventual stop-and-collect for self-referent structures.

Each of these techniques has been shown to improve performance in the cases they were designed to handle; the difficulty is that they are ad-hoc to some degree. In particular, none provides a general scheme for memory management in a functional language evaluator. There is also little provision for reducing in-line overhead through parallel processing.

2.5. Motivation for a new incremental approach.

Particular demands are placed on memory management for graph reduction evaluators. Since expressions are created and represented by (possibly cyclic) graphs *during execution*, there is need to provide a ready supply of fresh nodes without impeding the evaluation process. In typical graph evaluation [JoA83, Kie85c, Sar84], there are fewer cyclic graphs than acyclic graphs. Acyclic structures can be very effectively collected using a reference counting scheme. With special components (e.g., Wis85) to aid the in-line overhead of maintaining reference counts, this suggests a highly efficient collection strategy for graph evaluators may be developed by augmenting reference counting with a graph-traversal scheme. While this approach is of particular benefit to graph evaluation, it also provides a model from which a general list-structure memory can be developed.

3. A new collection algorithm.

In this section, we review an incremental collection algorithm devised by Deb. We then discuss the adaptation of this algorithm that permits execution in a multi-process environment and provide an informal proof of the adaptation.

Deb's algorithm is attractive for a variety of reasons.

* The approach is incremental. It is important to distinguish our notion of *incremental* from time-slice techniques [DeB76]. Previous incremental collection approaches regularly use a small piece of the processing resources (e.g., time-slicing) to perform part of a collection pass. Our approach is considerably different, in that the average *size* of the passes is reduced, then each is run *in its entirety*. The effect is that demand for collection is proportional to the amount of garbage created, and that reclamation of unused nodes is step-wise.

* The cost per collected node is constant. The incremental strategy yields a fixed average cost over all ratios of memory occupancy ρ. The costs of mark-sweep and copy-compaction grow as $\frac{1}{1-\rho}$ and $\frac{\rho}{1-\rho}$, respectively.

* The algorithm can be extended to allow parallel execution. By carefully applying limited synchronization constraints, and by adding an initialization traversal, the elements of the sequential approach can be used to establish a concurrent algorithm.

* Unlike many other memory management algorithms based on reference counting, cyclical structures are collectible.

Some particular drawbacks are inherited by the nature of reference counting.

* Reference counts require an extra field (i.e., extra bits). Moreover, reference counts of fixed precision can overflow. Depending on the interpretation of

reference count overflow, this may also have the bad influence of making a node uncollectible.

* There is in-line overhead for maintaining reference counts.

With special hardware, the latter problem can be avoided. A method and architecture for handling this problem will be discussed shortly. First, though, we present an interpretation of Deb's algorithm.

List-structure memory management.

In systems where explicit memory manipulation is allowed for the evaluator, a very general reclamation algorithm must be used. Since there is little control over how the evaluator uses memory, a specific protocol for the management of the memory cannot be established.

At the other end of the spectrum, an evaluator that forms only acyclic graphs could use reference counts effectively, since the major drawback of traditional reference counting techniques is the inability to collect cyclic structures.

A graph-reduction evaluator neither requires explicit manipulation of memory representations nor forms exclusively acyclic graphs. As indicated by Kieburtz [Kie85a], however, the incidence of cyclic graphs is much less than the incidence of acyclic graphs. In this situation Deb's algorithm might be applied effectively.

3.1. Deb's modified reference counting algorithm.

The environment for collection consists of a rooted, reference-counted graph, a process that traverses and possibly modifies the graph (mutator), and a process that finds nodes no longer accessible from the root of the graph (collector). Figure 3.1 shows the template of a graph node. A brief description of each field follows.

p0 indicates whether first data field is a pointer or simple data.

p1 indicates whether second data field is a pointer or simple data.

P persistent (private use by memory manager).

A allocated/not-allocated (private use by memory manager).

C is collectible (private use by memory manager).

U permanently uncollectible (private use by memory manager).

V mutator-visit bit is set whenever the mutator changes data.

loc_ref_ct local reference count (private use by memory manager).

ref_count reference count

thresh threshold defines a limit for pointers from a node's descendents.

data two 32-bit fields may hold pointer or simple data.

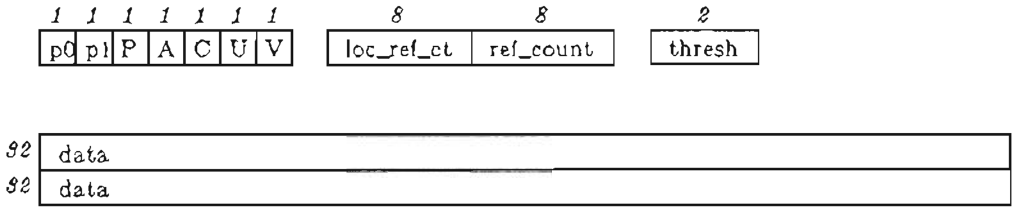


Figure 3.1. Template of a graph node.

If it seems like there are quite a few bits allocated for the memory management compared to bits used for data, consider this: In the past, use of available store was minimized since memory was expensive, relative to cpu cycles. In modern designs, though, available memory no longer needs to be a strict limitation. Not only have dense memories become rather inexpensive, but for custom VLSI designs, memory is the most regular circuitry, and therefore tends to be least expensive in terms of silicon real estate.

Deb's approach is based on the notion that only potentially collectible subgraphs are inspected. Whenever a pointer to a node is elided, the reference count of the node is decremented, and the address of the node is put into the "Garbage Can". At a convenient time, the mutator is stopped, and nodes pointed to from the Garbage Can are examined to determine if they are collectible.

Any node whose reference count is zero can be collected immediately. When a node is collected, the reference counts of the nodes it points to are decremented, and entries for those nodes are placed in the Garbage Can.

Any node **N** having a non-zero reference count could be collectible, if it were found that **N** is part of a cycle in which all the nodes reachable from **N** cannot be reached from a node which is not reachable from **N**. This situation gives rise to condition C[1].

(C) Given a node **R**, root of a strongly-connected subgraph, **R** is part of an unreferenced cycle if the reference count at each node in the subgraph is equal to the number of edges incident from nodes within the subgraph.

To test condition C, it is sufficient to make two traversals of a spanning tree through the graph rooted at **R**.

Traversal 1: count local arcs.

The purpose of traversal 1 is to account for all the references to a node **R** from nodes which are reachable from **R**. A field called the *local reference count*, is maintained for each node. At the start of the first traversal, assume that all local reference counts are zero. For each pointer encountered, the local reference of the node it points to is incremented. To handle cyclic structures, we must discern if a node

[1] Condition C integrates Deb's condition B [Deb84, page 7].

has been previously visited in the current traversal. If the node has not been visited, its local reference count is is zero, otherwise it is non-zero.

At the conclusion of the first traversal of the subgraph **R**, the local reference count for each node in the subgraph is exactly the number of edges incident from within the subgraph.

Traversal 2: collectibility.

The second traversal determines if the subgraph is collectible by comparing the actual reference counts with local reference counts. During this traversal, two states govern the collector's actions. We call these *optimistic* and *pessimistic*[1]. If a node is encountered when optimistic, and if the local reference count equals the actual reference count, the node is marked as potentially collectible and the collector continues to be optimistic. Otherwise, the collectible bit is cleared, and the collector is pessimistic while visiting all successors of the node. During this collectibility traversal, the local reference count for a node is reset to zero the first time the node is encountered. To ensure a cycle is only traversed once, the local reference count is checked whenever a node is visited; if the count is non-zero, the node has not yet been visited during this traversal, if the count is zero, the node must have already been visited.

At the end of the second traversal, the local reference count of each node is zero, and the collectible nodes will be exactly those that are both

i) marked as collectible, and

ii) reachable from **R** without traversing any node not marked as collectible.

[1] In Deb's nomenclature, optimistic is called *green* and pessimistic is called *red*

Traversal 3: collect.

The actual collection of nodes requires another traversal of the collectible subgraph rooted at R. This traversal deallocates nodes whose "collectible" bit is set. The traversal does not descend below nodes that are not marked collectible, and determines cycles by using the "allocated" bit. The "allocated" bit distinguishes between nodes that are free and nodes that are allocated. This bit is also convenient to allow duplicate entries in the Garbage Can without risking multiple collections of the same node. When a node is deallocated, its "allocated" bit is cleared.

On the surface, this approach may seem to imply considerable overhead, since two full traversals are required to determine collectibility, and a third is required to perform the actual collection. Closer observation shows, however, that this approach is actually quite reasonable. The primary factor improving its efficiency is that the collection strategy is *incremental*. Only those subgraphs rooted at a candidate node are inspected, rather than the entire allocated graph, as with mark-sweep or copying-compaction. Since candidates are defined as nodes having an incident edge deleted recently, these are the nodes (and likewise, subgraphs) most likely to be collectible. The average size of individual subgraphs does not grow in proportion to the total size of the active graph, given that the evaluator remains consistent. Also, the average cost per node collected is independent of the memory occupancy.

Deb's sequential algorithm can readily be extended to provide the constraints needed for a multi-processor environment. In this manner, a non-stop highly efficient memory management mechanism can be developed.

3.2. A concurrent approach.

The concurrent algorithm shares many traits of the stop-and-collect approach.

The step-wise, incremental features are retained. The fundamental components of each pass are the same. And simulation results, to be discussed later, indicate that the concurrent version also performs well. The most obvious difference is that the mutator does not have to be interrupted to perform collection. The benefits of the concurrent algorithm come at the cost of added complexity to both the mutator and the collector, plus an additional traversal through the candidate subgraph. Dijkstra notes that developing the constraints for parallel garbage collection is a somewhat tricky task [DLM78|. Deb's algorithm is no exception.

It is possible to develop scenarios in which the collection condition (C) appears to apply, yet there remains a reference to the root node **R** from outside the strongly-connected subgraph rooted at **R**. We say the condition "appears to apply" because the condition can only be tested with a sequential algorithm. The condition (C) is in fact the correct one; the difficulty, as indicated in [Kie85a], is formulating a reliable way to test it.

An obvious problem arises when the action of the mutator causes modification of the actual reference count of a node being inspected for collection. This problem is solved by adding a *mutator-visit* bit to the set of node tags. This bit indicates which process has most recently modified the node. The mutator-visit bit is only cleared and set according to the following conditions.

(mv.0) The mutator-visit bit is cleared only by the memory manager in the initial pass of the collection cycle.

(mv.1) The mutator-visit bit is set upon allocation of a node when its reference count is zeroed, and set whenever the reference count is either incremented or decremented.

If the mutator actions cause a 0→1 transition of the mutator-visit bit, the collector detects this, and collection of the node is invalidated for the current collection cycle. To detect the transition, the collector must test the bit during the *collectibility* pass.

A second problem is to provide a known state of the nodes being inspected for collection. During the *count local arcs* pass described above, the local reference count is assumed to start at zero. We could simply reset the count to zero at the conclusion of inspecting the graph for collectibility. However, this approach is not satisfactory. If in the course of executing the inspection algorithm, the actions of the mutator disconnect a portion **p** of the subgraph **R**, the nodes in **p** may no longer be reachable from the original root, leaving the memory manager unable to later reset the local reference count of the disconnected node(s). In a similar manner, **p** could be reattached to *another* part of **R** with non-zero local reference counts. These actions could lead the memory manager to a faulty calculation of the local reference counts, and thus, believe a subgraph to be collectible, when in fact, it is not.

To solve the second problem, the collection algorithm makes an additional traversal through the subgraph being inspected. This traversal we call *clear reference*. It precedes the passes listed above, becoming pass 0. In *clear reference*, the local reference count is zeroed, and the mutator-visit bit is cleared. The test for first visit to a node is that either one of these values is non-zero. Note that according to (mv.0) and (mv.1), once a node has been visited with the *clear reference* pass, it cannot be disconnected from the graph by mutator actions without causing the mutator-visit bit of the node or one of its ancestors to be set.

Proposition 1:† Following the initial pass, *clear reference*, either every node **a** in

† Both the theorem and proof are from the justification provided by Kieburtz in [Kie85a].

the graph rooted by **R** will have a local reference count of zero, or every path from **R** to **a** will contain at least one node whose mutator-visit bit is set.

Proof: Given a graph **G** rooted on **R**, and a node **b** reachable from **R**: suppose that, following the *clear reference* pass over **G**, the local reference count of **a**, a node in **G**, is non-zero. Since the local reference count for **a** can only be modified by the collector, it must be that the collector did not visit **a** during the pass. This situation means that **a** was added to the graph during the pass. In order to add **a**, either

1) an edge leading from a node **b** to **a** has been added, or

2) some edge leading from **b** to an ancestor of **a** has been added.

In the first case, the mutator-visit bit for **a** will be set, according to (mv.0). In the second case, the ancestor of **a** was not visited during the pass. The proof follows by induction on the depth of the unvisited node.□

The *count local arcs* and *collectibility* passes follow the *clear reference* pass. Both of the non-concurrent passes must be modified to operate properly.

Pass 1, *count local arcs*, must now consider whether it is the root node or a successor being visited. In Deb's original version, the actual reference count of a Garbage Can entry is not decremented until *after* collection scanning. In the concurrent environment, actual reference counts are decremented immediately, just prior to placement in the Garbage Can. Roots referenced by the Garbage Can will, in Deb's approach, have reference counts which are one more than those in the concurrent approach. Therefore, in the concurrent algorithm, *count local arcs* must not increment the local reference count of a root node **R** unless it is part of a cycle. Then, only the incident edges from a successor of **R** are counted.

The *collectibility* pass also must be changed slightly. If, subsequent to pass 0, the mutator actions cause the mutator-visit bit to be set for a node, the collector

i) becomes pessimistic,

ii) does not mark the node (potentially) collectible, and

iii) continues to be pessimistic for all successors of the node.

Initially, it might appear that the collector could avoid traversing below a node with the mutator-visit bit set. This is not the case for shared subgraphs.

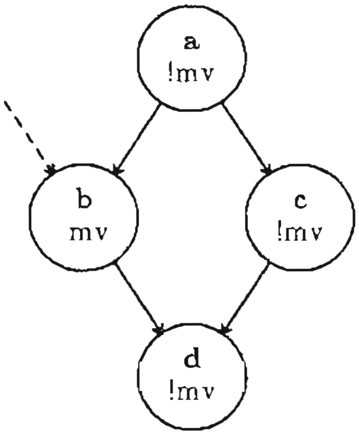


Figure 3.2. A shared subgraph.

