A DUAL-PORTED REAL MEMORY ARCHITECTURE FOR THE G-MACHINE

Linda J. Rankin
B.S., Lewis & Clark College, 1976

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

August, 1986

The thesis "A Dual-Ported Real Memory Architecture for the G-machine" by Linda J. Rankin 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

Dan Hammerstrom
Associate Professor,
Department of Computer Science and Engineering

Bill Bain
Intel Corporation

Peter Borgwardt
Tektronix, Inc.

ACKNOWLEDGEMENTS

I would like express my gratitude to Dr. Richard B. Kieburtz for providing me with the opportunity to participate in the G-machine research project. It was an opportunity to learn, and thus a rewarding experience. It was also his creative approach to computer design which provided much of the foundation for this work. His open door policy was always appreciated.

I wish to thank the members of my committee for their efforts, especially Dr. Dan Hammerstrom, a convincing and inspiring teacher. I gratefully acknowledge the help of Bill Bain and the opportunity to use his **Teamwork** subroutine library. Credit is also due to Mark Foster for his initial work on the algorithm and memory architecture.

Most of all, I would like to thank my husband, Larry, for his encouragement and moral support throughout my graduate studies.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

A Dual-Ported Real Memory Architecture
for the G-machine

Linda J. Rankin
Oregon Graduate Center, 1986

Supervising Professor: Richard B. Kieburtz

A dual-ported real memory architecture is described which supports the requirements of a list-processing evaluator, the G-machine. The architecture provides support for allocating available nodes and a concurrent garbage collection scheme. This scheme uses reference counts and requires traversal of sub-graphs to collect cyclic structures. The architecture requires only one customized hardware component that provides support for maintaining reference counts. Simulation of the architecture shows that it is efficient and meets the requirements of the G-machine given certain assumptions about the number and size of sub-graphs that are traversed. Cyclic structure information provided by the compiler would reduce the number of sub-graphs requiring traversal. Simulation shows that this optimization improves performance of the design, particularly for allocation rates greater than 100K nodes per second.

INTRODUCTION

List-structured memory is considered to be an architectural primitive of list processors. Memory is viewed by the processor as an infinite heap of binary nodes. When new objects are created, memory must be allocated to contain their representation. When an object is no longer in use, garbage collection is the process by which unused storage space is reclaimed. The memory system must, therefore, have a mechanism to allocate new nodes and be able to reclaim storage at the same rate at which new objects are created.

The performance of a memory management scheme directly affects the performance of a list-processor. Early experience with large LISP programs indicated that substantial execution time - 10 to 40 percent - was spent in garbage collection [Ste75,Wad76]. Since that time, some performance gains have been realized through the development of more efficient garbage collection algorithms[Coh81]. For example, an algorithm developed by [Lie83] concentrates on those objects that are the most likely to be collected.

As list-processing systems have evolved, additional performance gains have been made by supplementing garbage collection either by microcode or direct hardware support. Adding microcode support for free variable lookup in the DoradoLisp system resulted in a speedup factor between two and four [Bur80]. More recently, two commercial LISP machines, one by Texas Instruments and the other by

Symbolics, have included in their design direct hardware support for garbage collection. Both of these machines use a modified version of Baker's classical copying collector algorithm [Bak78]. The Symbolics design includes hardware support for a barrier which lies between the evaluator and memory. This barrier prevents the uncontrolled propagation of references to objects in the old space and assists in identifying objects according to their lifetime. Implementing this barrier in hardware has a dramatic effect on performance [Moo84].

Designing memories for list-processing systems, therefore, has become a synthesizing process where the memory requirements of the garbage collection algorithm and the processing requirements of the evaluator define the architecture. This thesis presents an architecture that was created in such a manner. The memory is designed to meet the requirements of the G-machine, a graph-reduction evaluator that uses a dynamic list-structured memory. The garbage collection algorithm is a modified reference counting scheme developed at Oregon Graduate Center [Deb84, Kie86, Fos85] that provides for concurrent garbage collection using a separate processor. The memory architecture is a dual-ported memory with hardware support for communication between the G-machine and the collector, reference counting, and the allocation of free nodes. Perhaps the most exciting aspect of the design is its potential for real-time list-processing.

The remainder of this thesis describes the memory requirements of the G-machine and the garbage collection algorithm which, in turn, define the functionality of the memory architecture. A microarchitecture is then presented for which a prototype design is specified. Commercially available components have been used in the design wherever possible, and thus only one customized component is required to build the memory system. This prototype design provides timing information so the design could be simulated to obtain performance information. Simulation results are presented and analysis of the design is conducted with respect to critical performance issues. Finally, suggestions are made for further research and improvements in the design.

THE G-MACHINE

The G-machine architecture was originally defined by Johnsson [Joh83] as an evaluation model for an ML compiler. A sequential evaluator has been developed at Oregon Graduate Center (OGC) [Kie85A] based on that abstract model. This evaluator, which will be referred to as the G-machine, performs graph reduction where expressions are represented as graphs rather than strings. The memory architecture and management scheme has been designed to meet the requirements of the G-machine. This section describes the memory requirements of the G-machine and those aspects of the G-machine which affect the implementation of the garbage collection algorithm.

Memory Requirements

Dynamic list-structured memory is considered to be an architectural primitive for list processors, including the G-machine. Memory is list-structured in that it is made up of nodes which consist of a pair of individually accessible data elements that could be basic values or pointers. The G-machine views memory as an infinite heap where new nodes are allocated upon request. The G-machine has an instruction *alloc* which is a primitive instruction to memory as is read or write. The G-machine uses alloc to request a memory address in which it will store the representation of an object. When that object is no longer is use, garbage collection is required to reclaim the storage space so it can be used again. Thus, the illusion of an infinite heap.

The G-machine also requires a tagged memory. Each data field of a graph node includes a tag bit. This tag, called the *is_pointer* tag, is used by the G-machine to identify if the data is a pointer or a basic value. The graph node itself also requires a tag called the *is_evaluated* bit. The G-machine uses this tag to distinguish whether or not the node has been evaluated by a previous reference.

The G-machine stores all expressions and data in a graph, no static structures such as arrays are used. Therefore, the demand for free nodes is expected to be greater than that of conventional applicative language processors. The projected allocation rate obtained from a static analysis of G-machine microsequences is between 75K and 125K nodes per second. This is an order of magnitude greater than the allocation rate of the benchmark programs used to test the efficiency of the memory management scheme on the Symbolics machine [Moo84]. In order to support this high allocation rate without significantly hindering the evaluation process, a concurrent memory management scheme may be a necessity.

In addition to supporting a high allocation rate, the garbage collection algorithm must be able to detect cyclic structures. The G-machine does not form exclusively acyclic graphs; recursive functions may exhibit one or more cycles during reduction. However, the incidence of cyclic graphs is expected to be less than acyclic graphs.

In summary, the memory requirements of the G-machine are as follows: 1) memory must be list-structured, 2) memory must be able to provide the address of an available node upon request, 3) the memory architecture must provide for tags, 4) garbage collection must be efficient, and perhaps a concurrent process, and 5) the garbage collection algorithm must be able to collect cyclic structures.

Memory Management

The garbage collection algorithm used in this design is a modified reference counting scheme. This scheme requires a count of the number of references to a node to be maintained. In the classical reference counting scheme, the *reference count* is used to determine the collectibility of a node. When the reference count is zero, then it is assumed that there can be no external references to a node and it is collectible. To maintain the reference count a complex operation on a write to memory is required:

(1) Check whether the previous contents of the node was a pointer, and if so, decrement the reference count of the node to which it points.

(2) Check whether the datum being written is a pointer, and if so, increment the reference count of the node to which it points.

There are two aspects of the G-machine which affect the manner in which a reference counting scheme is implemented. The first is related to how an expression is evaluated, and the second is due to the way the G-machine architecture provides for an internal representation of the expression graph.

The G-machine evaluates applicative expressions, i.e. applications of functions to argument expressions. The expression is represented by a graph in memory, and during the process of evaluation, is mutated by a series of reduction steps until it reaches a normal form. Graph reduction is accomplished through the manipulation of a traversal stack that contains pointers into memory. A possible configuration of the traversal stack is shown in Figure 1a.

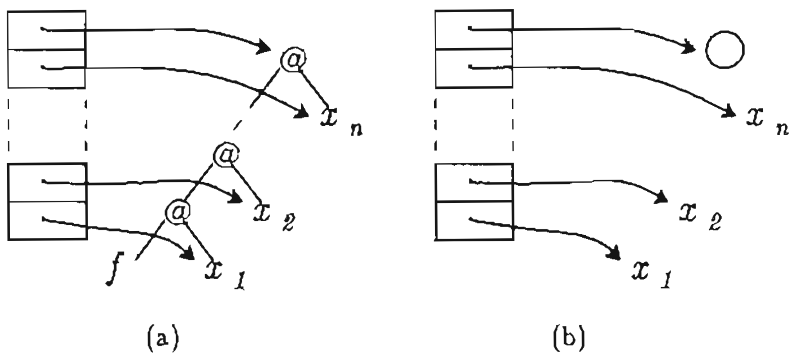

Figure 1. Traversal stacks.

The traversal stack contains pointers directly to the argument expressions and to the principal application that is being reduced. To reduce the expression, a program compiled for the function *f* is executed. After reduction, the principal application node is overwritten with the representation of its value (figure 1b).

The instruction set of the G-machine includes a subset of *update* instructions which are used to overwrite the contents of a graph node. Only on the update of a graph node are the contents of a node overwritten. This fact can be exploited to simplify the number of operations required to maintain the reference count of a node on a write to memory. A new instruction *trash* is added to indicate that the contents of a node are to be overwritten. Therefore, on a full node update, the following sequence of instructions would be observed:

```
trash address1
write address1, datum1
trash address2
write address2, datum2
```

Trash is used to indicate in the above example that the contents of address1 will be overwritten. If the contents of address1 is a pointer, then the reference count of the node to which it points will be decremented. The only operation that is now required on the write is to test if the datum is a pointer, and if so, increment the reference count of the node to which it points.

The method of graph reduction used by the G-machine allows the addition of the primitive instruction trash which simplifies the maintenance of reference counts. On the other hand, implementation of a reference counting scheme is complicated by the fact that the G-machine's architecture provides for an internal stack which holds pointers to the graph in memory. During evaluation of an expression, the traversal stack can copy or destroy graph pointers without mutating the representation in memory. These operations are executed in a single cycle and maintaining reference counts for these internal references would be costly. Instead, a stack allocation scheme called *stack allocation with persistence* [Kie86] is used.

During a function call, many nodes are allocated to the G-machine. Some of these nodes are short-lived and never connected to the rest of the active expression graph. Upon a return from a function call these nodes are no longer used and should be collected. Other nodes that are allocated during a call are written into the graph to represent the value of the function application. These nodes (persistent nodes) persist after a return from a call and are not collected. All nodes that are allocated during a function call will be called *temporary* nodes because they are considered to be temporary until they are identified as *persistent*.

The stack allocation scheme is used so that references from the internal stack need not be counted. Nodes allocated during a function call are placed in a stack. Upon a return, each node in the stack that was allocated during the call is examined for collectibility. A stack discipline can be used for deallocation because the G-machine meets three conditions:

(1) The G-machine has only local control transfers, contexts are saved and restored according to a LIFO discipline.

(2) No component of the state of the G-machine persists after completing evaluation of a function application.

(3) No node allocated during evaluation of an application is accessible after return from the function unless it is reachable in memory from the root of the result expression.

In order to implement stack allocation with persistence an additional tag bit is added to the graph node. This tag, called the *persistent_bit*, is used to distinguish temporary nodes from persistent nodes. A temporary node is only collectible upon the return of a function call. This insures that a node will not be collected as long as the state of the G-machine may include a pointer to that node. Temporary nodes that are to be examined for collectibility are identified using the stack mechanism described above. On the other hand, persistent nodes can be examined for collectibility at any time.

GARBAGE COLLECTION ALGORITHM

The garbage collection algorithm used in the design is a concurrent modified reference counting scheme. A reference counting scheme is well-suited for a dual-processor implementation because the operations required to maintain reference counts are inherently local operations. As a result, little explicit cooperation is required between the evaluator and the collector. In addition, using a separate processor to collect garbage in parallel with the evaluation process is the only way truly real-time list-processing is achievable.

A shortcoming of the traditional reference counting scheme is that it cannot collect cyclic structures. A simple example illustrates this:

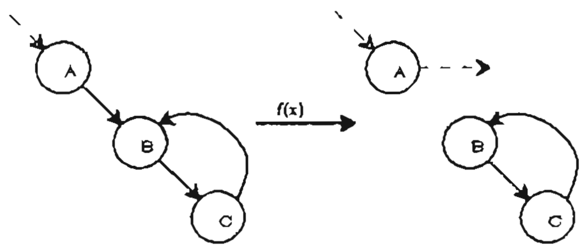

Figure 2. Example of garbage creation.