Given the subgraph rooted at **a** (Figure 3.2), if node **b** is referenced by a node which is not reachable within the subgraph, it could have its mutator-visit bit set following traversal 1. Since there is no implicit *forwarding* of the update to **b**, none of the successors of **b** will be considered invalid for collectibility during the current cycle. The collector could (incorrectly) decide that all references to **d** have been accounted for, and that the node is only reachable through **a**, thereby allowing **d** to be deleted. The *collectibility* traversal must therefore propagate the mutator-visit information by continuing, *in a pessimistic state*, to the successors of the invalidated node. This method ensures that such successors are *not* marked as potentially collectible, and is necessary for (C1).

Proposition 2:† At the conclusion of the *collectibility* pass,

(C1) each node reachable from the root **R** along a path containing only nodes marked as collectible is assured not to be reachable from any node of the active graph, and

(C2) any node in the active graph having a successor whose local reference count is non-zero either has its own local reference count non-zero, or has its mutator-visit bit set.

Proof: Let $\mathbf{G_R}$ be the maximal, strongly-connected graph rooted on **R**. (C1) assumes the local reference count of every node visited in Traversal 1 was zero at the start of the traversal, and that no node's pointer contributes to the local reference count of any other node unless both were present in $\mathbf{G_R}$ at the start of Traversal 1. Thus, the local reference count of each node at the end of Traversal 1 is less than or equal to the number of references to the node from within the graph. If no node of the graph has its mutator-visit bit set, then all will have been visited in Traversal 1, and the local reference counts equal the number of references from within $\mathbf{G_R}$. Thus, the nodes of $\mathbf{G_R}$ are marked as collectible in Traversal 2 if and only if Condition (C) holds. This establishes (C1). The proof of (C2) is analogous to the proof of Proposition 1. □

Condition (C1) guarantees that only unreachable nodes will be collected. Condition (C2) is sufficient to assure that at the conclusion of pass 0, every node reachable from a root **R** along a path containing no node whose mutator-visit bit is set will have a local reference count of zero.

Some improvements and optimizations can be made to the algorithm if the

† Both the theorem and proof are from the justification provided by Kieburtz in [Kie85a]

domain in which it is applied is restricted. Additional discussion regarding these possibilities is provided in Sections 4 and 8. The concurrent algorithm, as described, has formed the basis of a fairly complete simulation of a sophisticated list-structured memory design, to be discussed in Sections 6 and 7.

4. An application of the concurrent algorithm.

The concurrent algorithm presented in section 3 relies on the use of two processors. It also relies upon a list-structured "graph memory" that is shared between the processors. Many machines could exploit list-structure memory. One architecture that can readily make use of this arrangement is the G-Machine.

The G-Machine [JoA83, Kie85c] is a programmed graph-reduction processor. It is designed to be a co-processor in a dual processor system with shared memory. The memory management tasks are implemented with a minimum of cooperation from the G-Machine. The G-Machine is able to view memory as an (unlimited) list-structure store, implemented by the concurrent memory manager and memory controller. This perspective can be helpful to the design and efficiency of list processors such as the G-Machine. Two particular issues arise in a G-Machine application. A concrete method for node allocation must be selected, and a mechanism for dealing with memory references internal to the G-Machine must be established.

A general technique for allocation of new nodes is to maintain free nodes on a one-way linked list (freelist). Deallocated nodes are added to the tail of the list, nodes to be allocated are taken from the head of the list. This technique is attractive because it is both simple to implement and only requires producer-consumer constraints for concurrent access. It does, however, introduce fragmentation; this problem can be reduced, and a possible solution is referenced in Section 8.

The G-Machine uses a stack of pointers (P-stack) into the graph to locate function arguments. Operations on this stack can create or destroy copies of pointers without actually altering the graph store. We do not want to burden the memory management system with maintaining reference counts of nodes referenced by the P-stack; this would require a reference-count processor somewhat faster than the G-

Machine itself. If a node referenced by the P-stack is never connected to the active expression graph, it will never have its reference count incremented, and thus, would be considered a temporary node.

A suitable technique for inspecting these temporaries for collection would be modified stack deallocation. This technique is based on the notion of *procedure contexts*. A context is opened when a procedure becomes active, and is closed when the procedure returns. While a context remains open, pointers to the nodes allocated (and added to the P-stack) would be pushed onto a list (another stack) of nodes pending inspection/deallocation. This list, called a Context Frame, would be examined following closure of the context. When the context closes, the G-Machine removes from the P-stack all node references that were added to the P-stack during the context. The only pointers to these nodes that might remain accessible to the G-Machine are those that were written into the graph. These nodes are said to *persist* following closure of the context. The Context Frame for the just-closed context will consist of

(i) nodes with zero reference counts,

(ii) nodes with non-zero reference counts, but that are contained in a graph not connected to the active expression graph, and

(iii) persistent nodes.

Nodes in group (i) could be deallocated outright (stack deallocation), since they are no longer accessible via the P-stack and were never referenced by the active graph. Nodes in group (ii) would have to be submitted to the modified reference counting algorithm to detect cycles that might account for their non-zero reference counts. Persistent nodes would be collected as described in Section 3, through the Garbage Can.

To accommodate the persistence notion, an additional tag bit is associated with each node. When a node is allocated (removed from the freelist), its persistence bit is cleared, and a pointer to it is stacked on a list of nodes to be inspected when the current context closes.

When the collector examines nodes referenced by a closed Context Frame, it will find each node to be either (1) immediately collectible or (2) persistent. If a node α is in group (i), it is added directly to the freelist. A node α is in group (ii) if α is part of a cycle with the property that every node reachable from α is referenced only by nodes reachable from α. If a node is neither in group (i) or (ii), then it must be reachable from a node of the active expression graph, and therefore, is marked as persistent. This situation, in turn, affects the conditions under which the nodes are passed to the Garbage Can.

According to Deb's algorithm, whenever a node has its reference count decremented, a pointer to the node is passed to the Garbage Can. With the use of persistence, this rule does not stand. If the persistence bit is set for a given node, then whenever the reference count for the node is decremented, a pointer to it is passed to the Garbage Can. If the persistence bit is *not* set, then either (a) the node is part of an open allocation context, or (b) the node is part of a closed allocation context that has not been fully inspected for collectibility.

Only under certain conditions of (b) should a pointer to the node be passed to the Garbage Can. If the node satisfies (a), then it may be referenced via the P-stack, and will remain uncollectible until the context closes; subsequently it will be inspected for collectibility via the closed context frame. However, when a reference to a node satisfying (b) is deleted, and the node is pointed to by a closed Context Frame, an entry for the node must be added to the Garbage Can. We can detect if a node satisfies (b) by examining its mutator-visit bit.

Whenever a reference to a node is deleted, its mutator-visit bit is set (mv.1). When a node is examined for collectibility, its mutator-visit bit is cleared (mv.0). If at the time a reference to a node α is deleted, the mutator-visit bit for α is zero, then the initial pass of collectibility scanning has visited α. In addition, if the persistence bit for α is not set, then it is in a closed Context Frame that has not been completely inspected. Thus, α satisfies (b), and a reference to it must be placed in the Garbage Can.

The following scenario illustrates the problem incurred if we fail to add certain nodes that are not persistent to the Garbage Can.

Given that a node **n** was allocated during an allocation context **AC**, assume that **AC** has been closed. This means that all nodes referenced by **AC**, including **n**, can be inspected for collectibility. Given that the active expression graph still contains one reference to **n**, consider the following:

1. The Memory Manager performs *clear local reference* pass over **n**, zeroing the mutator-visit bit and clearing the local reference count of **n**.
2. The Mutator deletes the edge leading from the active graph to **n**. This decrements the reference count of **n** to zero, but since the persistence bit is not set, a pointer to **n** is not placed in the Garbage Can. The mutator-visit bit is set.
3. The Memory Manager continues collection, but when testing for collectibility, discovers the mutator-visit bit for **n** is set. The collection of **n** is thereby disqualified, and the persistent bit is set.

Once the collection scan of **AC** completes, all references to **n** will have been discarded. Since the frame in which **n** was allocated is no longer active and since the active graph no longer contains any references to **n**, the node is both uncollectible and unreachable.

Stack allocation with persistence, as described in this section, provides for simple, dynamic allocation. The overhead of counting P-stack references has been avoided by introducing the notion of persistence for non-temporary nodes. Thus, the requirements of a list-structure memory mechanism for the G-Machine have been established. The following section discusses the architecture of such a memory.

5. Architecture for a dual-ported list-structure memory.

Any system that has use for list-structured memory should be able to view storage as an unlimited supply of nodes. This model, in fact, is what proposed "real time" garbage collection [LiH83,Wad76] has attempted to provide. In these approaches, the mechanisms used to provide the seemingly limitless supply require multiple processors. Our approach, likewise, relies upon a separate processor to perform complex memory management activities. Also important to our design is a special dual-ported memory that provides concurrent access and atomic manipulation of tag fields. This section describes a macro organization of processors that provide automatic, dynamic list-structure memory.

The design presented here is driven by two factors.

(1) The need to develop and evaluate a system that makes use of the parallel, incremental garbage collection strategy described earlier.

(2) The requirements of the G-Machine [Kie85c] for a dynamic, list-structure memory.

In the most general sense, our design provides a foundation for both. The fundamental processors of the design map well into components in a complete system that contains a G-Machine co-processor. Furthermore, the top-level architecture is compatible with processors other than the G-Machine that require list-structure memory.

The memory system is embodied in three primary "processors"[1], with two simpler supporting processors. We call two of the main processors *Memory Manager*

[1] We will use the term *processor* loosely in this section to represent a functional processor Some of the processors must be programmable machines, while others may be realized as sophisticated hardware with embedded logic (e g , a hardware fifo)

and Collector and *Mutator* (Figure 5.1). In a typical arrangement, these two processors would correspond to the *host* and the *co-processor*, respectively.

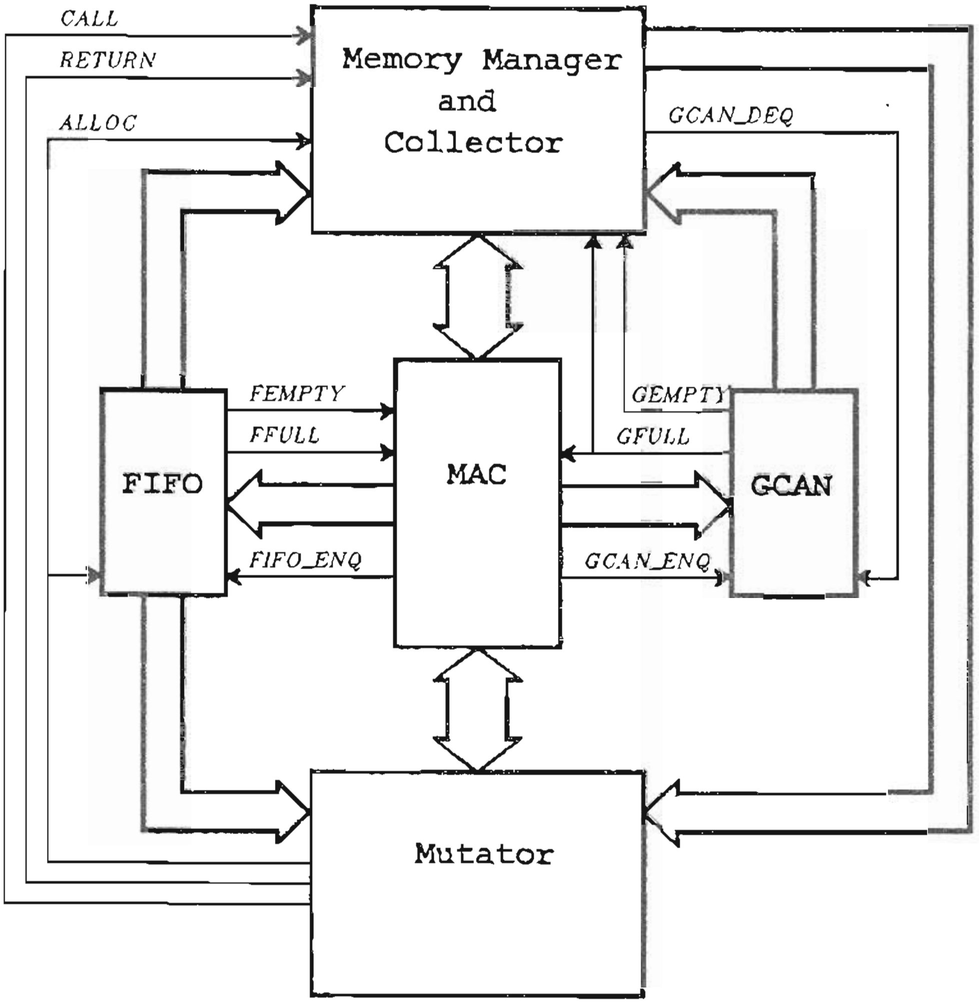


Figure 5.1. Dynamic, list-structure memory components.

The Memory Manager is responsible for concurrent collection of garbage nodes while the Mutator is a graph-operation engine. These two share the main store via the third main processor: the *Memory Access Controller* (MAC). Part of the MAC is the complete list store available to the Mutator. The MAC also provides high-level operations at the node level.

As a co-processor, the Mutator has local instruction and stack store. The Memory Manager, or host, also has its own local store and provides access to secondary store. The host provides user services, such as I/O, operating system software, and initialization and control of the co-processor.

The three main processors are assisted by two hardware queues. The FIFO provides a demand-driven stream of nodes from the freelist to the Mutator. The Garbage Can buffers a list of nodes that are potentially garbage between the MAC and the Memory Manager. The general tasks required of each processor are described below. A detailed explanation of the inter-communication and signal requirements is provided in Appendix A.

5.1. Duties of each component.

Each processor has a particular task assigned. The task may take the form of a single operation or many operations. A processor repeatedly performs a sequence of operations, unless interrupted to provide some alternate service. Once the the alternate service has been provided, the process returns to its previous sequence of operations.

5.1.1. Mutator

The Mutator accesses and modifies the graph (G-Memory). This manipulation is done by one of the following operations.

1) Allocate a new node, to be added to the graph in a subsequent operation.

   The FIFO is asked for the next available node. Allocation does not affect the structure of the graph per se, however, it causes the number of unused nodes to be reduced, and makes a node available for subsequent Read/Write operations.

2) Read the contents of a node.

Reading involves the retrieval of particular cell data in a given node. The MAC is asked for the contents of a node. The state of the node accessed remains unchanged.

3) The Trash operation. Write by Update.

The MAC is instructed to trash a node. Reference counts need to be adjusted when a node is to be overwritten and the node contains pointers. The trash operation immediately precedes a write to a node containing pointers. The reason for Trash is two-fold. Trashing a node serves to simplify the write operation of the MAC. It also provides the possibility of overlapping Mutator procedures associated with a write of pointers with the MAC operations required to maintain reference counts and tag bits.

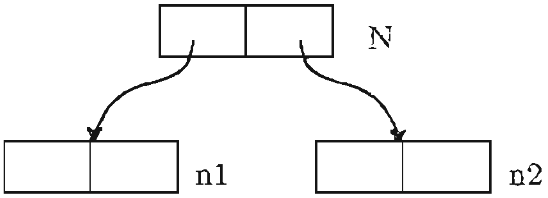


Figure 5.2. A graph node and its successors.

For example, given the nodes in Figure 5.2, if the MAC is told to Trash **N**, then the Mutator-visit bit will be set and the reference count decremented for both **n1** and **n2**.

4) Write the contents of a node.

The MAC is instructed to write data into a given node. The tag bits are passed to the MAC to indicate the type of data (pointer or simple). The Mutator-visit bit for the referenced node is set. If a pointer is being written into a node, the destination of the pointer has its Mutator-visit bit set and its reference count incremented.

5) Context switch.

In addition to explicit graph operations, the Mutator enters and leaves contexts[1]. During a given context, the Mutator maintains pointers to nodes in the graph memory and to freshly-allocated nodes (that may not have been written to yet) on a local stack. When a context is closed, the Mutator removes the pointers on its stack associated with the context, thereby releasing its internal references to those nodes.

The design of the Mutator requires it to repeatedly modify the graph using one of the operations described above. The Mutator is never interrupted and is only delayed if there are no nodes available for an allocation operation.

5.1.2. Memory Manager and Collector

The Memory Manager coordinates the recovery of garbage nodes, and instructs the MAC to add them to the unused node pool[2]. A record of nodes allocated during active function contexts is maintained by an Active Alloc'd Nodes Stack (AANS), with each set of nodes for a given context grouped into separate Context Frames (Figure 5.3). An entry for each context frame is placed on the AANS. The nodes "referenced" by the AANS entries represent all those possibly on the pointer stack of the Mutator. A list of nodes that are potentially collectible is obtained from the AANS: whenever a function call returns, the Memory Manager is notified, and pops the reference to the Active Context Frame (ACF) from the AANS. Each of the

[1] This corresponds to procedure calls and exits

[2] The *unused node pool* represents nodes that are known to be available for use. One common technique is to keep these nodes as a *freelist* We employ the freelist mechanism here for simplicity. A potentially superior scheme is suggested later

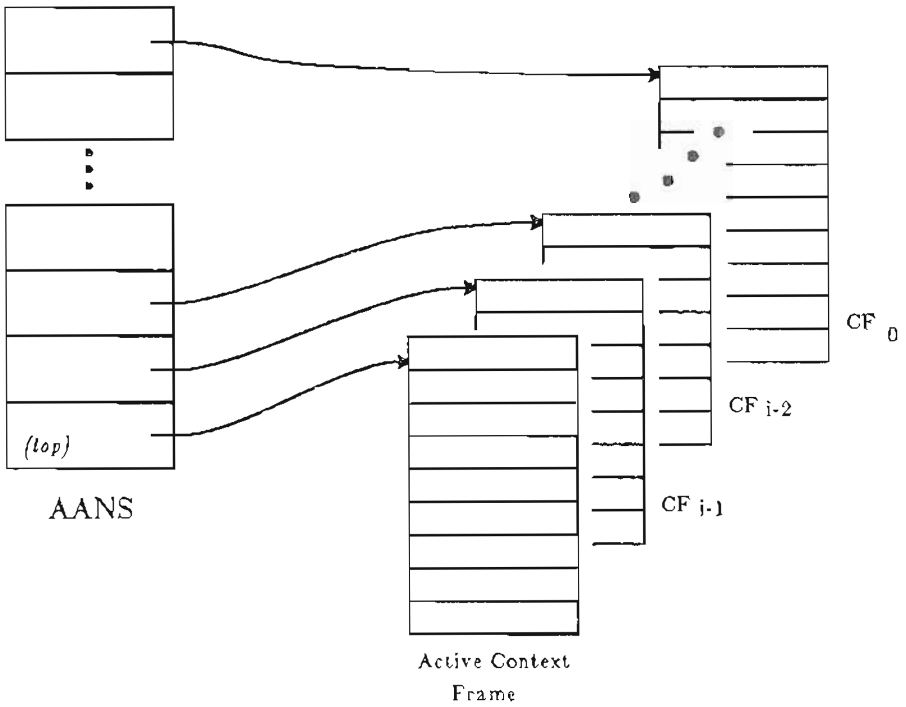


Figure 5.3. Memory Manager allocation frame structures.

nodes from this now-closed context frame is inspected for collection and is immediately collected, or is marked as not currently collectible.

In addition, the Memory Manager receives a list of roots of potentially collectible subgraphs from the Garbage Can. Each of these subgraphs are inspected for collectibility. Any part of the subgraph that is collectible is collected.

A node from the Active Context Frame that is determined to be collectible is passed to the MAC with the directive to add the node to the unused node pool. Nodes that are not collectible are marked as persistent, and are left to be collected by a different mechanism, through the Garbage Can (see the operations of the MAC).

The duty of the Memory Manager then, comprises the following operations:

1) When a node is allocated, add it to the Active Context Frame.

2) When a CALL is signaled, push a pointer to a new ACF onto the AANS. Subsequently allocated nodes are added to the new ACF until a RETURN.

3) When a RETURN is signaled, remove tbe AANS entry for the ACF, adding it to the list of nodes to be inspected for collection. The previous Context Frame on the AANS becomes the Active Context Frame.

4) When a node is available from the Garbage Can, add it to the list of nodes to be inspected. Continually attempt to exhaust the list of nodes to be inspected for collection.

5) When GFULL is signaled, the Garbage can is full, and must be (at least) partially emptied.

5.1.3. Memory Access Controller (MAC)

The MAC provides "dual-port" access to the graph memory. The following operations support the needs of the memory structure.

1) READ/WRITE a cell in a graph node. Some types of writes may be complex: they may (implicitly) change a reference count and they may cause the node to be a candidate for collection.

2) Maintain the unused node pool:

   (i) preallocate nodes from the unused node pool into the FIFO,

   (ii) add nodes to the unused node pool.

   The unused node pool is kept as a freelist: a queue of pointers. The freelist head and tail addresses are kept in special registers of the MAC. The Memory Manager may detect a node in the graph is garbage and should be added to the unused node pool (DEALLOCATE). Nodes pointed to by a collected node have their reference counts decremented (and are not added to the Garbage Can)[1]. Whenever possible, and until the FIFO is full, the MAC will preallocate nodes by removing the head of the freelist, placing it in the FIFO. When nodes are preallocated they are initialized.

3a) Make reference counting operations automatic: Increment and decrement reference counts when needed. This is decided based on the type of write the Mutator requests, and the tags of both the existing and new data.

3b) Determine when a node is potentially collectible. A node *may* become garbage whenever its reference count is decremented. When the reference count is decremented, the value is compared with the threshold[2] for the node.

i) If the reference count is zero and the persistent bit is set, the node is immediately collectible, and is added to the unused node pool.

ii) If the reference count is less than or equal to the threshold and the persistent bit is set, the node is potentially collectible, and is passed to the Garbage Can for collection scanning.

If the persistent bit is not set, there exists a pointer to the node in some frame on the Active Alloc'd Nodes Stack, and will be subsequently inspected for collectibility.

[1] There is no need to consider putting the subordinate nodes in the Garbage Can, even though the reference count is decremented. If they are garbage, the Memory Manager will detect this by reaching them from the parent node, and will issue a subsequent DEALLOCATE If they are not collectible, then the Mutator still has access to them and can cause them to be added to the Garbage Can by subsequent actions.

[2] Systems that do not implement thresholding would consider a node to be potentially collectible whenever the reference count is decremented. See subsection 8 1, "Suggestions for additional work" for details.

5.1.4. FIFO

The First In First Out queue contains preallocated graph nodes. The FIFO receives node addresses from the MAC and, on demand, sends a node address to the Mutator and Memory Manager. The status of the queue is returned to the MAC for each address stored. If the queue is full, the MAC defers FIFO "filling" operations. Upon receipt of the ALLOC signal from the Mutator, the head of the queue is supplied to both the Mutator and the Memory Manager. If the queue is empty, and the FIFO receives an ALLOC, the FIFO signals the MAC with FEMPTY, then awaits delivery of a new node address. Once supplied by the MAC, dispatch of the node address proceeds as if dequeued from the FIFO.

5.1.5. Garbage Can

The Garbage Can contains addresses of roots of potentially collectible subgraphs. The Garbage Can acts very much like the FIFO. Node addresses are received from the MAC, and are placed in a local queue. The Memory Manager requests addressed from the Garbage Can with GCAN_DEQ.

If the Garbage Can is allowed to fill, GFULL is sent to the Memory Manager and the MAC. The Memory Manager then devotes as much resource as possible to relieve the full Garbage Can. If another node must be sent to the Garbage Can during the GFULL state, the MAC suspends until the Garbage Can can receive the node. Note that with a hardware fifo, the Garbage Can can be read from and written to simultaneously, as is allowed here, thus the lock-step nature of the GFULL state is minimized.

6. Detailed simulation organization.

To evaluate the workability of the list-structure memory design, a simulation has been constructed. The simulation incorporates the parallel version of Deb's algorithm and a specific emulation of the proposed architecture. This Section provides a discussion of the simulation implementation.

The simulation constructed provides for apparent concurrent functioning of each of the components of the previously described macro architecture. The multiprocess capabilities of the UNIX 4.2bsd† operating system were exploited by a group of programs. Each separate program, running as an individual process, represents component of the architecture. In the process structure, an additional *Parent* process is added to facilitate instrumentation, data collection, and simulation control. As a consequence of simulation, some aspects of the original design of were revised to arrive at the specification given in Section 5. Specifics of the simulation and how it differs, in detail, from the architecture of Section 5 will be discussed shortly. First, though, we describe the system facilities used.

6.1. Computing/utility environment.

The simulation was developed on a variety of machines, each running UNIX 4.2bsd. The figures reported later are summaries from the final version of the simulation, run on both a DEC VAX 11/780 and a VAX 11/750. Some preliminary versions of the simulator were also run on a high-performance 68000-based Tektronix workstation (no data presented was generated on this machine, however). A variety of machines was used to confirm that the real-time computations were not an aberration of a particular host machine.

† UNIX is a trademark of Bell Laboratories.

Several features available in UNIX 4.2bsd were exploited. Multiple-process capabilities were realized with fork(2) and exec(3). Inter-Process Communication (IPC) employed the IPC utilities, including socket(2) and kill(2). A set of utilities to provide simple byte-oriented sockets was developed[1]. For controlling activities of the Mutator, pseudo-random sequence generation made use of the random(3) library. To provide control of the comparative (real-time) activities of processes involved in the simulation, an adapted profiling scheme was used. The application of these libraries and utilities makes possible a fairly large-scale, informative simulation.

6.2. Profiling: control of comparative performance.

Traditionally, profiling has been used to discover inefficiencies or to track the execution flow in a set of procedures. Our usage of profiling was to monitor real execution times for the various processes. The multiple-process nature of the simulation required adaptation of the default profiler mechanism.

The execution profiler *gprof* [GKM82] and its associated libraries are designed for single-process programs. The raw profiling data is placed in a file "gmon.out". This file represents call counts and execution times for each profiled routine. To examine the characteristics of a program, the "gmon.out" file is interpreted, using the object code of the program. However, when multiple, concurrent programs are being executed, more than one "gmon.out" file needs to be created.

A special-purpose object code editor was constructed to alter the executable code for a given process and thereby, provide different ".out" files for each process. The *gprof* utility could then be used to compare the "simultaneous" execution of multiple processes by separately examining an executable program and its associated

[1] Specifically, the utilities used AF_UNIX domain, SOCK_STREAM style sockets.

".out" file. This information, in turn was used to monitor the effective time allowed for both the Mutator and the Memory Manager, ensuring that the processor resources given to each were equal. By assuming that the computing power of each of the two processors is equivalent, we can readily compare the ability of one to "keep up" with the other.

6.3. Modeling hardware characteristics.

Three basic characteristics of an actual implementation are modeled. Digital logic, both programmed and hardwired, is provided by collections of procedures, algorithms, and data structures designed to represent the logic behavior. Communication, or data transfer, is made possible by the use of *sockets*, an inter-process communication facility. Asynchronous controls, interrupts, are provided by *signals*, another component of the IPC facilities.

6.3.1. Processes: component to UNIX process mapping.

Each component of the macro architecture is simulated as a separate process. This approach provides a model for concurrent operations, much as would be the case in a hardware implementation. The functionality of a given component is encoded into a set of procedures that compile into an almost stand-alone main program[1]. A supervisor process, the *Parent*, initializes the multi-process environment, then arranges for the execution of each of the other processes. In this manner, just as with a hardware implementation, the system is initialized and booted.

Once all processes are running, communication channels are opened between pairs of communicating processes. This sequence is done by a carefully synchronized

[1] We say "almost" because each process relies on the proper establishment of communication with other processes to become fully functional

set of procedures that first create, then open, the communication link. At this point, each process functions autonomously, much as multiple users seemingly operate independently.

Synchronization occurs as a matter of data demand via the socket mechanism, or via signals sent and received. Each of these mechanisms provides the functional component interactions needed to make the system operate.