Let A, B, and C be a set of linked data elements in memory as shown in Figure 2. After the operation f(x), A no longer references B. The graph rooted at B is garbage, but its reference count is one due to the reference from C to B. Therefore, the reference counting scheme does not detect that B and C can be collected, since only those elements with a reference count of zero are identified as collectible. Concurrent garbage collection schemes have been proposed which use reference counting, but collection is augmented by an independent trace-of-accessible-storage collection in order to collect cyclic structures [Deu76, Wis77].

The algorithm used in this design is a concurrent reference counting scheme which was developed at OGC [Deb84, Kie86, Fos85]. Included in the algorithm is a mechanism for detecting and collecting cyclic structures. Cyclic structures are identified by traversing sub-graphs whose roots have been identified as potentially collectible. Since the algorithm supports a dual-processor implementation, the G-machine should be interrupted only if it is creating objects faster than garbage can be salvaged. Because of the parallel nature of the algorithm and its ability to detect cyclic structures, it meets the garbage collection requirements of the G-machine.

Conceptually, the garbage collection algorithm can be divided into two kinds of operations; 1) those operations which manage the reference counts, and 2) those operations where the collectibility of a node is determined. Both of these aspects of the algorithm are described.

Managing Reference Counts

As with the traditional reference counting scheme, reference counts may be modified when the graph image in memory is altered by the evaluator. In a write operation to memory, reference counts are incremented if the datum being written is a pointer. Similarly, in a trash operation, reference counts are decremented if the node being overwritten contains a pointer.

Reference counts are also modified when nodes are collected. If the contents of the node that is collected contains pointers, then the reference counts of those nodes pointed to are decremented.

Determining Collectibility

Nodes which are examined for collectibility are obtained from two sources; temporary nodes allocated during a function call, and persistent nodes whose reference counts have been decremented. Persistent nodes can have their reference counts decremented as the result of the G-machine trash instruction or when a node that points to the persistent node is collected. In the latter case, a distinction is made between those persistent nodes that are part of a cyclic sub-graph that is being collected and those that are not. The reason for this will become apparent when the cycle-detection portion of the algorithm is explained.

If the reference count of a node N is zero, then N is immediately collectible. If N contains a pointer, then the reference count of the node to which it points (M) is decremented. If node M is persistent, it is added to the set of nodes that are to be examined for collectibility. Otherwise node M is a temporary node and is already (or will be) in the set of nodes to be examined for collectibility upon the return from a function call.

If the reference count of the node N is greater than zero, then the sub-graph rooted at node N is traversed to determine if it is a cyclic structure that is collectible. The main function of the traversal process is to count the number of local arcs (or references) in the sub-graph and compare this value to the actual reference count. If the values are equal, then it is assumed there can be no external references to the node and it is collectible. The concurrent algorithm requires three additional data fields besides the reference count for each graph node. These are 1) the *local reference count*, 2) the *recently_visited* bit which is used to indicate that the evaluator has modified the expression graph, and 3) the *is_collectible* bit which is used to indicate the collectible status of a node.

The concurrent garbage collection algorithm requires four traversals of the sub-graph. Four traversals may seem as an excessive amount of overhead, but it is important to note that the approach is *incremental* in that the sub-graph being traversed is a small portion of the expression graph in memory. In Figure 3 the sub-graph rooted at node N is traversed to show how the collectibility of the nodes N, O, P and Q is determined.

In Traversal 0, the local reference count and the recently_visited bit are cleared. A non-zero value for the local reference count terminates the traversal. During Traversal 1, the local reference count is incremented each time the node is visited. Traversal 1 does not visit the children of those nodes with the recently_visited bit set. Note in the example that the G-machine has visited node O during the time between Traversal 0 and Traversal 1 (node O has had its reference count incremented, and its recently_visited bit is set). Therefore, node Q remains unaffected after Traversal 1.

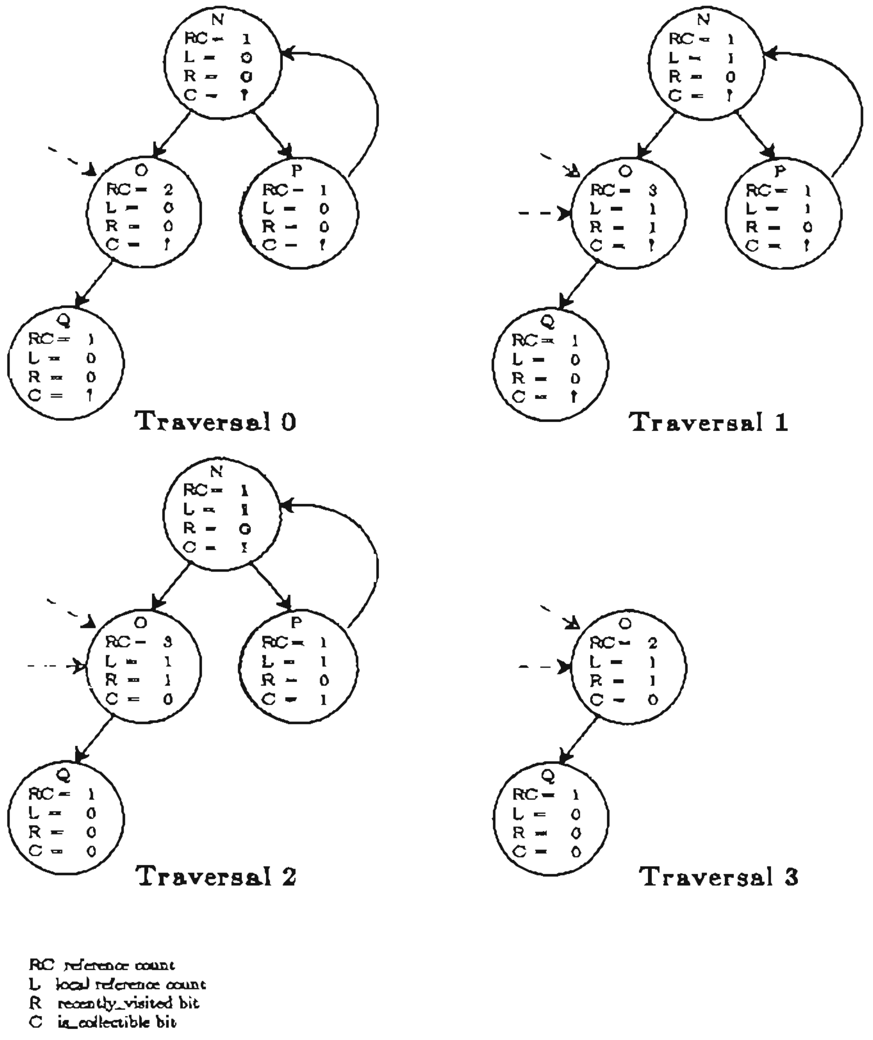

Figure 3. Example of sub-graph traversal.

In Traversal 2 the local reference count is compared to the reference count. If they are equal, then the node is marked as collectible. Once a node is encountered that is uncollectible, then all nodes that are children of that node are marked uncollectible as well. In Figure 3, both nodes *O* and *Q* are marked as uncollectible after the completion of Traversal 2.

In Traversal 3 the nodes that are marked as collectible are collected. For each node that is collected, the reference count is cleared. During Traversal 3, the first node that is encountered in the sub-graph that is not collectible (node O in the example) must have its reference count decremented. Since the collectibility of the node has already been determined by the graph traversal process, it is not necessary to add it to the set of nodes to be examined for collectibility.

MEMORY MANAGEMENT ROLES OF THE HOST

As the G-machine is evaluating the expression graph in memory, garbage collection is conducted in parallel by a separate processor. This processor, called the host processor, performs other related functions such as maintaining a free list, allocating nodes to the G-machine, and implementing the stack allocation with persistence scheme. The remainder of this section describes how these functions are implemented.

Maintaining a Free List

The method used by the host for maintaining the list of free nodes is a variant of the "buddy system" [Knu68] as described in [Kie86]. The memory address space is divided into blocks, each of which has a buddy (save the largest one) whose address differs from its own in just a single bit. Each of these blocks has a flag bit associated with it known as the *allocated_flag*. As nodes, and thus blocks become allocated, the allocated_flag associated with each node (and block) is set. Conversely, as nodes, and thus blocks are collected, the allocated_flags are cleared. Figure 4 shows the blocks and how the allocated_flags might be set for a memory with eight nodes. Three nodes (2, 6 and 7) are not allocated.

Address	allocated_flags		
000	0	1	1
001			1
010		0	0
011			1
100	0	1	1
101			1
110		0	0
111			0

Figure 4. Example of buddy system memory allocation.

If the host started at the beginning of the memory address space, the time required to find the next free node would be O(log(x)), where x is the size of memory in nodes. If the host stores the address of the last node allocated, however, the average time required to find the next available node becomes constant and is independent of memory size. Therefore, when the host allocates a new node, it stores the address in a register to be used in processing the next allocation request.

Stack Allocation with Persistence

Temporary nodes to be examined for collectibility are identified through the use of the stack allocation with persistence mechanism described earlier. To restore the addresses of temporary nodes allocated during a function call, the host processor can take advantage of the way the buddy system allocates nodes in memory. Since the host is using the address of the last node allocated to identify the address of the next free node in memory, nodes are allocated linearly through the address space. As a result, nodes allocated during a function call are in the set of nodes that lie between the address of first node allocated for the function and the address of the last node allocated.

Given the above conditions, storing the address of each node allocated during a function call in a stack is not required. Instead, a stack, called the *allocation stack*, can be used where each word in the stack need only contain the address of the last node allocated for each function call. Upon a return from a function call, the addresses of those nodes that were allocated during the call will lie between the addresses of the top two words of the allocation stack. The host tests the allocated_flag and the persistent_bit for each node in this address space to obtain the set of nodes that are to be examined for collectibility. Only those nodes that have the allocated_flag set and the persistent_bit cleared will be examined.

Allocating Nodes

One of the memory requirements of the G-machine is that memory must be able to provide the address of the next free node upon request. Since the host maintains the free list, its functions also include identification of the next free node to be allocated. Rather than having the host respond to each G-machine allocation request directly, a buffer is used to provide the addresses of free nodes to the G-machine. This buffer allows the G-machine to continue processing without have to wait for the host to process each request. The host, therefore, upon an allocation request passes the address of the next free node it identifies to the buffer instead of the G-machine.

Unfortunately, allocating nodes is not as simple as merely passing free node addresses to the buffer. The host must be able to obtain the address of the node that was allocated to the G-machine in order use the stack method for identifying temporary nodes. Since the situation could occur that the G-machine is requesting nodes faster than the host could process the output of the buffer, the host must keep its own copy of the buffer.

To process an allocation request, the host moves the address of the node at the head of its copy of the buffer to the top of the allocation stack. The next free node is found using the buddy system described earlier and passed to the buffer. This free node address is also added to the hosts copy.

Collecting Garbage

The garbage collection algorithm used by the host is the modified reference counting scheme described in the previous section. The set of nodes to be examined for collectibility is obtained from three sources; 1) temporary nodes as described above, 2) persistent nodes whose reference counts have been decremented by a trash, and 3) persistent nodes whose references counts have been decremented as a result of garbage collection. The host places the highest priority on allocating free nodes and collecting temporary nodes and the lowest priority on examining persistent nodes whose reference counts have been decremented. This priority scheme corresponds with the demand of the G-machine.

Because of the parallel nature of the algorithm and its implementation, situations where read-modify-write collisions of shared data can occur. A collision occurs when two processors simultaneously access a data element, modify it, and write out their new value to memory. As a result, the new value is different than what would be obtained if one processor performed the same data operations sequentially, and, most importantly, is incorrect.

The only shared data element that the host processor does not explicitly set or clear is the reference count. Therefore, only the reference count is subject to read-modify-write collisions. There are two occasions where the host modifies reference counts of objects in memory; 1) when a node is immediately collectible and it contains pointers to other nodes, and 2) when a node in a subgraph is uncollectible and the nodes which point to it are collected. On these occasions, the host reads the reference count, decrements it, and writes the new value out to memory. These situations occur relatively infrequently when compared to the number of times reference counts are modified due to G-machine write and trash operations. Therefore, detecting situations where a collision could occur is not a function of the host processor, but a function of the component that maintains reference counts on G-machine write and trash operations.

To assist the detection of collisions the host processor outputs an additional signal during the time it is decrementing a reference count. This signal indicates that the host is entering a sequence of operations where a collision could occur. If a collision has occurred, then the component which has detected the collision will fault and restart its memory access when the host has signaled it has completed.

In summary, the host performs several memory management functions. It collects garbage nodes, maintains the free list and allocates new nodes when requested. The memory requirements of the host processor, the garbage collection algorithm and the G-machine can now be used to define the functional requirements, and thus the architecture, of the memory.

MEMORY ARCHITECTURE

Given the memory requirements of the G-machine, the concurrent garbage collection algorithm, and the host processor, a memory architecture designed to meet those requirements is specified. This section provides a high level description of that architecture and thus lays the foundation for the microarchitecture and prototype design.

Tagged Architecture

Both the G-machine and the memory management scheme require several data items that are to be stored with each graph node. Figure 5 shows the size and provides a brief description of each of these data items. Since the local reference count, is_collectible bit, and the allocated_flag are only used by the host, these fields are stored in a memory bank accessible only to the host processor. The remainder of the fields will be stored in a memory that is shared. Separating the data in this manner reduces the amount of memory contention during processing.

In order to provide efficient access of shared data, memory is dual-ported. This dual-ported memory includes a mechanism so that two independent busses (the host bus and the G-machine bus) can access a common memory.

Field Name	# bits	Description
local reference count	8	Holds local arc count during sub-graph traversal.
is_collectible	1	Indicates if node is collectible during sub-graph traversal.
allocated_flag	1	Set when node has been allocated to the G-machine.
reference count	8	Represents the number of references from the active expression graph.
recently_visited	1	Set when reference count of node has been modified by G-machine operations.
persistent_bit	1	Indicates if node is persistent or temporary.
forever_uncollectible	1	This bit is set when the reference count field overflows.
is_evaluated	1	Indicates if node has been evaluated by a previous reference.
is_pointer1	1	Indicates if first data field is a pointer or a basic value.
data1	32	First data field.
is_pointer2	1	Indicates if second data field is a pointer or a basic value.
data2	32	Second data field.

Figure 5. Fields of a graph node.

Memory Functions

As with conventional memories, the memory must provide a mechanism for data storage and retrieval for both the host processor and the G-machine. However, in order to meet the requirements of the G-machine and the memory management scheme, the role of the memory grows beyond that of conventional memories. As a result, memory consists of a set of interacting units which form a complex whole, and will be referred to as the *Graph Memory System* (GMS). The term *memory* will be used to refer to that portion of the GMS that provides the conventional memory functions. Besides data storage and retrieval, other GMS functions include the maintenance of reference counts and the buffering of data for the host and the G-machine.

Maintaining Reference Counts

The garbage collection algorithm requires the maintenance of reference counts, which can be an expensive operation if executed in software by the G-machine. [Ung84] estimates that an additional twenty percent more CPU time is required to maintain reference counts. This overhead certainly justifies the design of specialized hardware to support reference counting and moving that task to the GMS.

The design of this hardware is simplified with the addition of a G-machine trash instruction. The G-machine emits the trash instruction when it is about to "update", or overwrite the contents of a node. This is the only instance where reference counts are decremented. As a result, the hardware will always increment reference counts on a write operation and decrement on a trash operation.

Buffering Data

The benefit of buffering is that it reduces the number of wait states and interrupts that may be required during processing. There are three kinds of data that are used in the memory management scheme that are buffered. The first type of information that is buffered are the addresses of free nodes that are to be allocated to the G-machine. Free node addresses are placed in the buffer, called the *FIFO*, by the host processor and given to the G-machine upon request. A hardware buffer can respond to an allocation request from the G-machine much faster than can the host processor.

A hardware buffer is also provided for signals emitted by the G-machine to indicate a call, return or alloc. This information is required by the host processor to control garbage collection. This buffer, which is called the *Signal Queue*, relieves the G-machine from having to enter a wait state in order to synchronize the transfer of data to the host. As long as the host processes these signals in the same order as they are generated by the G-machine, the internal state of the host will accurately trace that of the evaluation as it proceeds.

A third buffer, the *Garbage Can*, is used to store the addresses of persistent nodes whose reference counts have been decremented. These nodes are identified and placed in the buffer by the reference counting hardware. They are retreived by the host processor and examined for collectibility. This buffer allows the reference counting hardware to continue to process write and trash operations without having to wait for the host to determine the collectibility of a node it has previously identified as potentially collectible.

Data and Signal Paths

Figure 6 shows the data paths and signals between the GMS, the host and the G-machine that would be required to implement this functionality. The following explains each of the signals shown in Figure 6:

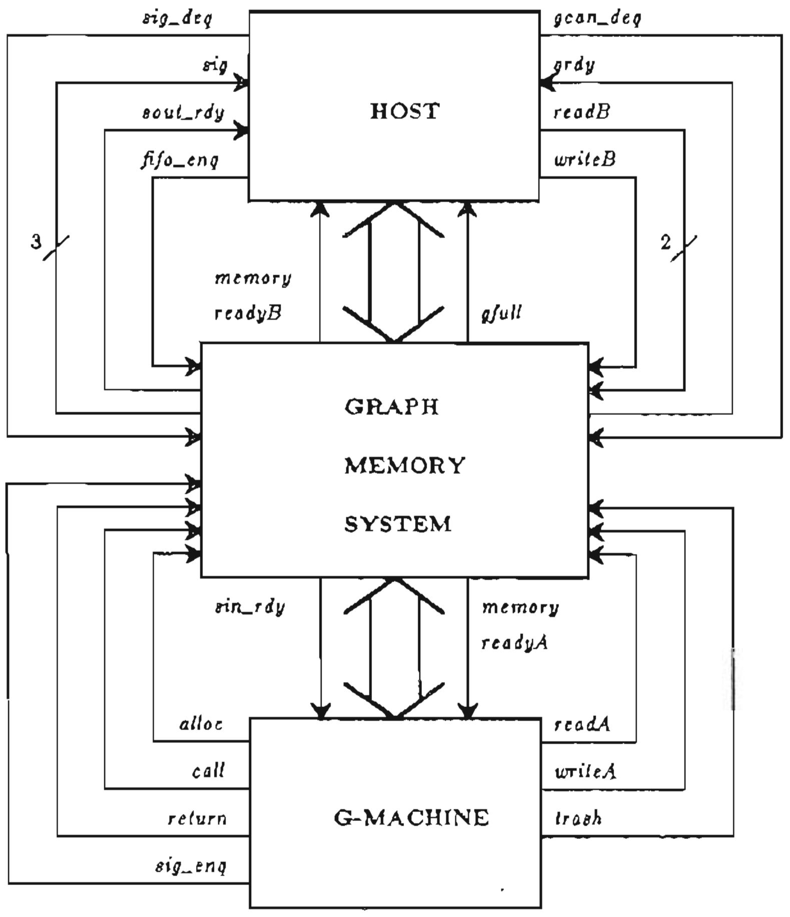

Figure 6. Data paths between the GMS, the Host and the G-machine.

(1) *sin_rdy*. The value of sin_rdy indicates the ready status of the Signal Queue. If the Signal Queue is ready, then instructions can be enqueued for later processing by the host.

(2) *alloc*. If memory is ready and sin_rdy indicates that the Signal Queue is ready, then the G-machine can emit the alloc signal. Alloc dequeues a new node address from the FIFO, and is itself enqueued by the Signal Queue to be processed later by the host.

(3) *call*. If the Signal Queue is ready, the G-machine can emit the call signal.

(4) *return*. If the Signal Queue is ready, then the G-machine can emit the return signal.

(5) *sig_enq*. Sig_enq is used by the G-machine to enqueue instructions on the Signal Queue.

(6) *memory_readyA*. Memory_readyA is used to indicate the ready status of the memory port that is used by the G-machine. The ready status of port A is independent of the ready status of port B.

(7) *readA*. If port A of memory is ready, then the G-machine can emit a readA signal.

(8) *writeA*. If port A of memory is ready, then the G-machine can emit a writeA signal. The writeA signal will be processed by the memory portion of the GMS and the reference counting hardware.

(9) *trash*. The G-machine can emit a trash when port A of memory is ready. Trash is processed by the reference counting hardware of the GMS.

(10) *memory_readyB*. Memory_readyB is used by the host processor to obtain the ready status of its memory port.

(11) *writeB*. The host may write to memory when port B of memory is ready.

(12) *readB*. Similarly, the host may read from memory when port B of memory is ready. One bit of the hosts readB signal is used to indicate that the host is modifying the reference count field. When this occurs, the reference counting hardware of the GMS will enter a detect collision state.

(13) *grdy*. The value of grdy indicates to the host processor that the Garbage Can contains persistent nodes whose collectibility need to be determined.

(14) *gcan_deq*. The host uses the gcan_deq signal to dequeue nodes from the Garbage Can.

(15) *gfull*. The value of the gfull signal indicates to the host the full status of the Garbage Can. If the Garbage Can is full, then immediate attention is required.

(16) *sout_rdy*. The value of sout_rdy indicates to the host if G-machine instructions have been buffered by the Signal Queue.

(17) *sig_deq*. If the Signal Queue contains instructions to be processed, then the host processor uses sig_deq to dequeue an instruction.

(18) *sig*. The value of sig represents the alloc, call, or return that was enqueued by the Signal Queue.

(19) *fifo_enq*. If the value of siq was alloc, then once the host has identified the next node to be allocated to the G-machine, it uses fifo_enq to add the address of the node to the FIFO.

GRAPH MEMORY SYSTEM MICROARCHITECTURE

The Graph Memory System (GMS) architecture consists of 1) three FIFO queues that provide buffering of data between the host and the G-machine, 2) a dual-ported memory that allows two different busses to independently access the memory, 3) an addressing scheme that supports tags and a list-structured memory, and 4) a custom VLSI component whose main function is to maintain reference counts. The GMS must be able to interface with the G-machine whose clock speed is expected to be 10 MHZ and the Host processor (the Motorola 68020 [Mot85]) whose clock speed is 16 MHZ. Figure 7 shows the microarchitecture of the GMS. A description of this microarchitecture and the hardware components which were used in the prototype design follows.

The Buffers

The GMS has three queues that buffer information passed between the Host and the G-Machine. The component that implements these buffers is Monolithic Memories First-In First-Out (FIFO) Cascadable Memory [Fir84]. The component is asynchronous with a shift-in/shift-out rate of 10 MHZ. Not shown in Figure 7 are the latches [Int84] required to tri-state the outputs of the FIFO and the Garbage Can (GCAN). Functionality of each queue is as follows:

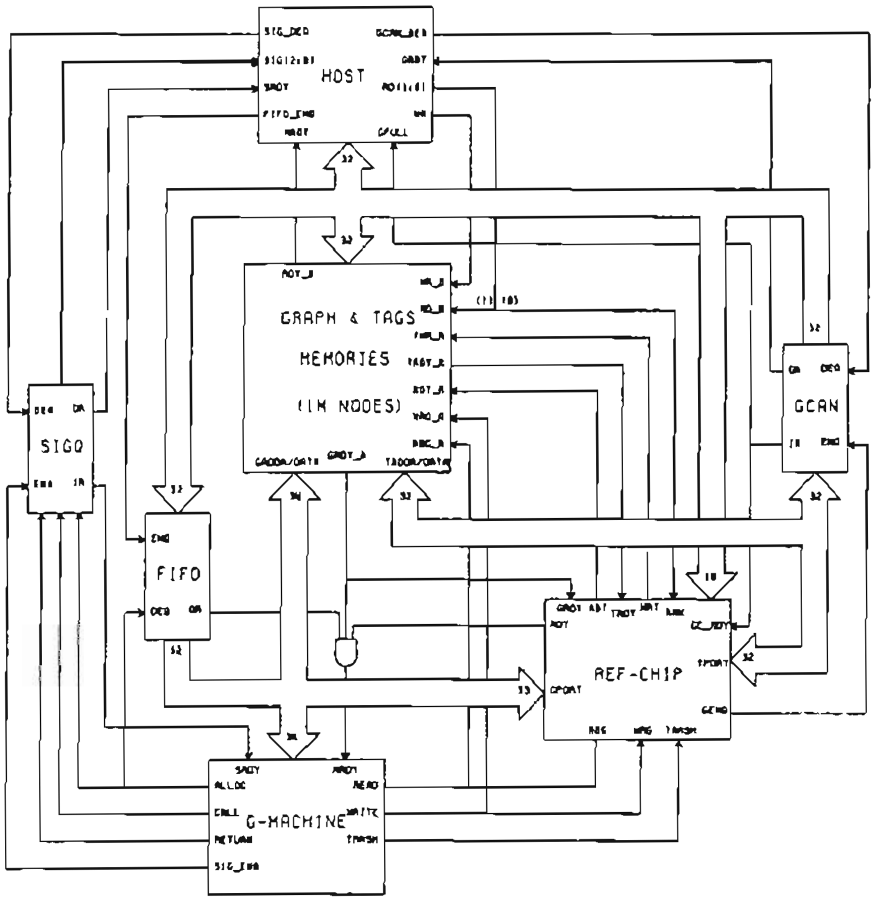

Figure 7. Data paths with Graph Memory System expanded.