6.3.2. Data paths: sockets.

The communication paths between processes are treated analogously to a data/address bus. Data is written by the sender to the socket and is read by the receiver from the socket. The receiver can rely on one of three methods to acquire the data.

i) *poll.* The receiver checks to see if there are any data to be read, without actually reading them. If data are available, reading can proceed when convenient.

ii) *interrupt.* The receiver is signaled when data become available to be read. If this technique is used, the data should be read immediately to avoid overlap with a subsequent signal on the same channel, and possible loss of data.

iii) *block.* Finally, the receiver may simply block on a socket. To block, the receiver goes into a read-wait loop that continues until data become available.

Most data paths implemented in this simulation require the receiver either to poll or to block. As it turns out, this approach may be somewhat inefficient for some cases. The polling technique caused the Garbage Can, one of the least complex logic components, to consume the largest amount of CPU cycles. See the "Simulation Artifacts" subsection below.

By default, sockets are allowed to provide data buffering. Since a bus, itself, does not provide data buffering, this characteristic is completely disallowed for the simulation. This restriction requires careful checking of the status of a socket prior to and following the data transfer. In some cases, two-way communication between the sender and receiver provides the status information. This communication control corresponds to a typical request/acknowledge handshake. The recipient of the data transfer transmits a status word *on the same socket* once reading is complete. When a process desires to write to a socket, it waits for the recipient to transmit receipt status before it actually does the write. In summary, this method disallows the writer on a socket to "runaway" until it reaches the socket buffer limit; rather, it inhibits socket writes to a transaction-level (a few bytes), and thus more closely simulates the operation of a bus.

The particular type of sockets used allow multiple simultaneous writers and multiple simultaneous readers for the same socket, with no collision. The simulation did not attempt to employ the multiple-users-per-socket scheme, however. One socket is the endpoint for communication between *exactly* two distinct processes. By applying this restriction, no special tagging or encoding of data is necessary; the source and destination of a data transfer are unambiguous.

6.3.3. Control: signals.

Asynchronous control is implemented with signals. Signals provide 31 different software interrupts, each with equal priority. Most signals are generated by conditions in the operating system and run-time libraries. Four are left available to the user. Several system-defined signals may be "aliased" for other uses, providing the user ensures the aliased signal *cannot* be generated by the default mechanism.

Each process defines a set of procedures to handle signals it wants to accept. When a process A sends a process B a signal, the procedure designated in process B to handle that signal is invoked. Termination of the procedure returns process B to the control state it was in immediately prior to the delivery of the signal. Side effects are allowed, however, so global states may be altered by the signal handling procedure.

To provide indivisible operations, a process B blocks all applicable signals. During the atomic operation, no other process may interrupt B with a signal. Once the critical operations are complete, the process B unblocks the blocked signals. If signals are sent to a process while it is blocked, they are delivered once the process unblocks them.

Signals may be missed if more than one signal of the same type is sent to a process before it completes handling the first one. To help avert this situation, signal handlers are kept fairly simple. Detection and correction of missed signals must still be provided, though. Part of this need is due to the somewhat irregular mechanism used by the operating system to deliver the signals; part of it is due to the inherent complexity of some of the processing required of the signal handlers. See subsection 6.5, "Simulation artifacts and limitations", for further details.

6.4. Description of processes, dataflow, and control.

The functional characteristics of components of the system correspond closely to the description given for each component in Section 5. However, there are some differences in how the desired behaviors are achieved. These details are considered to be implementation specific, but are important to the generation of the simulation data. The following discussion separates implementation details into three areas: process operations, communication between processes, and asynchronous control.

Within each of these areas, the changes and additions to the description in Section 5 are detailed.

6.4.1. Process operations.

The behavioral characteristics are somewhat different for all components. An additional process, called the *Parent* process, is added for simulation setup, simulation control, and data collection. Each architecture component described in Section 5 corresponds to a process.

Mutator. For convenience, we review the possible actions of the Mutator. An ALLOC causes the allocation of a new node, to be subsequently added to the active graph. A READ simply examines the contents of a node at a specified address. WRITE provides data and tags to be stored in a particular node; the tags distinguish "simple" and "pointer" data. The TRASH operation is used immediately preceding the WRITE of pointer data. Finally, context switching is defined by CALL and RETURN. When a context is closed, the Mutator discards references to nodes that are not in the active graph.

There are two significant changes to the Mutator. The cycle of operations of the Mutator does not correspond to a sequence of instructions performing useful graph operations; rather, the cycle is pseudo-random and insensitive to particular sequences of graph operations. In addition, the destinations for particular graph writes are sometimes provided by the Mutator and sometimes come from a node list maintained by the Memory Manager.

The chain of operations performed by the Mutator is driven by a pseudo-random sequence with parameterized, discrete distribution. The actions are controlled as follows.

For each Mutator operation, a given desired frequency is assigned. The frequency is unconditional on previous operations, and is expressed in terms of total Mutator operations. For example, in a particular simulation we could specify an ALLOC rate of 1/10. This means that one ALLOC would be performed for approximately every ten Mutator operations. The relative frequencies are mapped into disjoint ranges across a scale of 0-99, where the range for each operation represents the frequency. A pseudo-random number generator (modulo 100) is used to probe the scale. The Mutator operation that covers the range probed is selected. Thus, a wide range gives higher occurrence frequency, a narrow range yields lower frequency. This technique hopefully provides statistical properties of a most general set of graph operations without relying on explicit evaluation of particular algorithms. Also, this approach allows us to "tweak" the allocation rate back and forth to examine just how much allocation/deallocation the collection mechanisms can handle.

For each operation executed, the Mutator "visits" a node of the graph. The recursive traversal of the graph is preorder. The recursion ends once a cycle is completed, a NIL pointer is encountered, or non-pointer value is encountered. Traversal proceeds until all nodes in the graph have been visited. Upon completion of a traversal, a new traversal commences at the same root. During each traversal, an explicit stack of the edges leading from the root to the present node is maintained. This stack is used both for the traversal mechanism and for generation of destinations on some graph writes.

Whenever a pointer is written into a node of the graph, its target is either a newly-allocated node or is a node already in the graph. In an actual implementation, the pointer to a node already in the graph would be held within the Mutator's own local store. This structure could take the form of a stack of pointers into the graph [Kie85c, Sar84].

Graph writes may create cycles in the graph. To ensure the generation of a new cycle, a pseudo-random probe is made of the traversal stack. The resultant address is used as the destination of a pointer. This mechanism is used in conjunction with another tactic that does not necessarily produce cycles, but still allows creation of pointers back into the graph.

A stack of references to nodes allocated is kept by the Memory Manager. We call this the Active Alloc'd Nodes Stack (AANS). Each time the Mutator needs a pointer to a node already in the graph, it makes a request to the Memory Manager. The Memory Manager responds with a node selected from the AANS. The node is guaranteed to be in the graph already.

Thus, two mechanisms provide creation of arcs that lead back into the graph, and thereby allow some control of both the incidence of cycles and of shared subgraphs in the active graph.

Memory Manager. The Memory Manager interacts differently with the Mutator and with the Garbage Can. First, as described above, the Memory Manager handles requests from the Mutator for addresses of nodes guaranteed to be in the graph. These are called *Active Node Requests.* Whenever the Mutator receives an active node request, it uses a pseudo-random number generator to probe the nodes referenced by the Active Alloc'd Nodes Set. Since these nodes are part of open contexts, they are guaranteed either to already be part of the graph or to be presently uncollectible (being in open Context Frames), so can safely be destinations of active graph nodes. An additional communication path is added between the Memory Manager and Mutator to return the selected address, and an additional signal allows the Mutator to request a node address asynchronously.

Secondly, the interactions between the Memory Manager and the Garbage Can are modified in detail from that described in Section 5. The Memory Manager regularly queries the socket from the Garbage Can. If an address is available, the Memory Manager "reads" the socket and places the address in a queue of nodes to be inspected. If no address is available, the Memory Manager proceeds with other tasks without reading the socket.

Memory Access Controller. Contrary to the description in Section 5, the MAC does not add nodes to the freelist immediately upon detecting them to be unreferenced. This difference is viewed as an optimization at the cost of complicating the control logic of the MAC. Our intention in the design of the MAC process was to attempt to keep it simple and fast. This notion is important since the MAC is shared by both the Mutator and the Memory Manager and is not intended to be a "programmed" machine, and because the MAC consumes real time in the simulation. Other changes to the MAC involve how it interacts with the FIFO and the Garbage Can; these changes are described below.

Fifo. The FIFO does not receive or send signals regarding its state. Instead, it uses a handshake with the Memory Access Controller. If a node address sent by the MAC causes the FIFO to be full, the status returned to the MAC is "fifo full". Otherwise, the status returned is "ok". If the MAC receives "fifo full" status reply, it suspends filling, then regularly checks the status of the fifo. Only once the status is "ok" does it resume FIFO filling.

Garbage Can. Unlike the description in Section 5, the Garbage Can receives and sends no signals. Rather, it uses handshaking and polling protocols. Whenever a node address is passed to the Garbage Can from the Memory Access Controller, it is enqueued, as before. In addition, if the Memory Manager has "read" the previous node placed in the MMGR ⟷ GCAN socket, the head of the Garbage Can queue is dequeued and placed in the socket. If the Garbage Can queue becomes full, the return status to the MAC is "full", and the MAC does not attempt to enqueue additional nodes until the status is "ok". Both the checking for Memory Manager reads and the return status to the MAC are handled as special codes passed *through the respective socket*, rather than as signals or control line levels.

6.4.2. Process datapath interconnect.

The communication channel provided by sockets is used to transfer data, addresses, processor directives (analogous to instructions), and to return event status. Each of the sockets used in the simulation is described by name below and is labeled in Figure 6.1.

mac→fifo Carries addresses of preallocated nodes from the MAC to the FIFO. The MAC gets a response for each address sent to the FIFO. Both the node address and the response are passed through the same socket.

mac→gcan Provides transfer of candidate node addresses from MAC to Garbage Can. Similar to mac→fifo, the MAC gets a response for each address sent to the Garbage Can. Both the node addresses and the response are passed through the same socket.

mmgr←fifo When an new node is actually requested by the Mutator from the MAC (see the signal ALLOC), the address is sent to both the Mutator and the Memory Manager. This procedure allows the Memory Manager to maintain the Active Alloc'd Nodes Set. This socket provides the data path from the FIFO to the Memory Manager for the address. The Memory Manager "read" of this port is triggered by the ALLOC signal.

mmgr←gcan Provides transfer of candidate node addresses from Garbage Can to Memory Manager. The Garbage Can gets a response for each address sent to the Memory Manager. The Memory Manager polls this socket to see if an address is available. The response simply indicates that the previous address in the socket has been "consumed", and another may be dequeued and placed in the socket. Both the node addresses and the response are passed through the same socket.

mmgr←mac The data and tag fields requested are transferred through this socket. The MAC "writes", the Memory Manager "reads".

mmgr→mac A directive, node address, write mask, and associated tags are transferred. A directive in this context defines the type of data transfer to or from Graph Memory. Directives specified in Section 5, under "Communication Interfaces", for MMGR ⟷ MAC are used. The mask defines which parts of the tags are stored. The tags may be the local reference count, or the mutator-visit bit. Return status from the MAC is used to enforce unbuffered transfer. For a directive and associated data, the Memory Manager "writes", and the MAC "reads". For status, the MAC "writes" and the Memory Manager "reads".

mmgr→mut An Active Node Request signal to the Memory Manager generates a node address. This address is transmitted back to the Mutator. This socket is an artifact of the simulation implementation.

mmgr→mut2 A backlog estimate of nodes to be collected, a current value for the context frame, and the total number of actual deallocations are

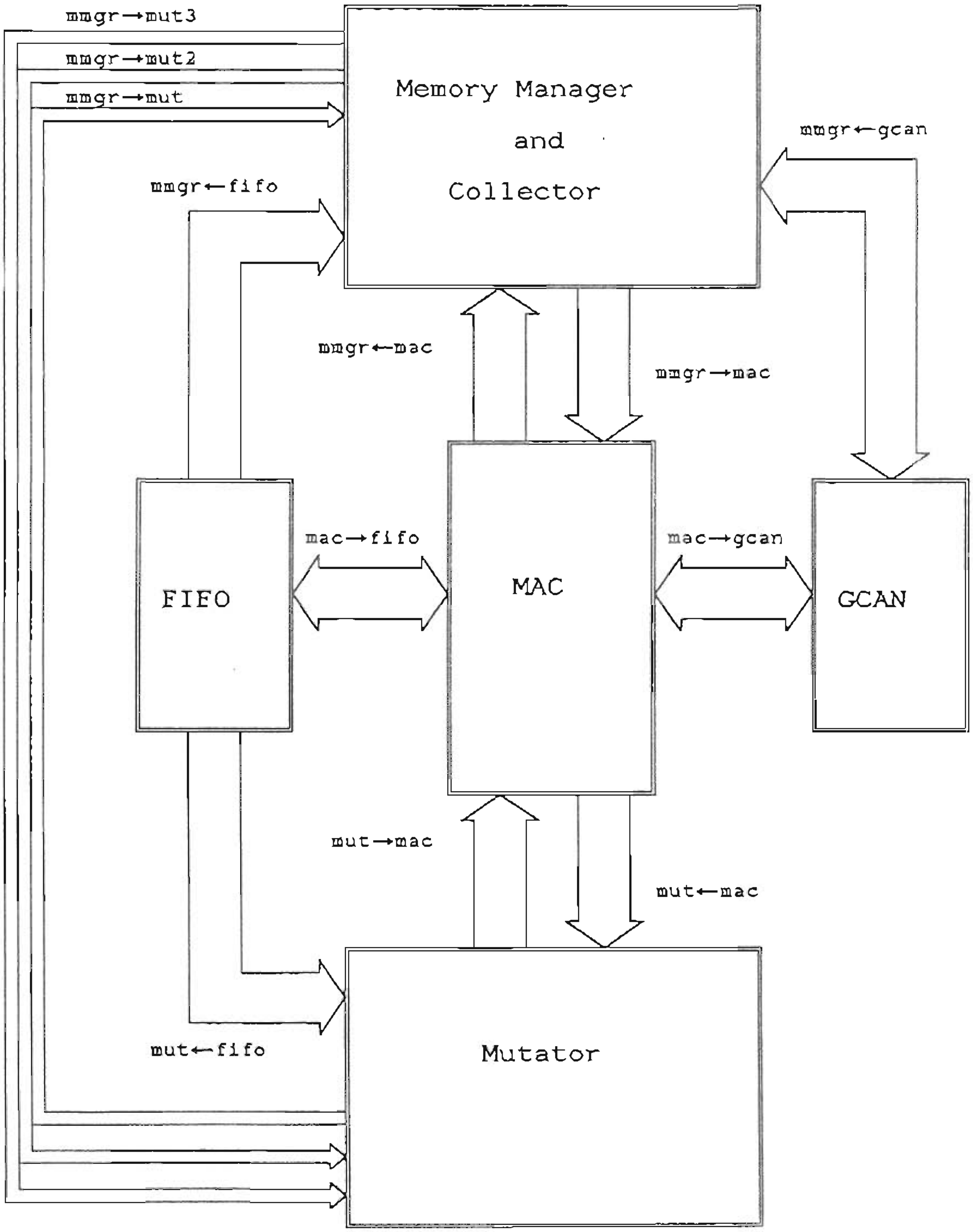


Figure 6.1. Datapaths: sockets used in the simulation.