(1) *SIGQ*. This is the Signal Queue that buffers G-machine instructions that are to be processed by the host.

(2) *FIFO*. The FIFO holds addresses of free nodes in memory and services allocation requests from the G-machine. When no valid output is available the memory_ready signal seen by the G-machine is unasserted. The Monolithic Memories FIFO component has a 1.3 microsecond data throughput or "fall through" time. Therefore, the FIFO queue should be filled by the host processor before evaluation commences in order to avoid a lengthy delay.

If the FIFO frequently becomes empty, then perhaps another component with a shorter "fall through" time should be considered. The FIFO only becomes empty when the G-machine emits several allocation requests in a short period of time, or the host cannot keep up with garbage collection. Data from simulations suggest that the SIGQ will fill up before the FIFO becomes empty. This implies that the G-machine will wait more frequently for the availability of the Signal Queue rather than for data to "fall through" the FIFO.

(3) *GCAN.* The Garbage Can buffers the addresses of nodes that are potentially collectible. These nodes are identified by the reference counting hardware.

The Dual-Ported Memory

The components that implement the memory portion of the GMS are Intel's Advanced Dynamic RAM Controller i8207 [Adv84] and Texas Instruments 256K Dynamic Random-Access Memory components [Tex84]. The i8207 provides the dual-ported memory interface allowing two different busses to independently access memory. The main functions of the i8207 are to arbitrate memory requests between the two ports, provide the control for the dynamic RAMs, generate the refresh control given a refresh period, and generate the memory ready signals.

Since the i8207 can run at 10 MHZ, the G-machine/memory interface is synchronous and is designed as a no-wait state system [Rig84]. A no-wait state system takes advantage of the fact that the components are synchronous and minimizes the delay required for memory accesses based on their timing. The minimum number of cycles for a G-machine memory access, from the time it sends the signal to memory to the time it can latch the data, is three cycles (or 300 ns with a 10 MHZ clock).

The Host/memory interface is asynchronous and a bus controller (i82288 [Int84], for example) is required. The bus controller adds at least one cycle to the memory access time. The acknowledge signal produced by the i8207 is also emitted one cycle later than the acknowledge signal for its synchronous port. The delay in the acknowledge signal is required to insure that valid data is on the address/data bus before the host latches the data. A minimum of five cycles (or approximately 300 ns for a 16 MHZ clock) is required for a host memory access.

Memory Addressing Scheme

The portion of the graph node that is shared by the host, the G-machine, and the reference counting hardware is divided into two parallel memories. The first area is called the Graph Memory and contains the is_evaluated bit, the two data fields, and the two is_pointer bits. These data are modified only by the G-machine, but read by both the reference counting hardware and the host. The second area (the Tags Memory) contains the remainder of the graph node; the reference count, recently_visited bit, persistent_bit, and the forever_uncollectible bit. The G-machine does not access these data. By thus partitioning the physical memory, memory contention is significantly reduced.

Figure 8 shows how a memory module, holding up to 10^6 graph nodes would be configured. The i8207 supports up to a maximum of four 256K memory banks that are low-order interleaved. For four 256K banks, twenty bits of address are required. To support a list-structured memory, the data fields of the graph nodes must be individually addressable. Therefore, two additional signals are required to select the DRAM controller which controls the memory bank holding the desired data field.

Since the tags are stored in a separately accessible area, one signal line must be used to select the i8207 which controls access to that data.

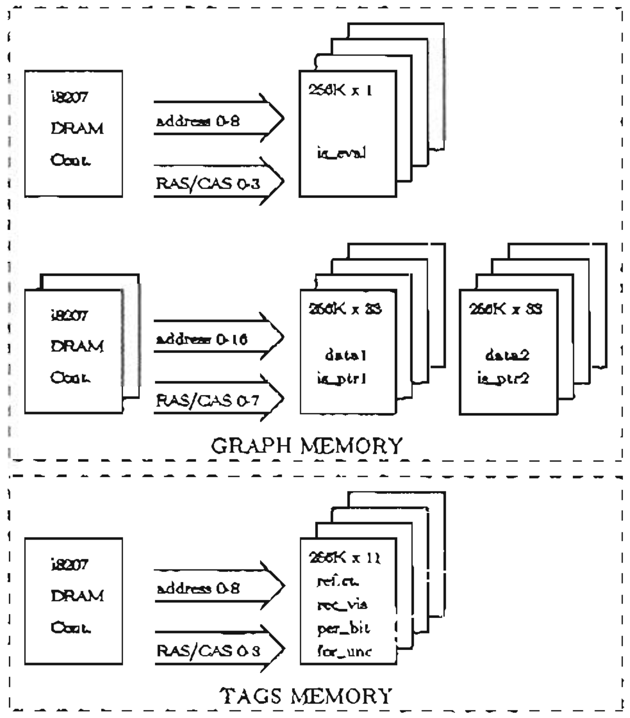

Figure 8. A memory module.

If the signals needed to select the DRAM controllers on a memory access are encoded as part of the address, then an initial 23 address bits are required for the first memory module. Each additional memory module requires 3 additional address bits to select the DRAMs as described above. Using this addressing scheme, a 32 bit-wide address will support up to 4 memory modules, or a total of approximately 40 Mbytes of real memory.

The Reference Counting Hardware (Ref-Chip)

The only component in the GMS requiring a customized design is the ref-chip. Its main functions are: 1) to maintain reference counts, and 2) to identify potentially collectible nodes and pass them to the host (via the Garbage Can). The ref-chip is synchronous with the G-machine (and thus with memory) and runs at 10 MHZ. The ref-chip receives instructions from the G-machine and processes them one at a time in the order they are received. If the ref-chip is in the process of executing an instruction, a buffer of length one is included in the design which is able to store the next instruction from the G-machine. When this buffer becomes full, the G-machine must be inhibited from accessing memory until the ref-chip has completed processing its current instruction. The ref-chip inhibits the G-machine from accessing memory through the use of its *rdy* signal.

Because the graph node is divided into two separately accessible areas, the ref-chip has two ports; one which provides access to the Graph memory, and the other which provides access to the Tags memory and the Garbage Can. The only time the ref-chip accesses Graph memory is immediately after the receipt of a trash instruction. The remainder of the time, only Tags memory or the Garbage Can is accessed. By having two separate ports, the ref-chip can do most of its processing and memory accesses in parallel with the G-machine. The remainder of this section describes how the ref-chip implements its functionality. A more detailed description of the specifications for the ref-chip design is provided in Appendix A.

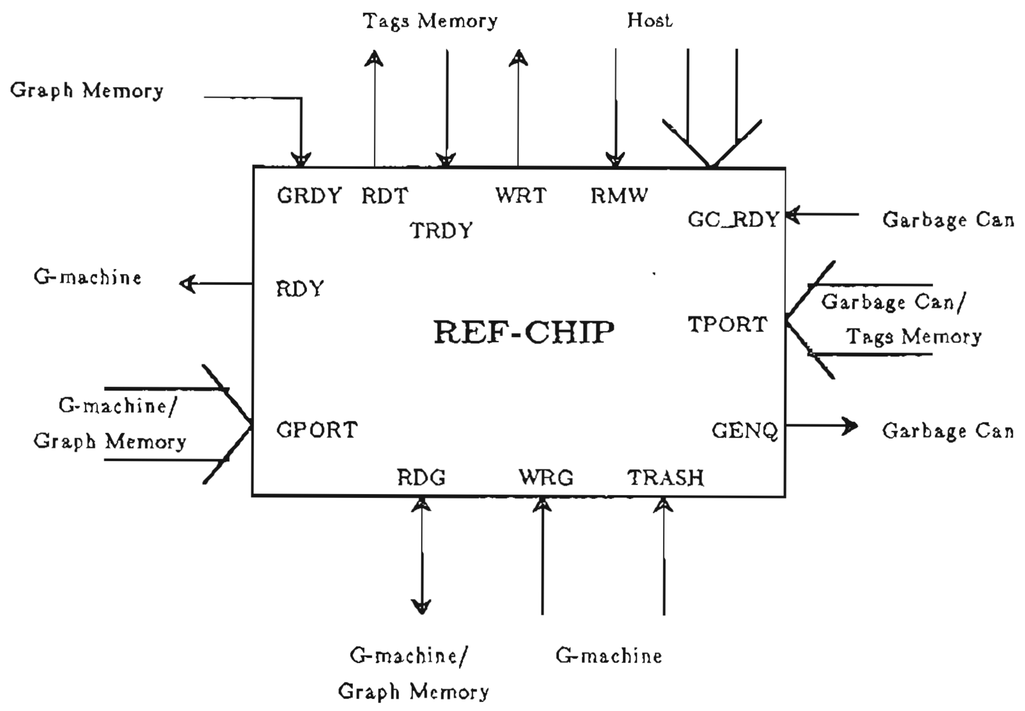

Figure 9. Sources and destinations of ref-chip signals.

Maintaining Reference Counts

Upon receiving a write signal from the G-machine, the ref-chip will latch the value of the datum being written from its GPORT. If the datum being written is a pointer, then the ref-chip will increment the reference count using its TPORT. Upon receiving a trash, the ref-chip will again latch the address of the trash from its GPORT. The ref-chip emits "not ready" to the G-machine prohibiting access by the G-machine to this address in Graph memory until the ref-chip is able to read its contents. If the data field of the node contains a pointer, reference counts for the node to which it points are decremented using the ref-chips TPORT.

The reference count field is the only data stored in memory that is subject to read-modify-write sequences by both the host processor and the ref-chip. To avoid collisions, the ref-chip includes a mechanism to detect these situations. This mechanism requires only one additional signal that is to be emitted by the host to the ref-chip. This signal (*RMW*) is to remain valid during the entire time the host is modifying a reference count. While this signal is asynchronous to the ref-chip, it is assumed that the bus-controller for the host/memory interface will provide the necessary synchronization.

The ref-chip samples the RMW signal every cycle. On the first cycle that the signal is valid, the ref-chip latches the address from the host address/data bus. The ref-chip enters a detect-collision state and does not return to its original state until the host signals it has completed.

If at any time during a read-modify-write sequence the ref-chip enters a detect-collision state, then the ref-chip must determine if the host is accessing the same memory address by testing the contents of the address it has latched. If the host is accessing the same address and the ref-chip is in the read or modify stage of the sequence, the ref-chip must fault and restart its action when the host indicates that it has completed. If the ref-chip has already written the data, then no fault is required.

The amount of processing time that is lost when a collision between the host and the ref-chip occurs is quite high (just accounting for memory access time by the host, a minimum of approximately 900 ns). However, if each address in a memory module is equally likely to be used, then the probability of a collision occurring is about one in a million.

Identifying Potentially Collectible Nodes

When the ref-chip decrements the reference count of a node, it tests the persistence bit to determine if the node should be passed to the Garbage Can. If the persistent_bit is not set, then this indicates that the node is a temporary node and will eventually be examined by the host. If the persistent_bit is set, then the TPORT is used to pass the node to the Garbage Can.

SIMULATING THE GMS DESIGN

An initial simulation of the garbage collection algorithm was conducted by [Fos85] in order to evaluate the performance and correctness of the algorithm. Included in the simulation was a dynamic graph image created by a mutator (the G-machine), and a host that performed the garbage collection. Analysis of equilibrium conditions in the simulation showed that the amount of processing required by the host was not excessive. What was not included in the simulation model by [Fos85] were delays introduced in mapping the algorithm to a hardware configuration, and the impact of memory contention. Therefore, the main purposes of simulating this memory design were to aid in validating the architecture and to measure the impact of resource contention upon performance. Simulation was also used as a tool for analyzing the relative performance of different design configurations.

The simulation was written in C and used the **Teamwork** [Tea86] subroutine library. **Teamwork** allows one to build programs as a set of concurrent cooperating tasks with scheduling and communication provided by **Teamwork** primitives. C augmented with the **Teamwork** library was chosen as the simulation language over others (particularly N.2) because of the freedom in the level of abstraction used to model the architecture. Also, it is much faster than N.2, so thousands of simulation cycles could be executed fairly quickly.

Since the purpose of simulation was to validate the design and evaluate the effect of resource contention upon performance, a detailed simulation of the graph and the garbage collection algorithm was not necessary. Instead, graph reduction by the G-machine and and garbage collection by the host were modeled stochastically by the frequency of memory accesses. On the other hand, the timing of the GMS hardware and the signals between the various functional units were modeled accurately. A few concessions had to be made due to the way **Teamwork** passes messages between tasks, but for the most part, the simulation is phase accurate for the signals generated during processing.