transferred from the Memory Manager to the Mutator. This transfer only happens during a snapshot. These values are used by the Mutator to monitor possibly missed signals and to control Mutator operation distribution during (early) simulation stages before the desired memory residency is reached. This socket is an artifact of the simulation implementation.

mmgr→mut3 The actual context level of the Mutator associated with each ALLOC is passed to the Memory Manager. The value is used by the Memory Manager to monitor possibly missed ALLOC signals. This socket is an artifact of the simulation implementation.

mut←fifo When an new node is actually requested by the Mutator from the MAC (see the signal ALLOC), the address is sent to both the Mutator and the Memory Manager. This socket provides the data path from the FIFO to the Mutator for the address. The Mutator "read" of this port immediately follows the emission of the ALLOC signal and blocks until the address is available.

mut←mac Data and tags associated with Mutator directives to the MAC are returned. The MAC "writes", the Mutator "reads".

mut→mac A directive, node address, and associated tags are transferred. A directive in this context defines the type of data transfer to or from Graph Memory. Directives specified in Section 5, under "Communication Interfaces" for Mutator ⟷ MAC are used. The tags may be the pointer bits, reference count, or the mutator-visit bit. Return status from the MAC is used to enforce unbuffered transfer. For a directive and associated data, the Mutator "writes", and the MAC "reads". For status, the MAC "writes" and the Mutator "reads".

6.4.3. Process control, signals.

The control of various asynchronous events for each of the processes is provided by signals. Each of the signals used in the simulation is described below and is labeled in Figure 6.2.

There are two types of signals transmitted and handled in the simulation. Global signals are transmitted by the Parent[1] to all the other processes. Local signals are point to point. For each signal, there is precisely one sender, and for all but two, there is precisely one receiver. We first describe the Global signals.

SNAPSHOT This signal controls the collection of data and is generated at regular intervals. Each process writes summary information about its present state to an individual snapshot file. The snapshot interval is measured by the UNIX 4.2bsd interval timer. The length of interval is a run-time parameter. All processes are synchronized at the conclusion of the snapshot period to ensure that no process gets restarted until all have completed snapshot procedures. The synchronization allows for varying costs of summarizing data without skewing the run time charged to the simulation proper. This step is especially important, since snapshots are allowed to interrupt nearly any activity of a process.

SIGTERM The termination signal is sent at the conclusion of a simulation. A run-time parameter controls how many snapshots are taken before the

[1] Actually, many global signals could also be transmitted by the run-time environment or operating system. We will not discuss any signals in this set since none of them are explicitly handled or caught by simulation processes other than the Parent

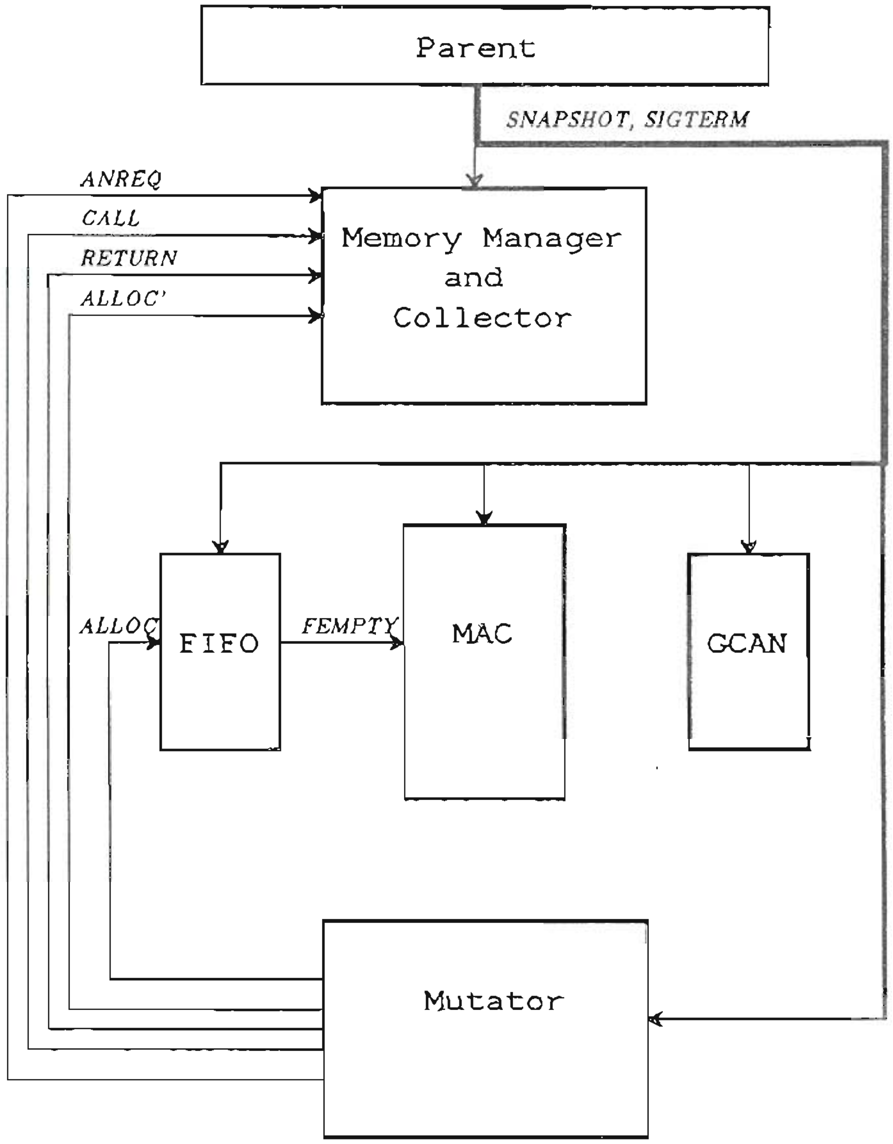


Figure 6.2. Control: signals used in the simulation.

simulation is scheduled to end. Other conditions that may arise during simulation can also cause the Parent to send SIGTERM. If available graph memory is exhausted, if deadlock (accidently) occurs, if a process dies, or if a run time error/interrupt is detected by the

Parent, the simulation will be terminated. Each process then outputs summary information and terminates.

The Local signals correspond very closely to the signals described in Section 5. Some of the signals previously described are not used. In particular, some of the FIFO and Garbage Can signals were not implemented; polling and handshake mechanisms were used instead. An additional signal "ANREQ" has been added.

ALLOC When the Mutator wants a fresh (preallocated) node, it sends the FIFO this signal. The Mutator then blocks until the FIFO transmits the node address. Once the Mutator has the node address, it sends this signal to the Memory Manager, to indicate a new node is available on the FIFO port. This delayed signal (indicated in Figure 6.2 as ALLOC') avoids the need of the Mutator to retransmit the ALLOC signal to the Memory Manager if it must wait for a node (the FIFO is empty) and protects against the loss of the signal if the freelist is empty.

ANREQ An Active Node Request is used by the Mutator to request an address of a node that is certain to be in the graph (allocated and referenced in an open context). The Memory Manager uses a pseudo-random probe into the AANS to get the address. This signal is most certainly a simulation artifact.

CALL The Mutator indicates to the Memory Manager whenever a new context is opened. This signal instructs the Memory Manager to create a new Active Context Frame and push it onto the Active Alloc'd Nodes Stack.

FEMPTY When the FIFO has received an ALLOC, and it is empty, this signal is sent to the MAC. The MAC preallocates a node from the freelist and passes it to the FIFO. The FIFO, in turn, passes the node to the Mutator.

RETURN The Mutator indicates to the Memory Manager whenever a context is to be closed. This instructs the Memory Manager to close the Active Context Frame, enqueue it onto the Closed Context Frame Queue, and re-establish the previous context frame on the Active Alloc'd Nodes Stack as the Active Context Frame.

6.4.4. Important data structures.

There are several crucial data structures supported by the Memory Manager. These provide useful constructions for handling allocated nodes and nodes that are candidates for collection.

Two structures have already been described to some extent. These are used for handling of nodes allocated during contexts that are still open. The Active Alloc'd Nodes Set (AANS) and the Active Context Frame (ACF) are used together. During a given context, references to all nodes allocated are put in the ACF. The top of the AANS references the ACF. When a new context is opened, the ACF becomes merely an open "Context-Frame", a new ACF is established, and the reference to the new ACF is pushed onto the AANS.

Another structure, somewhat analogous to the AANS, is the Closed Context Frame Queue (CCFQ). The CCFQ has references to Context Frames that have been closed. When an open context is closed, the ACF is popped from the AANS and enqueued onto the CCFQ. The previous Context Frame on the AANS then becomes

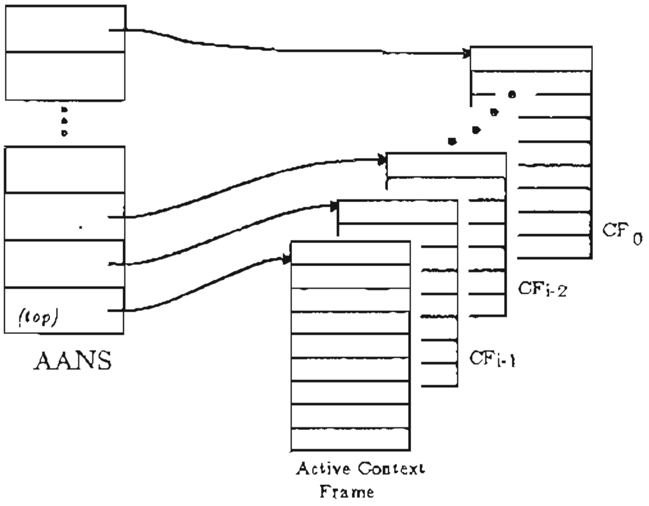


Figure 6.3. AANS, ACF, and Context Frames

the ACF.

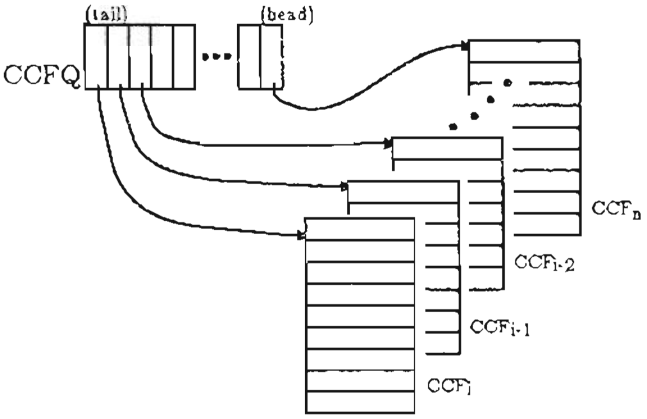


Figure 6.4. CCFQ

The CCFQ, then, comprises the set of nodes from allocation frames to be inspected for collection. An additional queue of nodes, the Candidate Queue, functions in parallel to the CCFQ. The Candidate Queue is contains nodes that are provided by the Garbage Can. The Memory Manager rotates its collection energies through these two queues of collection candidates.

6.5. Simulation artifacts and limitations.

There are several elements of the simulation that either would not exist or would be somewhat different in a concrete implementation. All of these factors affect the outcome of the data; some more seriously than others. None causes the results to be more optimistic than an actual implementation, but they should be used as "seasoning" with the data presented in Section 7.

Data noise reduction.

The multi-user environment used for simulation can, at times, introduce adverse effects in the generation snapshot output data. If during the course of a simulation the load factor of the time-shared host computer changes drastically, either a "flat area" or a "steep slope" could be effected in the overall performance of each of the processes. To avert this, multiple runs using the same run-time parameters are averaged together to generate *characteristic* data. In addition to data averaging, the profiling mechanism previously described is used to verify consistency between separate runs.

Handling lost signals.

Another artifact of the environment used is the ability to "lose" signals. Once a signal is sent to a process, if another signal of the same type is sent to the same process before it has completed handling of the original signal, the second signal could be missed. This situation is analogous to missing hardware interrupts.

A recovery mechanism was used for detecting and handling certain critical signals. Specifically, signals from the Mutator to the Memory Manager for management of the Active Alloc'd Nodes Set must be accounted for in precisely the order in which they occur. The recovery mechanism is ad-hoc, to some extent. Whenever the

Memory Manager receives one of ALLOC, CALL, or RETURN, it resolves any outstanding (previous) changes that have been specified. This procedure is performed by having the Mutator transmit its current context level to the Memory Manager for each node allocated. Thus, the socket from the FIFO and a socket from the Mutator to the Memory Manager are parallel; one has allocated nodes, the other has the context level at which the node was allocated. If, during the "resolution" procedure, the Memory Manager discovers the allocation context level to be inconsistent with its own notion, it performs processing to adjust its internal context level as if it had received signals to make the adjustment.

Other signals that *may* be missed are the global signals. Since these are only for simulation data collection and termination, a time-out mechanism is used. Once the Parent broadcasts a global signal, it sets a timer. Each of the processes responds once it has received the signal. If all processes do not respond by the time the timer expires, the Parent assumes an uncorrectable problem has arisen and terminates the simulation. The assumption in this approach is that if one of the global signals cannot be received by a process, either the process has crashed or deadlock exists. The time-out mechanism is a suitable approach for handling these conditions.

Mutator model.