The remainder of this section discusses how the workload was defined for the simulation, the assumptions that were made in the model, and the effect of those assumptions upon the simulation results. A more detailed description of how the design was modeled is provided in Appendix B.

Defining a Workload

The G-machine has two types of instructions; CISC instructions and RISC instructions. The CISC instructions are those instructions that are generated by the compiler and are to be executed by the G-machine. The Instruction Fetch and Decode Unit [Kie85,Ran86A] of the G-machine expands the CISC instructions into a sequence of one or more RISC instructions. These RISC instructions are sent to the Processor Control Unit (PCU), which dispatches control signals on a cycle-by-cycle basis [Ran86B]. The signals emitted to the GMS are based on the RISC instructions processed by the PCU. Therefore, the workload for the GMS is defined as the frequency of those RISC instructions that would emit the GMS signals call, return, read, write, trash or alloc.

Unfortunately, dynamic RISC sequences produced during evaluation were not available. Therefore, a static analysis of the RISC sequences defined for each CISC instruction in [GAH86] was used to derive the frequency of instructions. If each CISC instruction was processed exactly once, then out of the expected total number of cycles required to execute these instructions, approximately 16 percent of the cycles would include a signal sent to memory. For these memory signals, the relative frequency of each type of signal is shown in Figure 10.

Instruction	Percent
read	28.1
write	42.1
trash	12.3
alloc	10.5
call	3.5
return	3.5
Total	100.0

Figure 10. Relative frequency of GMS instructions.

The length of the simulation was defined by specifying the number of G-machine RISC instructions to be "executed". The workload for the GMS is the percentage of those RISC instructions that emit signals to memory. This percentage was defined by specifying a rate of G-machine allocations per second. Increasing the allocations per second increased the percentage of instructions executed by the GMS, but the relative percentage of each of the GMS instructions remained constant. Although an approximate rate of up to 100K allocations per second is expected for the G-machine, the actual rate is a function of the program and can vary. Therefore, the allocations rate was a simulation parameter that could be varied. Varying the allocation rate provided valuable information about the sensitivity of performance to the frequency of allocations.

Simulation Assumptions

Since the simulation did not include an actual memory or the creation of an actual graph image, several assumptions about the graph image had to be made. These assumptions were expressed as probabilities so that decisions in the simulation could be made using the value of a random variable. Simulation assumptions regarding the characteristics of the expression graph are as follows:

(1) When the Ref-chip receives a write or a trash, 50 percent of the time the node is a pointer and reference counts need to be modified.

(2) When processing a trash and the node is a pointer, the node whose reference count is decremented is always sent to the Garbage Can. This makes the model more pessimistic in terms of the number of nodes requiring examination by the host.

(3) The assumptions regarding the size of the sub-graphs traversed by the host were largely based on a crude analysis of data provided by [Fos85]. The data consisted of a series of time interval checkpoints where the number of root nodes, the number of nodes visited and the number of nodes collected for both temporary nodes and persistent nodes from the Garbage Can were provided. For each of these checkpoints, the average number of nodes visited (sub-graph size) and the average number of nodes collected per graph was calculated. Figure 11 shows the probability distributions of sub-graph sizes that were obtained from this analysis.

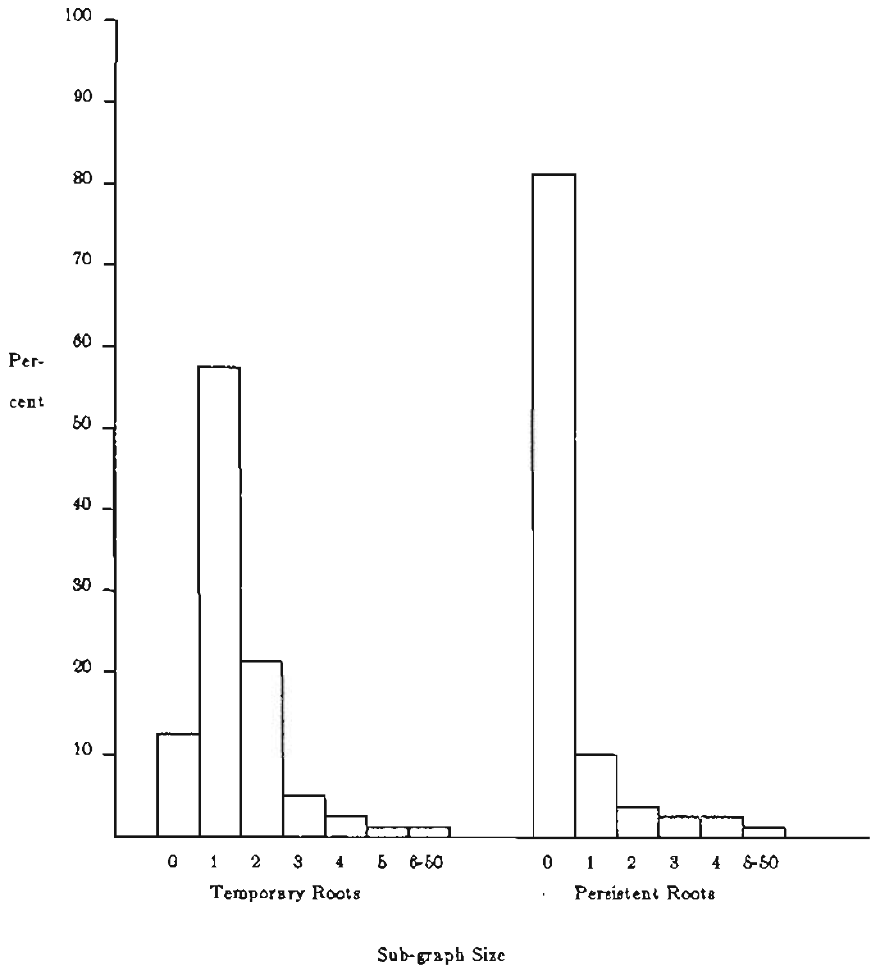

Figure 11. Probability distributions of sub-graph sizes.

As can be seen, the number of nodes visited by the host during graph traversal is expected to be quite small. Since no actual data was available to verify this information, it would be helpful to have information about the sensitivity of the garbage collection scheme to the average sub-graph size. To provide this information, simulations could be run using the above probabilities, or probabilities could be derived from Poisson distributions. The Poisson distribution was chosen because it could be defined to have an appearance similar to the distributions shown above, or by varying the parameter m, the distribution could be shifted to the right or left.

(4) Analysis of the [Fos85] data also showed that an average of .95 nodes were collected for each temporary root whose sub-graph size was zero. Similarly, an average of .80 nodes were collected for those persistent node roots with nil sub-graphs.

(5) During the Traversal 3 of a sub-graph by the host, 50 percent of the time a node is reached that is uncollectable and its reference count must be decremented.

Providing an accurate model of the host processor was also difficult. Since operations performed by the host processor are executed in software, including instruction execution times in the model would have been the best approach. Unfortunately, the design of the MC68020 makes it very difficult to calculate exact instruction timing. The combined effects of the on-chip cache, instruction prefetch, operand misalignment, and instruction execution overlap result in varying execution times from one context to another. Not only does the MC68020 overlap instructions, it has an internal bus controller that can perform bus operations in parallel with internal processor operations. Since the processing required by the memory management scheme is dominated by memory accesses, it was concluded that memory access time would be the limiting factor in the hosts performance. Therefore, the memory management functions of the host processor were modeled as sequences of memory accesses.

The two assumptions that have the largest impact on the accuracy of the simulation are 1) the manner in which workload was defined, and 2) the assumptions regarding sub-graph size. Both of these aspects of the model were included as simulation parameters. The ability to vary the workload and the size of the sub-graph in simulations provides information about the performance of the design under varying conditions.

SIMULATION RESULTS

The purpose of simulating the design was twofold; 1) to aid in validating the architecture, and 2) to provide information about the impact of resource contention upon performance. Simulation was quite effective as a validation tool. The architecture described herein is the result of several iterations in which flaws were discovered through simulation. In order to provide information about the impact of resource contention, the simulation was designed so that a variety of statistics could be gathered. This section presents an analysis of some of those statistics with respect to their implications about performance.

Analysis of the Design

The throughput of the G-machine was used to compare the effectiveness of the design under various conditions. This throughput was measured as the simulated time required to execute 500K G-machine RISC instructions. Several simulations were executed to find the point where various simulation statistics converged and became stable. At approximately 400K RISC instructions convergence was observed in the average number of cycles the G-machine waited for a memory ready signal and the average number of cycles for a G-machine memory access. Therefore, 500K was chosen as the number of instructions to simulate.

To compare the relative impact of the reference counting hardware and graph traversal by the host upon the throughput of the G-machine, three models were simulated. These models were:

(1) *G-machine only.* No garbage collection was performed at all.

(2) *Reference count collection only.* Garbage collection was based only on the value of reference counts (no sub-graph traversal to detect cyclic structures). This model was used to provide information about the cost of the hardware supported reference counting.

(3) *Graph traversal.* The full garbage collection scheme was simulated.

Several simulation parameters could be specified to analyze different design configurations. Two memory modules could be simulated instead of one, and the arbitration algorithm used by the i8207 could be varied. For all simulations presented here, however, only one memory module was simulated and the *PortA Priority* algorithm for arbitration was used (earlier simulations had shown that these parameters had relatively little impact upon overall performance). Graph sizes were generated using the data provided by [Fos85] unless otherwise specified.

The simulated time required to execute 500K G-machine RISC instructions for each of these models is shown in Figure 12. The allocation rate was set at the expected level of 100K nodes/second. From the data it can be observed that a traditional reference counting scheme could be implemented with little impact on the throughput of the G-machine. Also, most of the overhead of the full-traversal model is due to the cycle-detection portion of the algorithm.

Similar information for the Symbolics ephemeral garbage collector in [Moo84] is also provided in Figure 12. The Symbolics machine is a commercial LISP machine which uses a demand paged virtual memory. For garbage collection, a modified version of Bakers algorithm is used where the garbage collector process is interleaved with the user program. Memory is divided into spaces and hardware support is provided for a *barrier* which exists between *oldspace* and other spaces. This barrier prevents the uncontrolled propagation of references to oldspace. The barrier operates in parallel with the user process unless a word is read from memory that contains an oldspace address. When that happens, a transport trap occurs and the memory location is replaced with the new address or the object is copied.

The ephemeral collector used by the Symbolics machine divides objects into three categories; static, dynamic, and ephemeral objects. Ephemeral objects are those objects that are assumed to be likely to become garbage soon after they are created. Each category is collected independently with the collector concentrating its effort on ephemeral objects. A table of references to ephemeral objects is maintained which is used as a root set for the ephemeral scavenge. The hardware barrier also maintains this table.

The statistics shown in Figure 12 for the Symbolics collector are for two test programs each of which are designed to run for approximately one hour. *Compiler* compiles a medium size file 100 times and *Boyer* is the kernel of a theorem-prover. The statistics for the Boyer program with no garbage collection were extrapolated.

Although the Symbolics machine uses a virtual memory which presents a different set of problems with regards to garbage collection, it is a machine where a mix of hardware and software is used to perform garbage collection. It that sense, it represents the state-of-the-art and data regarding the performance of its garbage collector can be used to evaluate the relative performance of the GMS. Even with the consideration that the simulation data may be optimistic, the information in Figure 12 shows that the G-machine garbage collection scheme can effectively manage the higher allocation rate with approximately the same relative performance of the ephemeral garbage collector.

Performance of Simulation Models			
Simulation Model	Number of Milliseconds	Relative Time	Allocation Rate/second
G-machine only	53.12	1.00	100K
Ref. count only	53.78	1.01	100K
Full traversal	73.46	1.38	100K

Performance of Symbolics Collector				
Benchmark Program	GC Type	Number of Seconds	Relative Time	Allocation Rate/second
Compiler	none	3,134	1.00	2.5K
	ephemeral	3,244	1.04	
Boyer	none	3,535	1.00	19K
	ephemeral	6,853	1.93	

Figure 12. Execution times of simulations and Symbolics garbage collector.

More revealing is data produced by simulation of each of the three models with varying allocation rates. Figure 13 shows the total number of simulated milliseconds required to execute 500K G-machine instructions. (If each G-machine instruction took only one cycle, 500K instructions would be executed in 50 milliseconds.)