The Mutator model may predict rather pessimistic Memory Manager performance compared to that of an actual graph evaluator. The model provides for memory operations only. That is, there is no I/O *idle* time. The model essentially creates a worst case scenario, since it effectively asserts that the Mutator has a 100 percent graph memory duty cycle. Previous simulations show that a significant portion of the graph evaluation time is spent on *internal* storage manipulation and computation with data other than pointers [Sar84].

Outright bugs or errors.

Currently, several features of the simulation result in extra work for both the Memory Manager and the Memory Access Controller. In general, the existence of these anomalies can be attributed to oversights during the simulation development. Most were not discovered until late analysis of data from the simulation and profiling.

- The MAC queues Garbage Can overflow, if any. This procedure is less a bug than an implementation shortfall. The signal mechanisms described in Section 5 should have been used. In practice, this problem contributes no anomalies to the simulation, since the size of the Garbage Can queue is made sufficiently large to avoid back-flow into the MAC.
- Nodes that are immediately collectible are passed to the Garbage Can anyway. If the reference count of a given node reaches zero, it could be immediately appended to the freelist, and its descendents, if any, could have their reference count automatically decremented. Depending on the sophistication of the MAC, this action could be feasible. However, it was deemed rather expensive (especially if it causes recursive deletion of a particular structure). The mechanism actually simulated only makes it slightly more expensive to collect a node whose reference count has already reached zero.
- Data transfers from the MAC could be limited to needed data. Presently, most transfers between the Memory Manager and the MAC consist of the data fields and most of the tag fields of a node. Unused components are simply discarded. A more efficient mechanism would transfer only the needed information. In a bus-based implementation, where a complete transfer of all data could take place in a single bus cycle, this problem would not be an issue. However, in the

socket-based mechanism used in the simulation, this increases the number of bytes which must be transferred through the (byte-oriented) socket.

- Perhaps the *most* significant bug in the simulation is the inclusion of an extra subgraph traversal during the collection scanning of a closed context frame. . Each node referenced by an entry in a closed context frame must be inspected for collectibility. Tests must include the possibility of the node being part of a cycle. However, the entire graph rooted at the node being inspected need not be evaluated for collectibility. The oversight of this fact causes approximately a 30 percent increase in the number of nodes the Memory Manager must examine for collectibility scanning of the closed context frames.

Summary of affects on data.

In general, the simulation artifacts and bugs yield a performance penalty for the Memory Manager. This situation means that the reported data tend to examine a "worst case" environment. Some of the analysis on the impact of these deficiencies indicates that the effectiveness of the Memory Manager may be reduced by as much as 20 percent. This bias may not, in fact be detrimental, since it helps set a lower bound on the simulated performance capabilities.

7. Data generation and simulation results.

As outlined in previous sections, a multiple process simulation has been constructed to represent the functional characteristics of a parallel, list-structure memory management system. The simulation was designed for several purposes. First, it provides an implementation of the parallel version of Deb's modified reference-counting algorithm. Second, it allows generation of data based on a dynamic, stochastic simulation of a defined architecture. Finally, the data permit the evaluation of the characteristics and effectiveness of the parallel algorithm. These are considered to be useful both for further development of the G-Machine project and to the implementation of a list-structure memory.

The generation of raw data was made possible by careful instrumentation, simulation (run-time) parameters, and a periodic data collection mechanism. The instrumentation and parameters used for this section are described below. For each set of parameters, several identical simulations were run. The data for a given set was then averaged to generate "characteristic" data sets.

This section describes the observed characteristics in terms of the instrumentation and parameter sets used. Graphs of averaged data indicate the effectiveness of the memory management design and pinpoint its performance limits.

7.1. Data generation mechanisms.

Two elements of the simulation affect the generation of data. Several parameters are used to control the activity of various processes. These parameters are set on initialization of a process and, in general, do not change during execution. The data are gathered by periodically requesting all processes to dump particular facts about their present state. Essentially, the parameters represent a given simulation's characteristics. The collective state snapshot provides a means of viewing those characteristics.

7.1.1. Parameters affecting the simulation.

Those parameters that were used to control the simulation are described below. In the actual simulation, many controls are provided. Only the ones which were varied or were fundamentally necessary for the data reported are discussed.

Graph store size. The total number of nodes available in the graph store was kept at 2K for all simulations. This value affects the ultimate size of the active graph.

Initial graph. For each simulation, an initial graph was created before the actual simulation commenced. A wide range of parameters control the initial graph styles. The initial graph is first constructed as a tree, then edges are introduced to create cycles and shared subgraphs. Parameters are used to control the balance and depth of the tree. The percentage of edges that create shared subgraphs can be controlled. Finally, the number of cycles, and the degree of multi-cycles (cycles within cycles) can be adjusted. The run-time graph alterations of the Mutator are not nearly as finely controlled as the construction of the initial graph. However, the characteristics of the initial graph do impact the simulation, since the nodes in the initial graph are never collected. For the simulations used to generate the data reported here, an initial graph size of approximately 300 nodes was used. The depth of the pre-graph tree was set at 10, the tree was approximately balanced, about 20 percent of the edges created shared subgraphs, and multi-cycles were limited to a factor of two.

Percent of store active. The percent of store active is represented by the number of allocated nodes divided by the total number of nodes in the graph store. For the data reported here, the steady state threshold was set for a memory residency of 80 percent (± 8 percent).

Mutator instruction mix. Explicit control over the rate of allocations and the rate of context switching is provided. Both are specified in terms of total number of Mutator operations. That is, a value of 1/5 means that the specified operation will be performed once approximately every five operations. The term "approximately" is used here, since the value is used to define a distribution scale that corresponds to the desired frequency. The remaining operations are given equal distribution over the remaining distribution scale (e.g., if the specifications of the context switching rate and allocation rate together used 25 percent of the scale, the remaining operations would be assigned equal partitions of the remaining 75 percent). For the set of data that shows the ability of the memory management scheme to keep up, a context switch rate of 1/30 and allocation rate of 1/12 was used. For the set of data that shows performance failure of the system, a context switch rate of 1/15 and allocation rate of 1/12 was used. These numbers were established somewhat arbitrarily, through repeated adjustments and simulations.

The context switch rate provides control over the duration of a given context, and thereby control over the size of the context frames. The allocation rate defines the fixed demand for new nodes once the specified memory residency is reached.

Context depth. The number of open contexts is limited by the context depth parameter. For simulations reported here, a value of 5 is used.

Snapshot interval. To perform actual data collection, the Parent process interrupts the other processes after a timer set for the snapshot interval expires. The elapsed time is measured in real time. For all simulations, the length of a snapshot interval was specified as 10 seconds. In reality, the time averaged from 10 to 12 seconds because of the signal delivery mechanism used by the operating system interval timer.

Total snapshots. Each simulation is allowed to run for a specified number of snapshots. This parameter permits a consistent mechanism by which to terminate a simulation. Each simulation used to generate reported data ran for 1000 snapshots.

7.1.2. Key states gathered.

During the course of a simulation, many states of each process are kept. These states are either recorded each snapshot or are recorded upon termination of the simulation. The records can be used to contrast and compare events during the simulation. States that were used to generate the graphs below are defined here.

Freelist size. The number of nodes in the freelist is calculated and recorded by the Memory Access Controller during each snapshot.

Memory Manager nodes visited. In the process of scanning nodes for collectibility, the Memory Manager examines nodes from two origins. The nodes from the FIFO are in Closed Context Frames. The nodes from the Garbage Can are in a local queue[1]. Each of these lists is actually treated as a list of roots of subgraphs. Thus, there are four "types" of nodes the Memory Manager may inspect:

1) context frame (roots)

2) context frame (non-root nodes)

3) garbage can queue (roots)

4) garbage can queue (non-root nodes)

[1] In the simulation, this queue is called Candidate Queue.

The relationship between root nodes and non-root nodes is displayed in Figure 7.1

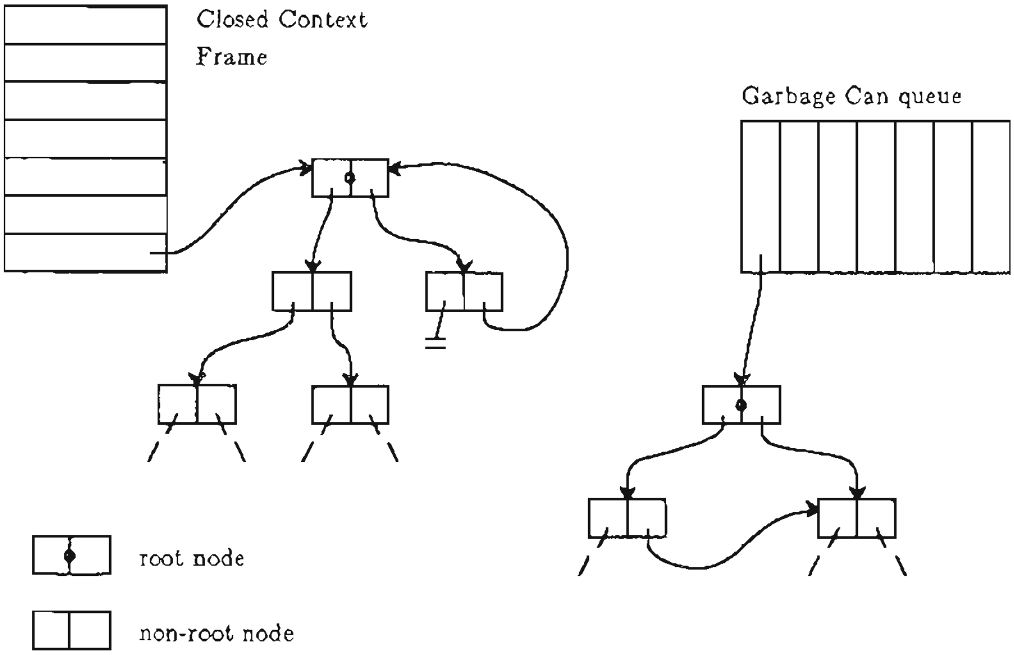


Figure 7.1. Relationship of root and non-root nodes.

This breakdown is made for two reasons. One, it allows us to compare the relative efficiencies of the memory management system: how much effort is spent collecting temporary allocation frame nodes compared to collecting subgraphs that have had references deleted? Two, it distinguishes between the roots inspected and the descendents of those roots. This provides values from which a backlog estimate can be made (see "backlog" below). Each of these categories is accumulated throughout the simulation and recorded during each snapshot.

Memory Manager nodes collected. A total of nodes actually deallocated is maintained throughout the simulation and recorded during each snapshot. As with nodes inspected, two categories are kept to distinguish between the nodes from context frames and the nodes from the Garbage Can.

Memory Manager backlog. An estimation of the number of nodes waiting to be scanned for collection and that are in fact collectible is made during each snapshot. This figure represents latent freelist nodes – that is, nodes that are no longer used but have not yet been returned to the freelist. The value is calculated by empirical evidence. The Memory Manager keeps track of the number of nodes actually deallocated and the number of roots and root-descendents inspected. The number of roots and root-descendents already inspected can be used, along with the actual number of roots still to be inspected, to estimate the total number of nodes yet to be inspected. Once the Memory Manager has deallocated some nodes, it can use the relationship between the nodes deallocated and total nodes inspected to estimate the number of nodes which await inspection and are actually collectible.

Mutator model evaluation. A total of the number of times each Mutator operation was performed is maintained throughout the simulation and recorded during each snapshot. These values allow us to examine the nature of the Mutator model and the apparent distribution of the operations performed.

7.2. Summary of results.

The following graphs and data display specific characteristics of the simulated memory management system. These graphs include examples of the system just keeping up with allocations, the system failing to keep up, and comparative collection costs between context frames and garbage can nodes. All graphs were generated from averages of multiple simulation runs with the same parameters. The fundamental characteristic of the system was that it is "steady state". That is, the demand for new nodes and the creation of garbage are balanced.

To simulate a steady state system, some parameters are dynamically adjusted until the specified memory residency is reached. In particular, the allocation rate is altered early in a run to speed the convergence to the specified state. This period is noticeable in most of the graphs as an initial interval of rapid change. For the graphs given, approximate residency is reached shortly after 400 snapshots; actual equilibrium at the specified residency is typically reached by 700 snapshots. In all graphs, the x-axis represents cumulative snapshots.

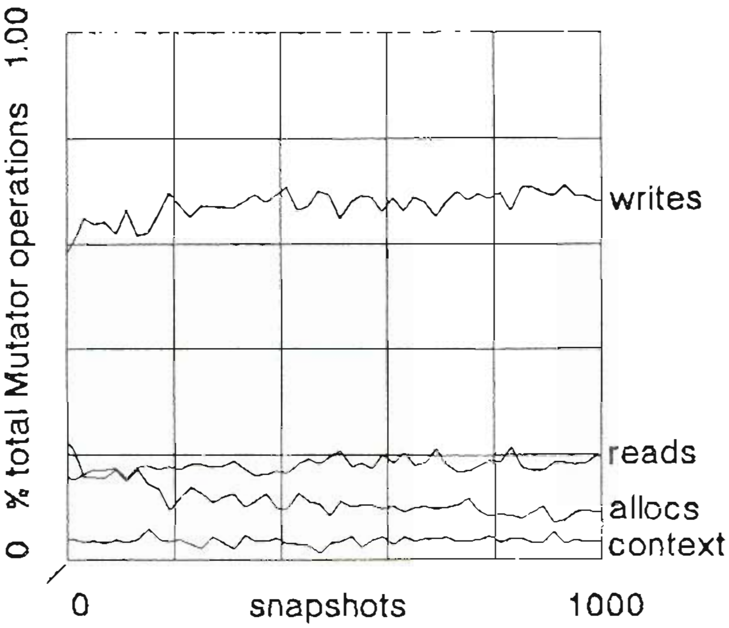


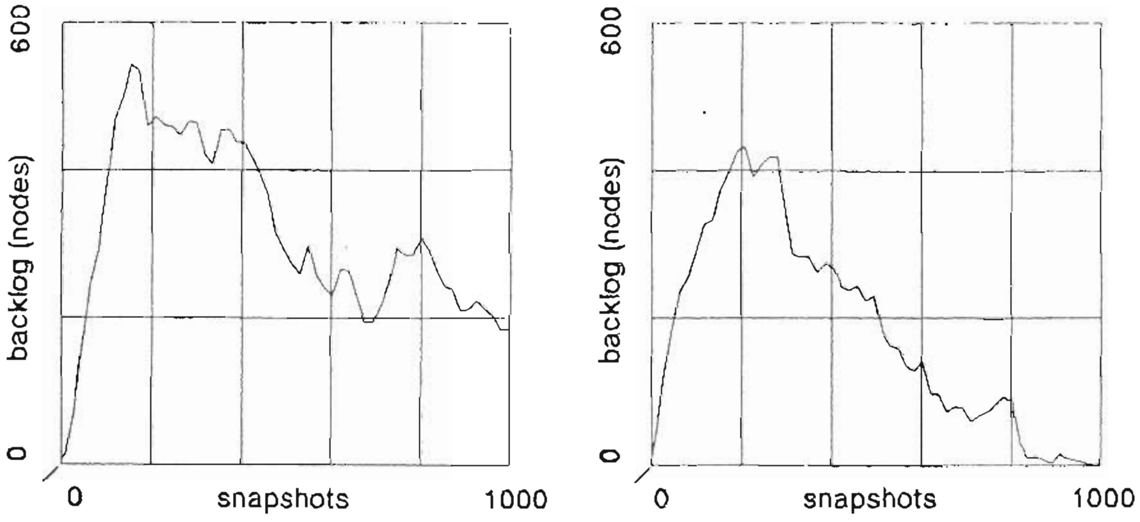
Figure 7.2. Distributions of mutator operations.