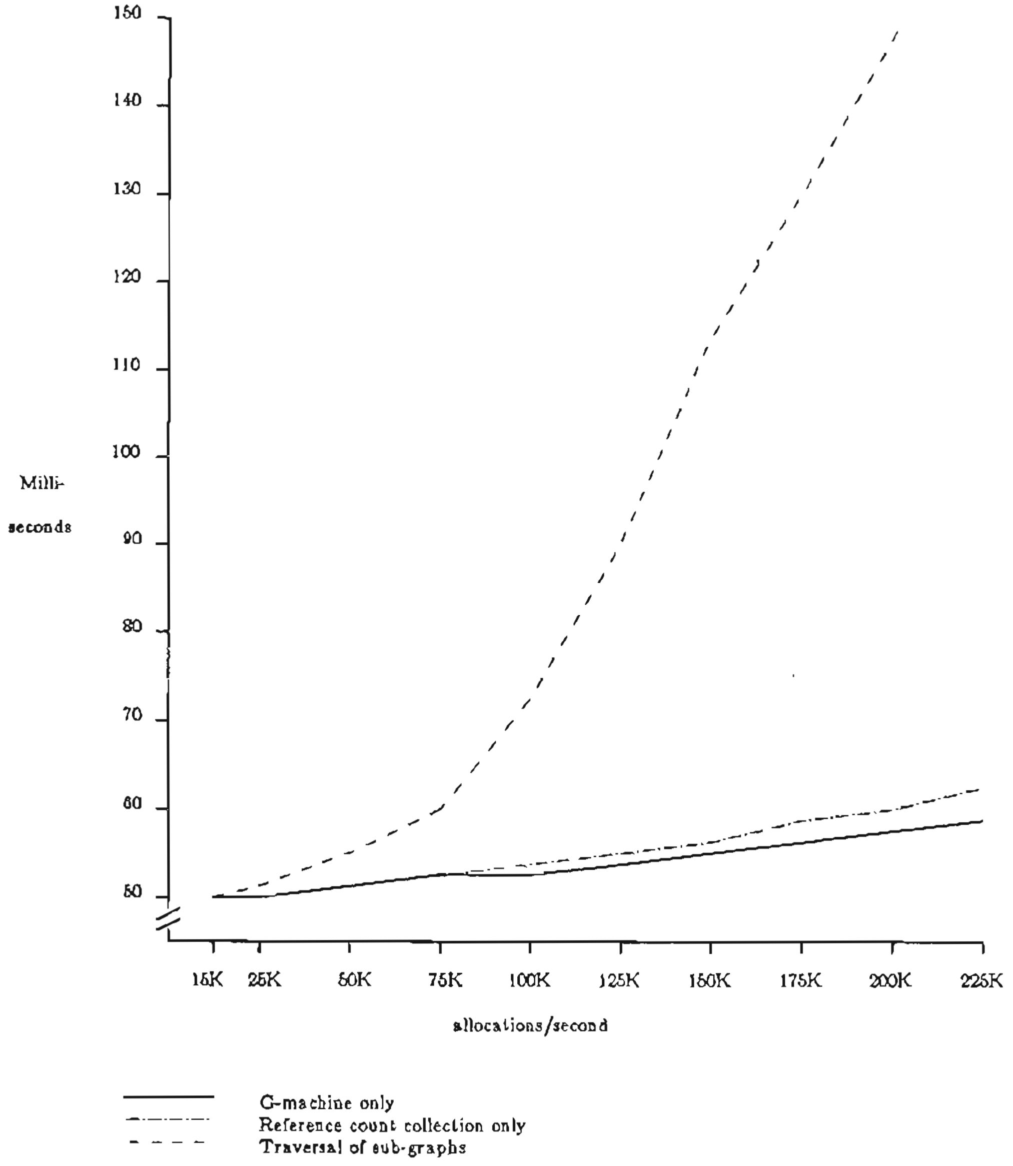

Figure 13. Execution time of 500,000 G-machine instructions.

As shown by the graph, the impact on the throughput of the G-machine of reference counting is relatively small. In addition, its impact on performance increases at a constant rate with respect to the allocation rate. This implies that the mechanism for maintaining reference counts is quite effective and interferes little with the processing of the G-machine.

On the other hand, the curve which represents the total execution time of the traversal model has a steep slope. The slope of the curve is different for allocation rates less than approximately 100K than for allocation rates greater than 100K. For allocation rates less than or equal to 75K nodes/second, the impact on the throughput of the G-machine of the graph traversal model can be attributed to resource contention. For allocation rates greater than 100K nodes/second, it is the workload of the host processor which becomes the dominant factor in defining the processing time. These conclusions were drawn from the following observations:

(1) For each simulation, the number of cycles that both the Tags and Graph memories were not servicing a memory request was recorded. This number was at a minimum when the allocation rate of 75K nodes/second was simulated (296K and 230K cycles for the Tags and Graph memories respectively). As the allocation rate increased beyond 75K per second, the number of free cycles for both memories increased as well. This increase was due to the fact that the host processor was not able to keep up with demand and the Signal Queue and the Garbage Can buffers were allowed to fill up. When this occurred, the G-machine was prohibited from accessing memory until these resources became available.

(2) For allocation rates less than 100K nodes/second, the average number of nanoseconds for the host to read from memory steadily increased from 479 to 513 nanoseconds. This increase could be attributed to the effect of resource contention. For allocation rates greater than 100K nodes/second, the average number of nanoseconds for a read from memory remained at about 519 simu-

lated nanoseconds.

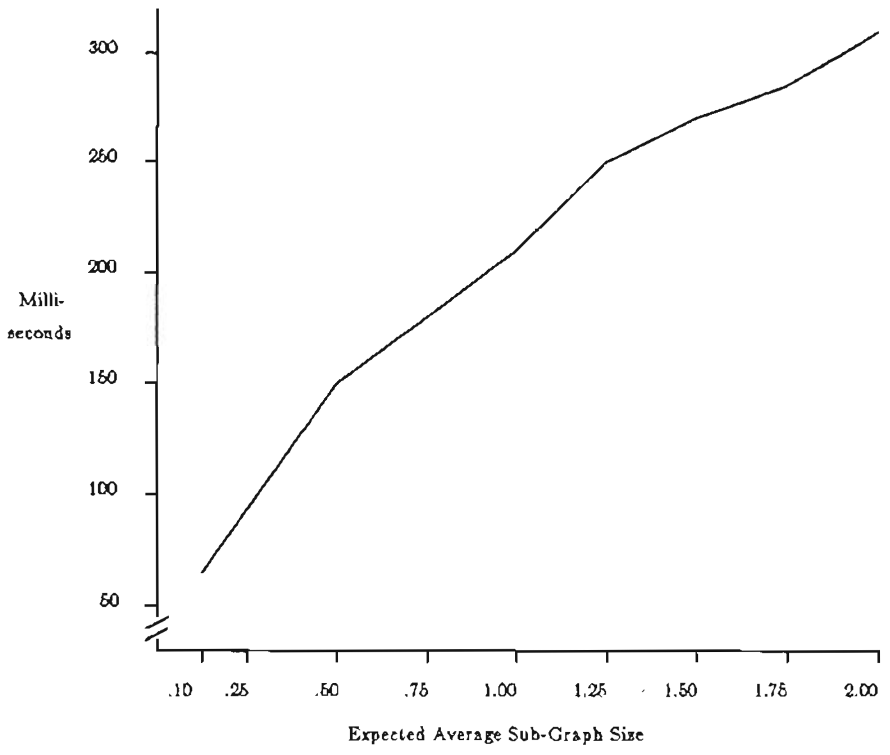

Figure 14. Execution time of 500,000 instructions using a Poisson distribution for sub-graph size probability.

In addition to the allocation rate of the G-machine, another aspect which is critical in defining the workload of the host processor is the size of the sub-graphs that are traversed. Figure 14 shows the number of cycles it took to "execute" 500K G-machine instructions for various sub-graph size probability distributions. The probability distributions were derived from Poisson distributions, and the probabilities were the same for both temporary and persistent nodes. Using a Poisson distribution to obtain probabilities and having the same probability for both types of nodes is an oversimplification of the model. However, this data highlights the sensitivity of the design to this aspect of the garbage collection scheme.

Optimization

Simulation results show that the performance of the design and the garbage collection scheme is sensitive to the workload of the host. The hosts workload is, in turn, sensitive to the size of the sub-graphs that are traversed. Therefore, if the number of sub-graphs requiring traversal were reduced, the memory management scheme would perform better at higher allocation rates.

One optimization that has been suggested in [Kie86] is the use of a *threshold bit*. The threshold bit is an additional graph node tag that is set when a node is first allocated. If the threshold bit is set to one, then the node might be a root of a cyclic sub-graph. When the host examines a node for collectibility, it tests the value of the threshold bit. If the reference count of the node is not zero, then the host decides if traversal is necessary based on the value of the threshold bit. If only a small fraction of all nodes are expected to be roots of cyclic structures, then the reduction in the number of sub-graph traversals would be significant.

In addition, adding the threshold bit to the design is simple. An additional bit is added to the G-machine instruction alloc. This bit indicates the value to which the threshold bit should be set for the newly allocated node (all recursive functions would be identified by the compiler as potentially cyclic). When the G-machine emits an alloc signal, the ref-chip will test the operand bit and set the threshold bit for the newly allocated node in Tags memory. No additional memory accesses are required by the host to read the *threshold* bit. It merely reads the *threshold* bit along with the reference count at the beginning of its collection processing.

Figure 15 shows the same information as Figure 13 except that data for the graph traversal model includes thresholding. For these simulations, twenty percent of the nodes examined by the host for collectibility have their threshold bit set.

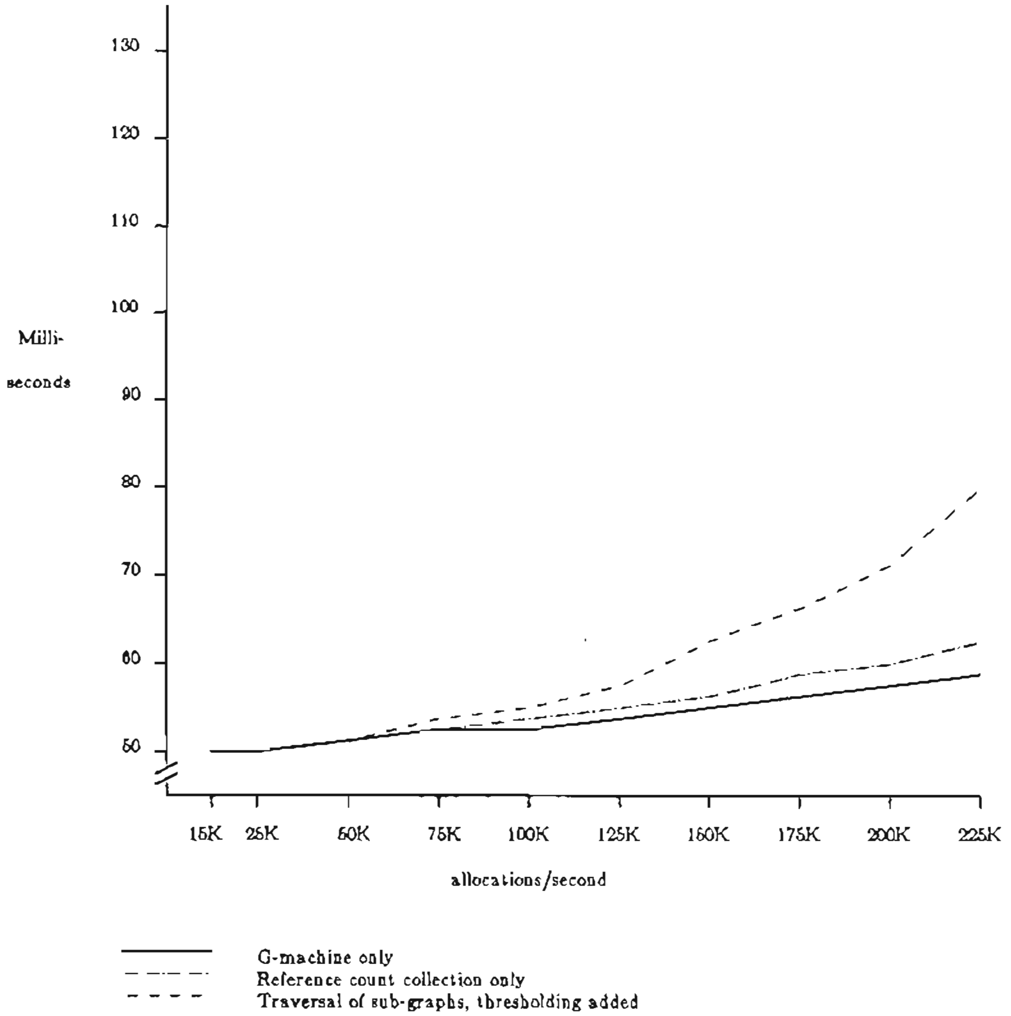

Figure 15. Execution time with thresholding added. 20% of the nodes have the threshold bit set.

The addition of one bit, the threshold bit, has a significant impact on the performance of the design. The workload of the host processor is reduced so that it is able to handle allocation rates greater than 100K nodes/second. In addition, since fewer sub-graphs are traversed, the performance would become less sensitive to variations in the sizes of the sub-graphs. Adding the *threshold* bit would certainly make the memory management scheme more robust and able to handle programs with a variety of memory management demands.

SUMMARY AND CONCLUSIONS

A dual-ported real memory architecture that meets the requirements of the G-machine has been designed and evaluated. Since the G-machine is expected to have a high allocation rate, a main requirement of its memory is that garbage collection be fast, which is accomplished best by having a parallel collection process. Therefore, a concurrent, modified reference counting algorithm was used. As a result, both the memory requirements of the G-machine and those of the collector were considered in the development of a design.

The architecture of the G-machine and the manner in which it evaluates a graph was also considered in the development of a memory management scheme. The fact that the G-machine only overwrites the contents of a node at the end of a reduction is exploited by the trash instruction. The trash instruction simplifies the number of operations required to maintain reference counts. Also, a stack allocation scheme was included in the design so that graph references from the G-machine traversal stack need not be maintained.

Once the data and the memory requirements of the host and the G-machine were defined, the memory architecture could be addressed. The design included three hardware buffers that queue the information passed between various units during processing. This buffering reduced the number of wait states and allowed the units to continue processing at their own pace (if the buffers do not become full).

The memory itself was made up of banks of dynamic RAMs which are controlled by Intel's 8207 Advanced Dynamic RAM Controller. The 8207 provides support for two different busses to independently access memory, creating a dual-ported memory. An addressing scheme was included to support a tagged memory and the individually accessible data fields of the graph nodes. Using an encoded addressing scheme, a 32-bit address supported 4 memory modules, or approximately 40 Mbytes of real memory.

The graph node was divided into two separately accessible areas; data which is used to represent the expression graph was stored in the Graph memory, and data that was used only for memory management was stored in Tags memory. Storing data in this manner reduced the contention for memory between those memory accesses required by the G-machine for evaluation, and those used in memory management.

Off-the-shelf components could be used to realize the memory system, with the exception of one custom VLSI component. The function of that component was to maintain the reference counts based on the memory instructions of the G-machine. This component exploited the fact that the graph node was divided into two separately accessible areas and had two ports; one which accessed Graph memory, and the other which provided access to the Garbage Can and Tags memory. Therefore, much of the processing required to maintain reference counts could be done in parallel with memory accessed by the G-machine.

Simulation of the design showed that it is effective in supporting the expected high allocation rate of the G-machine. In particular, the use of hardware supported reference counting had little impact on the throughput of the G-machine. Given the assumptions regarding the subgraph size and the workload, the cycle detection portion of memory management was effective as well. The design was, however, sensitive to high allocation rates and variations in sub-graph size. In order to make the memory management scheme more robust, the use of a threshold bit to aid in identifying potential cyclic structures was studied and found to be effective.

Further Research

A portion of the design that requires validation through a more detailed analysis is the collision detection mechanism of the reference counting hardware. It is unclear that the hosts bus controller will be able to provide a signal in time to insure that a collision does not occur. This aspect of the design should be explored to determine if external "glue logic" is adequate for signal synchronization, or if the reference chip design needs to include its own synchronization mechanisms.

Perhaps the most exciting outcome of this research is that this memory management scheme has practical real-time list-processing potential. The memory can be realized in mostly off-the-shelf components, so the cost of building such a system should not be excessive. The most fruitful research would be an analysis of the design to see if it can be used for different kinds of processors and environments. Are the portions of the design that tailor it to the G-machine intrinsically tied to the performance of the system and its feasibility? Are there similar situations on a LISP processor where an instruction like trash can be used? Can the scheme be used in a virtual memory system? The possibilities certainly warrant further research.

REFERENCES

[Adv84]

Advanced Dynamic RAM Controller (i8207), *Memory Components Handbook*, Intel Corporation, 1984, pp. 3-295 through 3-340.

[Bak78]

Baker, H. G., List-Processing in Real Time on a Serial Computer, *Communications of the ACM*, 21, 4 (April 1978), 280-294.

[Bur80]

Burton, R.B., Masinter, L.M., Bobrow, D.G., Haugeland, W.S., Kaplan, R.M., Sheil, B.A., Overview and Status of Dorado Lisp, *Proceedings of the 1980 LISP Conference*, Stanford, California, 1980, pp. 243-247.

[Coh81]

Cohen, J., Garbage Collection of Linked Data Structures, *Computing Surveys*, Vol. 13, No. 3, September 1981.

[Daw82]

Dawson, J. L., Improved Effectiveness from a Real Time LISP Garbage Collector, *Conf. Rec. 1982 Symposium on LISP and Functional Programming*, 1982, pp. 159-167.

[Deb84]

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

[Deu76]

Deutsch, P. L., Bobrow, D. G., An Efficient, Incremental, Automatic Garbage Collector, *Communications of the ACM*, vol. 19, September, 1976, pp. 522-526.

[Fir84]

First-In First-Out (FIFO) Cascadable Memory C67401, *Bipolar LSI Data Book*, Monolithic Memories, 1984, pp. 9-8 through 9-19.

[Fos85]

Foster, M. H., Design of a List-structure Memory using Parallel Garbage Collection, Master's Thesis, Oregon Graduate Center, Beaverton, OR, September 1985.

[GAH86]

G-Machine Architecture/Programmers Handbook, Oregon Graduate Center, February 1986. Unpublished document.

[Hic84]

Hickey, T., and Cohen, J., Performance Analysis of On-the-Fly Garbage Collection, *Communications of the ACM*, vol. 27, November, 1984, pp. 1143-1154.

[Hud82]

Hudak, P., Keller, R. M., Garbage Collection and Task Deletion in Distributed Applicative Processing Systems, *Conf. Rec. 1982 Symposium on LISP and Functional Programming*, 1982,168-178.

[Int84]

Intel Microsystems Components Handbook, Intel Corporation, 1984.

[Joh83]

Johnsson, T., *The G-machine -- an abstract architecture for graph-reduction*, Dept. of Computer Sciences, Chalmers University of Technology, Gothenburg, Sweden, 1983.

[Kie85A]

Kieburtz, R. B., The G-machine: A fast-graph-reduction evaluator, Tech. Rep. CS/E-85-002, Oregon Graduate Center, Beaverton, OR, January, 1985.

[Kie85B]

Kieburtz, R. B., Control of the Instruction Fetch Unit, Oregon Graduate Center, April, 1985. Unpublished document.

[Kie86]

Kieburtz, R. B., Incremental collection of dynamic, list-structure memories, Tech. Rep. CS/E-85-008, Oregon Graduate Center, Beaverton, OR, January, 1986.

[Knu68]

Knuth, D. E., in *The Art of Computer Programming, Vol 1 -- Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968.

[Lie83]

Leiberman, H., Hewitt, C., A Real-Time Garbage Collector Based on the Lifetimes of Objects, *Communications of the ACM*, Vol. 26, Number 6, June 1983, pp. 419-429.

[Moo84]

Moon, D., Garbage Collection in a Large LISP System, *Conf. Rec. 1984 ACM Symposium on Lisp and Functional Programming*, 1984, pp. 235-246.

[Mot85]

MC68020 32-Bit Microprocessor User's Manual, Second Edition, Motorola, 1985.

[Ran86A]

Rankin, L. J., Architectural and Functional Specification for the G-Machine Instruction Fetch and Translation Unit, Oregon Graduate Center, January, 1986. Unpublished document.

[Ran86B]

Rankin, L. J., Kuo, S., Micro-Instruction Dispatch and Decoding, Oregon Graduate Center, Beaverton, OR, May 1986. Unpublished document.

[Rig84]

Righter, W. H. and Altnether, J. P., Designing Memory Systems for Microprocessors Using the Intel 2164A and 2118 Dynamic RAMS, *Intel Memory Components Handbook*, Intel, 1984.

[Ste75]

Steele, G. L., Multiprocessing Compactifying Garbage Collection, *Communications of the ACM*, vol. 18, 9 (Sept. 1975), pp. 495-508.

[Tea86]

Teamwork Concurrent Programming Toolkit, User's Manual, Version 1.0, Block Island technologies, Portland, Oregon, 1986.

[Tex84]

TMS4256 262,144-Bit Dynamic Random-Access Memories, *Supplement to MOS Memory Data Book*, Texas Instruments, 1984, pp. 2-27 through 2-47.

[Tri78]

Trivedi, K.S., Analytic Modeling of Computer Systems, *IEEE Computer*, October, 1978, pp. 38-56.

[Ung84]

Ungar, D., Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm, *ACM SIGSOFT/SIGPLAN Practical Programming Environments Conference*, April 1984, pp. 157-167.

[Wad76]

Wadler, P. L., Analysis of an Algorithm for Real Time Garbage Collection, *Communications of the ACM*, vol. 19, 9 (Sept. 1976), pp. 491-500.

[Whi80]

White, J., Memory Management in a Gigantic LISP Environment, or GC Considered Harmful, *Proc. 1980 LISP Conference*, Stanford, CA, pp. 119-127.

[Wis77]

Wise, D.S., and Friedman, D.P., "The one-bit reference count," *BIT*, vol. 17, 4 (1977), pp. 351-359.

[Wis85]

Wise, D.S., Design for a Multiprocessing Heap with On-board Reference Counting, Technical Report No. 163, revised, Indiana University, Bloomington, IN, July, 1985.

APPENDIX A: DESIGN SPECIFICATIONS FOR THE REFERENCE CHIP

The main function of the Reference Chip (ref-chip) is to maintain reference counts and to pass potentially collectible nodes to the Garbage Can. The signals the ref-chip receives from the G-machine determines the sequence of operations the ref-chip performs. For the G-machine instruction write, reference counts may be incremented and for the trash instruction, reference counts may be decremented. If the node is marked as persistent when its reference count is decremented, the ref-chip passes its address to the garbage can.

A key to the ref-chip design is that it is synchronous to the G-machine and memory. A synchronous design simplifies the internal logic with respect to interfacing with other components and is the most efficient in terms of the number of wasted "wait states" cycles. This requires that the G-machine, memory, and ref-chip all operate at the same frequency of 10 MHZ. This 10MHZ clock is divided into two non-overlapping phases, $\phi 1$ and $\phi 2$.

Because the design is synchronous to the G-machine and memory, signals to and from the G-machine are sent and received during one phase of the clock, and memory signals are sent and received during the other phase. To aid in explanation of the functions of the ref-chip, G-machine signals are assumed to be valid during $\phi 1$, and memory signals are valid during $\phi 2$.

Inputs and Outputs

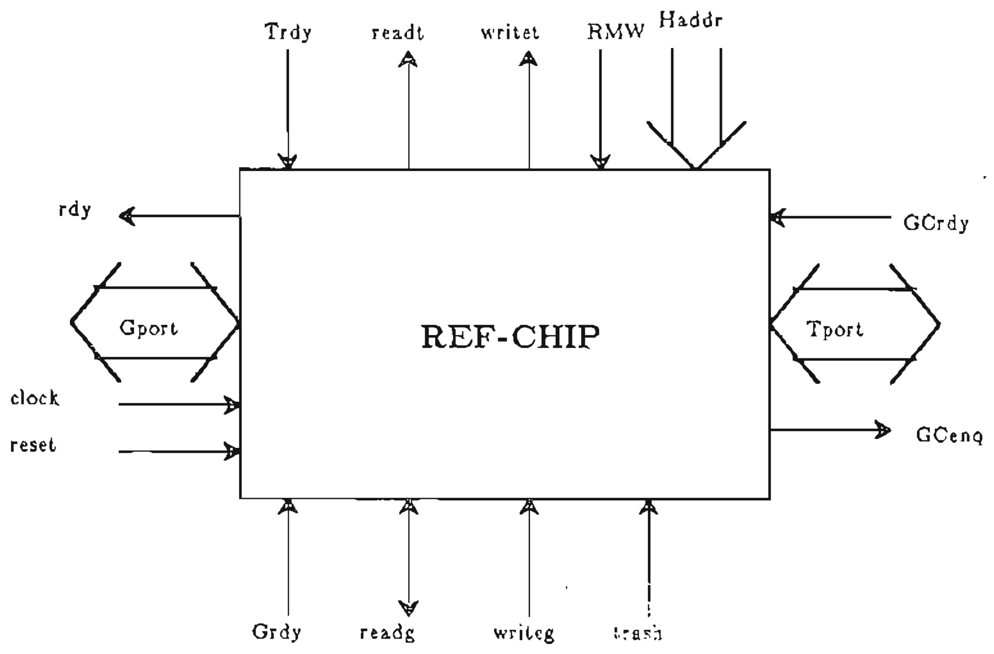

Figure A1. Ref-chip inputs and outputs.