7.2.1. Mutator model.

The graph in Figure 7.2 shows the relationship of a set of mutator operations. Each operation is expressed as a percentage of total mutator operations for a given period between snapshots. Notice that graph writes comprise a significant majority of the operations. Most of this imbalance is attributed to the method of selecting Mutator operations, and thus, is an artifact of the simulation. This graph displays the memory-intensive duty cycle maintained by the Mutator.

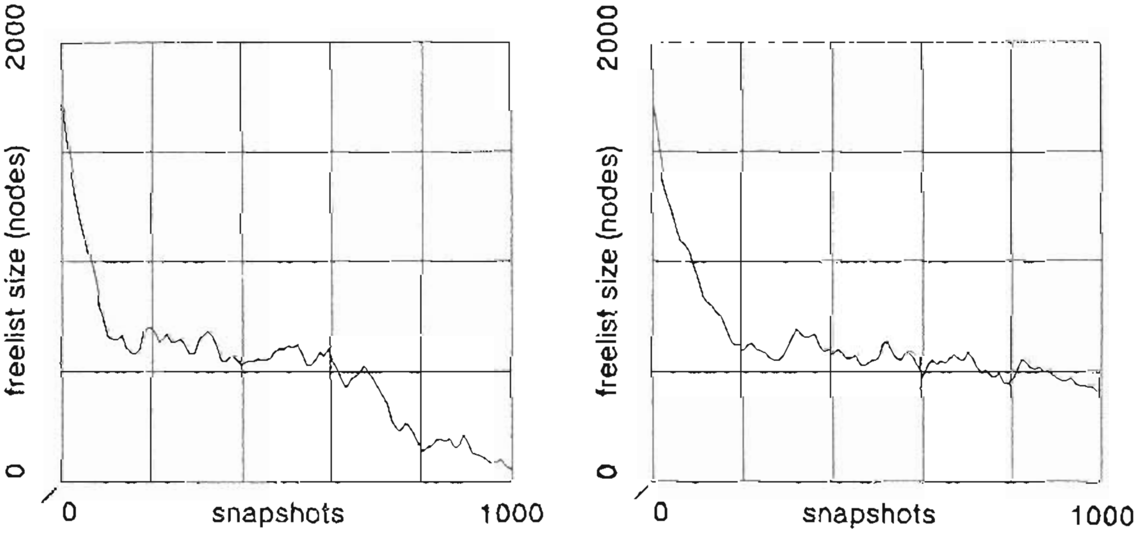
7.2.2. Memory manager capacity.

The ability of the memory management system to keep up can be shown with two sets of data. Given a fixed allocation rate, when the freelist size does not go to zero and the Memory Manager backlog is near zero, the management system is keeping up. However, if the backlog is somewhat more than zero *and* the freelist size drops to zero, this situation is an indication that the Memory Manager is saturated. Figures 7.3a and 7.4a together show the inability to maintain the desired residency at an allocation rate of 1/11. Figures 7.3b and 7.4b show the ability to just maintain the desired residency at an allocation rate of 1/12.



(a) (b)

Figure 7.3. Freelist size.



(a) (b)

Figure 7.4. Memory Manager backlog.

7.2.3. Expense of collecting long- and short-lived nodes.

The graph in Figure 7.5 shows the comparative effort spent by the Memory Manager for quantities of persistent versus short-lived nodes. The values represent the expense of scanning both root and non-root nodes from allocation Context Frames and from the Garbage Can. CRE and CNE represent the Context frame Roots Examined and Context frame Non-roots Examined, respectively. Likewise, GRE and GNE represent Garbage Can Roots Examined and Garbage Can Non-roots Examined.

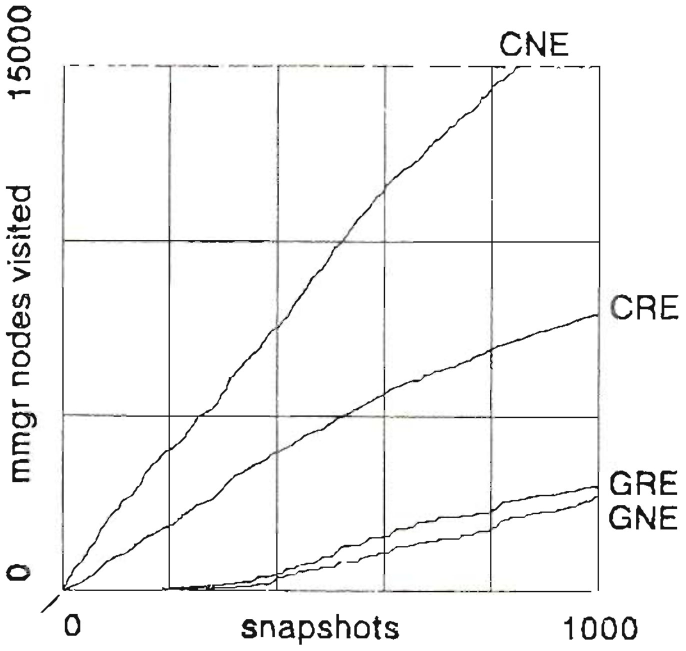


Figure 7.5. Collection expense by node source.

7.2.4. Memory manager effectiveness.

In Figure 7.6, the percentage of nodes actually collected in terms of the nodes inspected shows the amount of "wasted" effort. The upper line represents effectiveness of inspecting nodes from the Garbage Can. The lower line represents inspection of nodes from the Context Frames. This graph shows that approximately

40 percent of the nodes from the Garbage Can inspected were, in fact, collectible, while only about 20 percent of the nodes from the Context Frames were actually collectible.

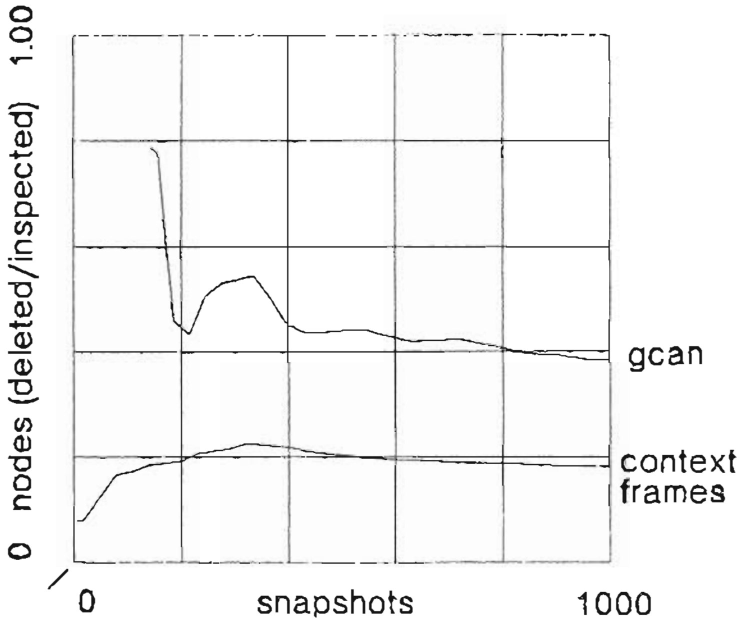


Figure 7.6. Memory Manager effectiveness.

8. Conclusion.

The design and simulation of a dynamic list-structure memory has been described. A concurrent algorithm and an architecture-level simulator have been developed. This simulator was used to demonstrate the feasibility and effectiveness of both the proposed architecture and the concurrent algorithm.

We have examined a variety of memory management approaches. These include popular techniques such as reference counting and mark-sweep collection. Previously established techniques either require severe synchronization constraints in a parallel application or are not sufficiently powerful for graph-reduction engines.

A concurrent adaptation of a new modified reference counting technique has been shown to perform effectively, given direct "hardware" support of particular operations. Traditional performance penalties of in-line reference counting are removed by an intelligent memory controller. The ability to collect cyclic structures is provided by subgraph traversal of potential cycles.

The simulation results pinpoint a lower bound on the performance limits of our implementation. We have shown that allocation of 1 node every 12 operations is achievable, while 1 node every 11 operations is not. This statistic compares favorably with the high demands of present systems. For example, Berkeley Smalltalk averages 1 object creation per 80 instructions |Ung84, UBF84|.

It is important to point out that the mean object size for Berkeley Smalltalk is 70 bytes, while the node size used in our system is just over 11 bytes. For comparison, this means Berkeley Smalltalk places demands for new memory equivalent to about 1 node every 12.5 operations. Admittedly, this is a rather coarse comparison, since a bytecode is not the same as a Mutator operation; they are considered to be roughly equivalent for this comparison, however.

8.1. Suggestions for additional work.

There are several areas of the architecture that were not evaluated. The Memory Access Controller may be a bottleneck if not carefully designed to provide high-speed dual-port access. Analysis of memory contention in the developed system model would be useful to define the performance limits and to suggest a likely dual-port mechanism. Furthermore, the details of the Memory Access Controller architecture have not been defined.

In the simulation, the Memory Access Controller was assumed to contain embedded logic to make reference counting atomic. It may be possible to avoid the added complexity by requiring the host (Memory Manager) to maintain the reference counts. This adjustment may make particular sense, as it is really only the Memory Manager that must resolve the reference counts to determine collectibility. Some type of sequential (reference count) transaction file, similar to the approach in [DeB76] may be sufficient to determine reference counts.

Section 4 introduced some adaptations to the concurrent algorithm for a graph-reduction engine such as the G-Machine. An additional improvement that has been suggested by Kieburtz [Kie85a, section 2.1, and ,Kie85b] takes advantage of sophisticated compiler technology. In this technique, a threshold field is associated with each node. The threshold is used to reduce the number of references passed to the Garbage can. The mechanism relies on the compiler successfully identifying all node allocations that may be roots of cyclic graphs, and thereby, bounding the maximum number of cycles (threshold) that could close a node. When a node is allocated, it is assigned a threshold value. When the reference count of a node is decremented, it is considered for placement in the Garbage Can only if its reference count is less than or equal to the threshold. This strategy is claimed to decrease nearly to zero the probability that a node is put in the Garbage Can when it cannot be collected. Adding this enhancement to the simulation would necessitate a somewhat more sophisticated mechanism for graph manipulation than the pseudo-random operations of the Mutator.

The allocation scheme used for the simulation does not provide a mechanism for memory compaction. When nodes are re-linked into the freelist, no address reordering occurs. As a result, we can expect the arrangement of addresses in the freelist to become random. This action has the bad effect that locality of reference is not maintained as new nodes are allocated from the freelist. One type of remedy involves the use of "buddy system" allocation [Knu75]. Kieburtz [Kie85a] and Baker, et al., [BCW85] have proposed variants of the buddy system that are designed to reduce fragmentation associated with standard freelist allocation. These strategies organize memory as a tree of blocks. Allocations and deallocations are localized within a range of blocks, and thus, reduce non-local references due to the allocation technique.

The present design has only considered a dual-processor system. Given the commercial availability of general-purpose multi-cpu machines, it is practical to consider a system based on many processors. One approach would consider memory as "distributed". Store would be logically partitioned. Each partition would be assigned to a given memory manager; a memory manager could include the combination of the duties of both the Memory Access Controller and the Memory Manager as described in this thesis. Alternatively, the MAC and Memory Manager could be separated, if some degree of custom hardware support for the MAC were available. Nodes could be allocated to a requesting "mutator" by any memory manager. The synchronization constraints provided by the "mutator-visit" bit would continue to function properly. A major stumbling block in this approach results from the mechanism used to handle temporaries. Separate context frames would need to be kept for each of the mutators serviced by a given memory manager. A protocol would need to established to determine when a given mutator changed context.

Further analysis of both the architecture and the presented algorithm can be performed using the simulator. Elementary use of the simulator is described in Appendix B. Some analysis can be done by simply adjusting run-time parameters. The modified buddy allocation system should be added and, once refined, additional support for multiple processors should be investigated.

References

[BCW85] Baker, B. S., Coffman, E. G. and Willard, D. E., Algorithms for Resolving Conflicts in Dynamic Storage Allocation, *J. ACM 32*, 2 (Apr. 1985), 327-343.

[Bak78] Baker, Jr., H. G., List processing in real time on a serial computer, *Comm. ACM 21*, 4 (Apr. 1978), 280-294.

[Coh81] Cohen, J., Garbage Collection of Linked Data Structures, *Computing Surveys 13*, 3 (Sep. 1981), 341-367.

[Deb84] Deb, A., An Efficient Garbage Collector for Graph Machines, Tech. Rep. CS/E-84-003, Oregon Graduate Center, Beaverton, OR, 1984.

[DeB76] Deutsch, L. P. and Bobrow, D. G., An Efficient, Incremental, Automatic Garbage Collector, *Comm. ACM 19*, 9 (Sep. 1976), 522-526.

[DLM78] Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S. and Steffens, E. F. M., On-the-Fly Garbage Collection: An Exercise in Cooperation, *Comm. ACM 21*, 11 (Nov. 1978), 966-975.

[GoR83] Goldberg, A. and Robson, D., *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, May 1983.

[GKM82] Graham, S. L., Kessler, P. B. and McKusick, M. K., gprof: a Call Graph Execution Profiler, *Proceedings, ACM SIGPLAN Notices '82 Symposium on Compiler Construction*, June 1982.

[HiC84] Hickey, T. and Cohen, J., Performance Analysis of On-the-Fly Garbage Collection, *Comm. ACM 27*, 11 (Nov. 1984), 1143-1154.

[JoA83] Johnsson, T. and Augustsson, L., *The G-Machine -- an abstract architecture for graph-reduction*, Department of Computer Sciences, Chalmers Univ. of Technology, Gothenburg, 1983.

[KaK83] Kaehler, T. and Krasner, G., LOOM -- Large Object-Oriented Memory for Smalltalk-80 Systems, in *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner (ed.), Addison-Wesley / Xerox , 1983, 251-270.

[Kie85a] Kieburtz, R. B., Incremental memory management for dynamic, list-structure memories, Tech. Rep. CS/E-85-008, Oregon Graduate Center, Beaverton, OR, Mar. 1985.

[Kie85b] Kieburtz, R. B., When chasing your tail saves time, research notes, Oregon Graduate Center, Beaverton, OR, 1985.

[Kie85c] Kieburtz, R. B., The G-Machine: A fast, graph-reduction evaluator, research notes, Oregon Graduate Center, Beaverton, OR, Jan. 1985.

[Knu75] Knuth, D. A., *The Art of Computer Programming, Vol. 1*, Addison-Wesley, Reading, MA, second edition 1975.

[LiH83] Lieberman, H. and Hewitt, C., A Real Time Garbage Collector Based on the Lifetimes of Objects, *Comm. ACM 26*, 6 (June 1983), 419-429.

[McC60] McCarthy, J., Recursive functions of symbolic expressions and their computation by machine, *Comm. ACM 3*, 4 (Apr. 1960), 184-195.

[Moo74] Moon, D. A., *MACLisp reference manual*, Project MAC, M.I.T., Cambridge, MA, Apr. 1974.

[Sar84] Sarangi, A. G., Simulation and Performance Evaluation of a Graph Reduction Machine Architecture, Master's Thesis, Oregon Graduate Center, Beaverton, OR, July 1984.

[Ste75] Steele, Jr., G. L., Multiprocessing Compactifying Garbage Collection, *Comm. ACM 18*, 9 (Sep. 1975), 495-508.

[Tei78] Teitelman, W., *Interlisp Reference Manual,* Xerox Corp. and Bolt, Beranek & Newman, third edition 1978.

[Ung84] Ungar, D., Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm, *Proceedings of Symposium on Practical Software Development Environments*, Apr. 1984, p. 157.

[UBF84] Ungar, D., Blau, R., Foley, P., Samples, D. and Patterson, D., Architecture of SOAR: Smalltalk on a RISC, *Proceedings, 11th Annual International Symposium on Computer Architecture*, June 1984, 188-197.

[Wad76] Wadler, P. L., Analysis of an Algorithm for Real Time Garbage Collection, *Comm. ACM 19*, 9 (Sep. 1976), 491-500.

[WaP77] Warren, D. H. and Pereira, L. M., Prolog -- The Language and Its Implementation Compared to Lisp, *Proceedings of Symposium on Artificial Intelligence and Programming Languages*, 1977.

[Wis85] Wise, D. S., Design for a Multiprocessing Heap with On-board Reference Counting, Tech. Rep. 163-Revised, Indiana University, Bloomington, IN, July 1985.

Appendix A: Control and communication between components.

This appendix provides a detailed description of the signals and data passed between components of the list-structure memory. Refer to Section 5 for a general functional description. The reader may find it useful to refer to Figure 5.1 while reading this appendix.

A.1. Signal Interfaces.

Signals are implemented as a state ("line high" or line low") or as a strobe (in'errupt). The present definition of interrupts use a single level of priority. Table A.1 describes the senders, receivers, and operations of each signal.

Table A.1. Signal Functions

Signals	Sender	Receiver(s)	Function and Results
ALLOC	Mutator	FIFO, MMGR	"Allocate" a new node. Address of an initialized node (preallocated from unused node pool) is presented to Mutator and Memory Manager. The Mutator uses the node as desired (purpose of request), the Memory Manager adds the node to its current Active Stack Nodes Set. If the node is not immediately available, the Mutator blocks until it becomes available. Only once the Mutator receives the node is the ALLOC signal is transmitted to the Memory Manager.

CALL	Mutator	MMGR	Signals entry into a new context. A new Active Context Frame (ACF) is created, and is pushed onto the Active Alloc'd Nodes Stack (AANS).
FEMPTY	FIFO	MAC	FIFO underflow: a request for a node has come in and the queue is empty. This signal is a priority request to the MAC. The unused node pool is preallocated by the MAC until a READ/WRITE request from the Mutator (or Memory Manager) or FFULL response from FIFO.
FFULL	FIFO	MAC	If the FIFO becomes full, it signals the MAC to defer further preallocations. The MAC will not preallocate until $\overline{\text{FFULL}}$.
FIFO_ENQ	MAC	FIFO	When the MAC preallocates nodes, it sends the address to the FIFO. This action effectively removes the node from the freelist, but makes it available for an immediate ALLOC from the FIFO.

GCAN_DEQ	MMGR	GCAN	The Memory Manager signals the Garbage Can when it wants another candidate node. The Garbage Can dequeues the head of its local list and sends it to the Memory Manager.
GCAN_ENQ	MAC	GCAN	The MAC indicates to the Garbage Can when there is a candidate node available. The Garbage Can enqueues the address for a (subsequent) GCAN_DEQ request.
GFULL	GCAN	MMGR, MAC	Garbage Can full: instructs the Memory Manager to spend as much resource as necessary emptying the Garbage Can. Indicates to MAC that it must block on the next write to the Garbage Can until the condition is relieved.
RETURN	Mutator	MMGR	Signals return the current context. All nodes in the current ACF are included in a list of "nodes to be checked for collection", the AANS is popped.

A.2. Communication Interfaces.

Channels of communication are needed to pass directives, addresses, status, and data values between the functional units.

Descriptions of the interprocess communications follow. "Action name" may be (i) a DIRECTIVE or (ii) a (SIGNAL). A directive may be thought of as an instruction. Directives are passed in parallel with data. Signals are as described earlier, and are included here since they result in some data transfer activity.

Processor path
Action name **Description**

Mutator⟷MAC

READ Mutator sends READ + Address + Cell to MAC. Node and tag data for the specified cell are sent from the MAC to the Mutator. The Mutator suspends until the data is returned.

WRITE Mutator sends WRITE + Address + Cell, and Tags + Data to MAC. If tags indicate Data is a pointer, the MAC increments destination's reference count sets its Mutator-visit bit. The Mutator-visit bit of Address is also set.

TRASH Mutator sends TRASH + Address to MAC. The MAC decrements the reference count of the nodes currently pointed to by the node at Address (possibly adding those nodes to the garbage can). The Mutator-visit bit is set for the de-referenced nodes and for Address. An immediately following pair of WRITE directives for the node's cells is expected.

Mutator⟷FIFO

(ALLOC) The Mutator receives a node address from the FIFO as result of the ALLOC signal. This address is simultaneously sent by the FIFO to the Memory Manager. The node is used for subsequent WRITE/UPDATE operations.

MMGR⟷MAC

CREAD The Memory Manager sends CREAD + Address + Cell to MAC. Node data and tags for the cell are sent from MAC to Memory Manager. The Memory Manager suspends until the data is returned.

DEALLOC Memory Manager sends DEALLOC + Address to MAC. The node specified by Address is appended to the freelist by MAC. Reference counts are decremented for nodes pointed to by this node.

CWRITE Functions similarly to Mutator WRITE, however, the cell data cannot be altered. Memory Manager sends CWRITE + Address and Tags to MAC. The Mutator-visit bit may be reset, and the Collector local-reference count may be altered.

MMGR⟷FIFO

(ALLOC) Memory Manager receives a node address from FIFO as result of ALLOC signal. The address is stored in the current Active Alloc'd Nodes Set.

MMGR⟷GCAN

GCAN_DEQ The Memory Manager sends the Garbage Can the GCAN_DEQ directive, indicating the head of the Garbage Can is to be passed on. The Memory Manager copies the address into local storage. This address is a pointer to node which is a candidate for collection.

MAC⟷FIFO

(FEMPTY) If the FIFO underflows (receives ALLOC when empty), the MAC receives an FEMPTY signal. The MAC then tries to fill the FIFO until another (higher priority) service is requested.

FIFO_ENQ MAC sends preallocated node addresses to FIFO (nodes are initialized). The FIFO status is either FIFOFULL or NOT_FIFOFULL. If the FIFO is full, the MAC leaves the node at the head of the freelist, and retries later.

MAC←→GCAN

GCAN_ENQ MAC sends potentially collectible node addresses to the Garbage Can. The Garbage Can status is either GFULL or NOT_GFULL. If GFULL, the MAC suspends on a subsequent GCAN_ENQ until the status returns to NOT_GFULL.

Appendix B: Usage of the simulator.

This appendix gives a terse discussion of the filesystem organization and usage of the list-structure memory simulator. The reader is expected to be familiar with UNIX 4.2bsd and its utilities.

Directory structure.

The simulation directory is in an expiremental arrangement. The source code, executables, and special utilities all share a directory tree with the control and data files.

The "root" or "home" of the simulation directory contains the C source code and Makefile necessary to reconstruct the executables. This directory also contains the executables that comprise the simulator. Each executable corresponds to one of the processes described in Section 6.

fifo The FIFO process. Provides preallocated node addresses from the MAC to the Mutator.

gcan The Garbage Can process. Buffers node addresses between the MAC and the Memory Manager.

mac The Memory Access Controller. Provides dual-port access to the main graph store. Contains the graph store, itself.

mmgr The Memory Manager and Collector. Contains the parallel, incremental garbage collection algorithm. Provides collection of allocation frames.

mutator The Mutator walks the graph in the gstore, performing graph-modifying operations. A Mutator operation may be one of ALLOC, READ, WRITE, TRASH, CALL, or RETURN.

parent The Parent process is used to get all the other executables started, and to control the snapshot period.

In addition to the source and executables, there are several subdirectories.

ctl The control directory is used frequently during the simulation. Zero-length control files are placed here. The AF_UNIX domain sockets appear here, as a side effect of the filesystem. While no data is actually stored, inodes are allocated. In addition to the sockets, several synchronization or "lockout" files are created during initialization, during snapshots, and upon termination.

data.in The parameter files that are used to control elements of the simulation are kept here. There are three files: *graph.style*, *mutator.params*, and *parent.params*. Each of these contains a description of, and the actual values used for the parameters.

data.out The snapshot files, and the summary output files are created here automatically during the simulation. Each process is given a unique process id. The process name and its id are used to create a unique name for the output file. The simulator also generates output on the stderr and stdout file descriptors; it may be useful to redirect these into a file in the data.out directory.

utils This directory contains source code and scripts useful to both the creation of executables and the processing of raw data. See "Utilities" below for more details.

Generation of data.

To generate additional data, the user needs to set the desired parameters in each of the "data.in" parameter files. If profiling is desired, the macro MONFLAG in the Makefile should be set to "-pg", and the executables recreated. The user must be in the "home" directory of the simulation, since it relies upon the arrangement of the various subdirectories during execution. It is then simply a matter of giving the UNIX command:

```
% parent
```

The parent process begins executing, initializes the environment for each of the other processes, then starts them executing.

Several caveats are in order if modifications are made. It is possible that certain sets of parameters will yield undesirable results. Because of the mechanism used to converge to equilibrium quickly, it is fairly easy to saturate the Memory Manager. Careful adjustment of the allocation rate, the context switching frequency and the context depth must be used. Furthermore, the signal mechanisms and the ipc mechanisms of UNIX 4.2bsd are rather fragile; this is especially the case with processes that may receive a rapid sequence of identical signals. This problem can cause a simulation that "has always worked before" to throw up its hands and terminate. Finally, a close eye must be kept during the execution of a simulation. If a run-time error occurs in one of the children processes (those spawned by the Parent), and it is not explicitly handled in the code, the process may inexplicably die, without announcement and without being detected until the end of a snapshot.

Utilities.

Once a simulation has completed, the raw data must be reduced to a palatable form. Several utilities, which are setup to be easily customized for each set of data have been created. A general utility "smooth" takes a list of data, and smooths the noise by averaging across a window of data. More particular utilities are prefixed with "plotcmd." These various shell scripts extract fields from particular data files, pass them through graph(1), and then to plot(1) for display.

If profiling is being used, it is important to ensure different "gmon.out" files are produced for each executable, and to retain the executables used to generate the *.out files.

The creation of a *.out file specific to the executable is provided by "monfilter". This program is a specialized binary editor that replaces occurrences of "gmon.out" with the first four letters of the executable file name followed by ".out" (it *could* have been done more easily, but this was a good solution at the time!). The Makefile automatically takes care of using monfilter. If profiling has not been specified, monfilter has no effect on the object code.

The *.out files that result from a simulation with profiling should be moved to the data.out directory. Copies of the executables used to generate the *.out files should also be put in the data.out directory. This approach allows complete examination of the data from a given simulation following subsequent simulations/changes. The gmon(1) utility can be used to examine the profiling characteristics of the system.

Biographical Note

The author was born 5 May 1961, in LaGrande, Oregon. In 1968, he moved to Madras, Oregon and graduated from Madras Senior High School in 1979. He entered Willamette University with Honors and received a Bachelor of Arts degree in Computer Science/Mathematics May 1983.

During his first two years at Willamette University, the author worked as an Assistant System Manager for the Willamette University Computing Center. During his second two years, and for the summer following graduation, the author worked as a Software/Hardware Consultant for Northwest Computer Communications Associates, Inc.

The author began study at the Oregon Graduate Center in September 1983. During the summer of 1984, the author participated in a Summer Student/Intern program at the Computer Research Lab of Tektronix. The author also helped teach the VLSI Design course at OGC during the fall quarter of 1984.

The author completed the requirements for the degree Master of Science in September 1985. In October 1985, he accepted a position at the Graduate Center as a Software Systems Engineer in the Computer Science and Engineering Department.