Figure A1 shows the inputs and the outputs of the ref-chip. The following explains each of these signals:

(1) *clock*. The clock runs at a frequency of 10 MHZ. The two non-overlapping phases, $\phi 1$ and $\phi 2$, are generated internally.

(2) *reset*. Initializes all internal states and registers to place the chip in a known state.

(3) *Grdy*. This signal is the memory ready signal output by that portion of memory called the Graph memory. The Grdy signal is used to determine the value of the internal state "graph memory ready." This signal is valid during $\phi 2$.

(4) *readg.* The majority of the time, readg is an input signal from the G-machine that is sampled every cycle during $\phi 1$. If the value of readg indicates that the G-machine is accessing memory, then the ref-chip sets its internal state "graph memory ready" to false. The only time that readg is used as an output is when the ref-chip needs to read the contents of a node address that was the argument of a trash instruction. When this occurs, the G-machine is prohibited from accessing memory (and using this signal) by the value of the ref-chip rdy signal. The ref-chip can then use the readg signal to access Graph Memory.

(5) *writeg.* Writeg is valid during $\phi 1$ and sampled every cycle. When the value of the signal indicates that the G-machine is conducting a write to Graph memory, it is also stored as an instruction to be processed by the ref-chip. When the ref-chip receives a write instruction it sets its "latch data" (data will be latched on the next cycle) and "graph memory ready" (graph memory not ready) states.

(6) *trash.* Trash is the signal emitted by the G-machine to indicate the contents of a node are about to be overwritten and is valid during $\phi 1$. The ref-chip stores the trash to be processed later and sets its "latch data" state (the node address will be latched on the next cycle) and its own "ready" state to false.

(7) *GCenq.* GCenq is the signal used by the ref-chip to add a node to the Garbage Can. The ref-chip also sets its "garbage can ready" state to false at this time. Like all memory related signals, this signal is valid during $\phi 2$.

(8) *Tport.* Tport is a 32-bit wide address/data bus which provides access to the Tags memory and the Garbage Can. Contents of this bus are valid during $\phi 2$.

(9) *GCrdy*. The ref-chip samples the *GCrdy* signal every cycle during $\phi 2$ and its value is used in determining the "garbage can ready" state.

(10) *Haddr*. Haddr provides access to the hosts address/data bus. If the ref-chip is not in a "detect collision" state, then the contents of this port are latched every cycle during $\phi 1$. The number of host address bits latched by the ref-chip can be varied, but the probability of a "detect collision" state occurring is directly proportional to the number of address bits that are sampled.

(11) *RMW*. This signal is output by the host processor and indicates that the host is entering a read-modify-write sequence. This signal is sampled every $\phi 1$, and the value of this signal sets the internal "detect collision" state. The Haddr and the RMW signals are the only signals that are asynchronous to the ref-chip. The current design relies on external logic to insure that the signals are stable before they are seen by the ref-chip (the host bus-controller would perform this function). However, it is uncertain that the ref-chip will see the signals in time to insure that a collision will not occur. When the host/memory interface is defined in more detail, then this aspect of the design should be reexamined to determine whether external logic is adequate. Otherwise, the ability to handle asynchronous logic should be included in the ref-chip design.

(12) *writet*. The ref-chip uses this signal to write to Tags memory.

(13) *readt*. The ref-chip uses readt to read from Tags memory. Use of both the writet and readt signals affects the "tags memory ready" state.

(14) *Trdy*. The Trdy signal indicates the ready status of Tags memory. This signal is valid during $\phi 2$ and is used to set the "tags memory ready" state.

(15) *rdy*. The ref-chip has its own status signal that is sampled by the G-machine (rdy is to be combined with status signals output by other memory components). There are two conditions where the ref-chip emits a "not ready" signal; the first is when its buffer of G-machine instructions is full, and the second is when it has received a trash instruction. This signal is valid during $\phi 1$.

(16) *Gport*. The Gport is a 33-bit wide address/data bus. The ref-chip latches data from this bus one cycle after it receives the G-machine signals write or trash. The ref-chip also uses the Gport when reading from Graph memory. The Gport is the only port that is tri-stated so that the port can be disabled when not in use.

Functional Description

The internal logic of the ref-chip can be divided into two main areas of functionality. The first manages the queueing of instructions received from the G-machine and the latching of the contents of the G-machine address/data bus. This portion of the logic will be referred to as the G-Machine Interface Logic.

The instruction queue can only hold two instructions at a time; the *current* instruction, and the *next* instruction. As instructions are received from the G-machine they are placed in this queue, always filling the current instruction first. When the current instruction has been removed from the queue, the next instruction is advanced to take its place.

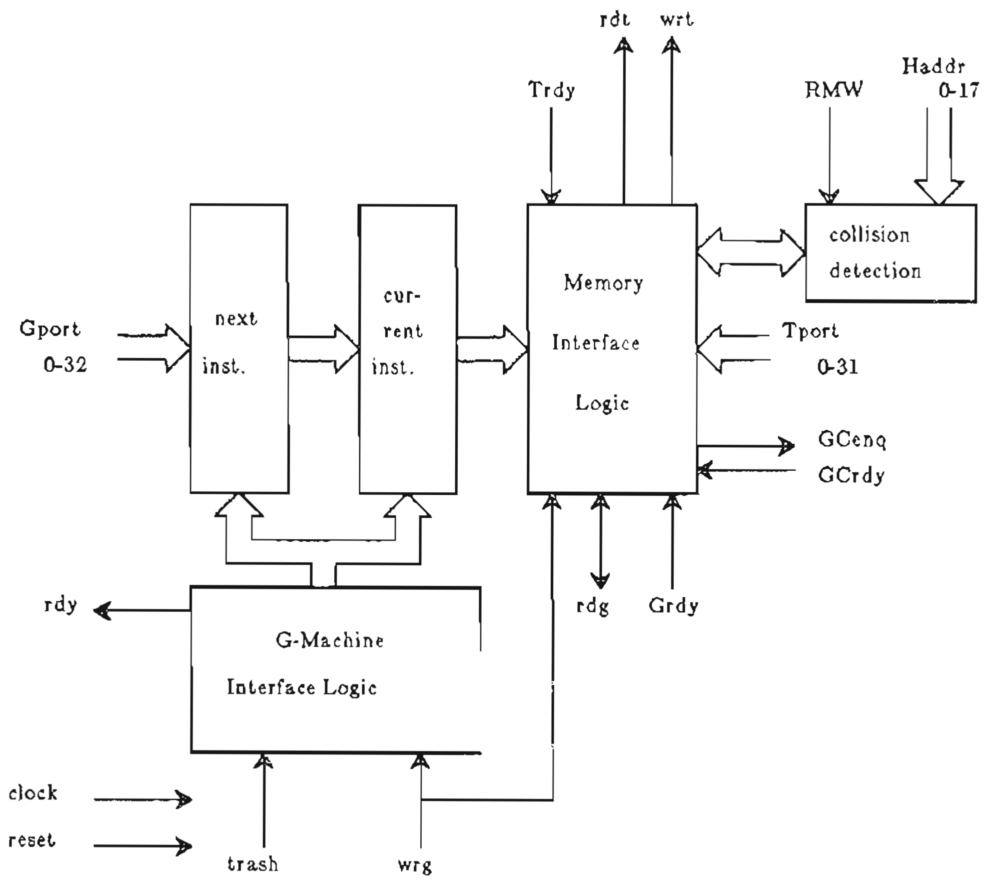

Figure A2. Ref-chip block diagram.

When an instruction is received from the G-machine, the G-machine Interface Logic sets a "latch data" state to indicate that data will be latched from the G-machine address/data bus on the next cycle. For the write instruction, the contents of the bus will be the data being written by the G-machine, and for trash it will be the address of the node to be overwritten. When the data is latched it is placed in the queue along with the instruction. Once the data has been latched, the instruction is marked as valid and ready for processing.

The G-machine Interface Logic also maintains on a cycle-by-cycle basis the ready state of the ref-chip. This ready state is a function of the type of instructions in the queue and whether both current and next are occupied. If the queue of instructions is full the ready state is set to false. In addition, when a trash instruction is placed in the current position of the queue, the ready state is again set to false. The ready state remains false until the ref-chip reads the contents of the node to be trashed from graph memory. To minimize the number of G-machine wait states, the ready status of the ref-chip should be emitted immediately (during the same phase) after the receipt of a G-machine instruction.

The second part of the internal logic, the Memory Interface Logic, controls the actions required to process each G-machine instruction. This logic mostly sequences through a series of memory accesses based upon the value of the instruction and the data that has been read from memory. Four internal states are maintained on a cycle-by-cycle basis which are used to determine the actions of the sequencer: 1) graph memory ready, 2) tags memory ready, 3) garbage can ready, and 4) detect collision.

If the current instruction is a write, the data that was latched from the G-machine address/data bus for the instruction is tested to see if it is a pointer. If it is a pointer, then a read-modify-write sequence is entered. Otherwise, since no action is required, the Memory Interface Logic is finished with the current instruction and it is removed from the queue.

If the contents of the current instruction is a trash, Graph memory is accessed to obtain the contents of the node to be overwritten. The address used to access memory is the data that was placed in the instruction queue. Again, if the node is a pointer, a read-modify-sequence is entered. Otherwise, the instruction is removed from the queue.

During a read-modify-write sequence, the ref-chip accesses Tags memory only. The data that are read from Tags memory includes the reference count, forever_uncollectible bit, and the persistent_bit. For a trash instruction the reference count is decremented, for a write instruction it is incremented. If the reference count overflows, the forever_uncollectible bit is set. After the reference count has been modified, the new reference count, forever uncollectible bit, and a logic value of one for the recently_visited bit are written back out to tags memory. If the current instruction is a trash, the persistent_bit and forever_uncollectible bit are tested. If the node is persistent and its forever_uncollectible bit is not set, the Memory Interface Logic emits the proper signals to add the node to the Garbage Can (once the garbage can is ready). The Memory Interface Logic then removes the current instruction from the queue making room for the next instruction.

If during the read-modify portion of a read-modify-write sequence the ref-chip enters a "detect-collision" state, the Memory Interface Logic is interrupted. At this point the address being used for Tags memory access must be compared to the address the host is using to access Tags memory. If the addresses are not equal, then the ref-chip can continue. Otherwise, the ref-chip will re-start the read-modify-write sequence when a signal is received that the host has completed.

While the configuration of memory and the "glue" logic required to interface with it will ultimately define the signal requirements for accessing memory, the steps required for a memory access that have been assumed for this design are as follows;

(1) *read from memory.* If memory is ready, the ref-chip emits the read signal and places the address on the address/data bus at the same time. The memory ready state at this time is set to false as well. The ref-chip waits one cycle before testing the memory ready signal and must continue waiting until it receives a signal that memory is ready. Once it receives a memory ready signal it can latch the contents of the address/data bus.

(2) *write to memory.* If memory is ready, the ref-chip emits the write signal and places the address on the address/data bus at the same time. One cycle later, the data to be written is placed on the address/data bus. At that point the memory ready state is set to false. The ref-chip does not have to wait for memory to be ready to continue processing, but it must continue to test the memory ready signal to reset its internal memory ready state.

APPENDIX B: MODELING AND SIMULATING THE ARCHITECTURE

The architecture is simulated to validate the design and measure the impact of memory contention upon the throughput of the G-machine. Detailed simulation of the graph and the garbage collection algorithm is not necessary, so the activities of the host and the G-machine are parameterized. The hardware timing is modeled accurately, however.

To explore the performance of different models and configurations, several parameters can be specified. These parameters are:

(1) Number of G-machine RISC instructions that access memory or are processed by the host. This number is expressed as an allocation rate per second.

(2) Several simulation models are available, some of which are designed to provide benchmark information. Benchmark models are a) G-machine only, b) G-machine and reference count maintenance only, and c) reference count garbage collection (no attempt to identify and collect cyclic structures). Other models include simulation of the full incremental garbage collection algorithm, and the addition of thresholding.

(3) The Intel Advanced Dynamic Controller (i8207) has two algorithms for arbitration of memory requests, *Port A Priority* and *Most Recently Used*. Either of these algorithms can be simulated.

(4) The number of nodes visited for each sub-graph traversed by the collector are derived from probability distributions. These distributions can be based on the data generated from [Fos85] simulations or a Poisson distribution.

(5) One or two memory modules can be used in the simulation. When two memory modules are used, it is assumed that addresses are low-order interleaved between the modules.

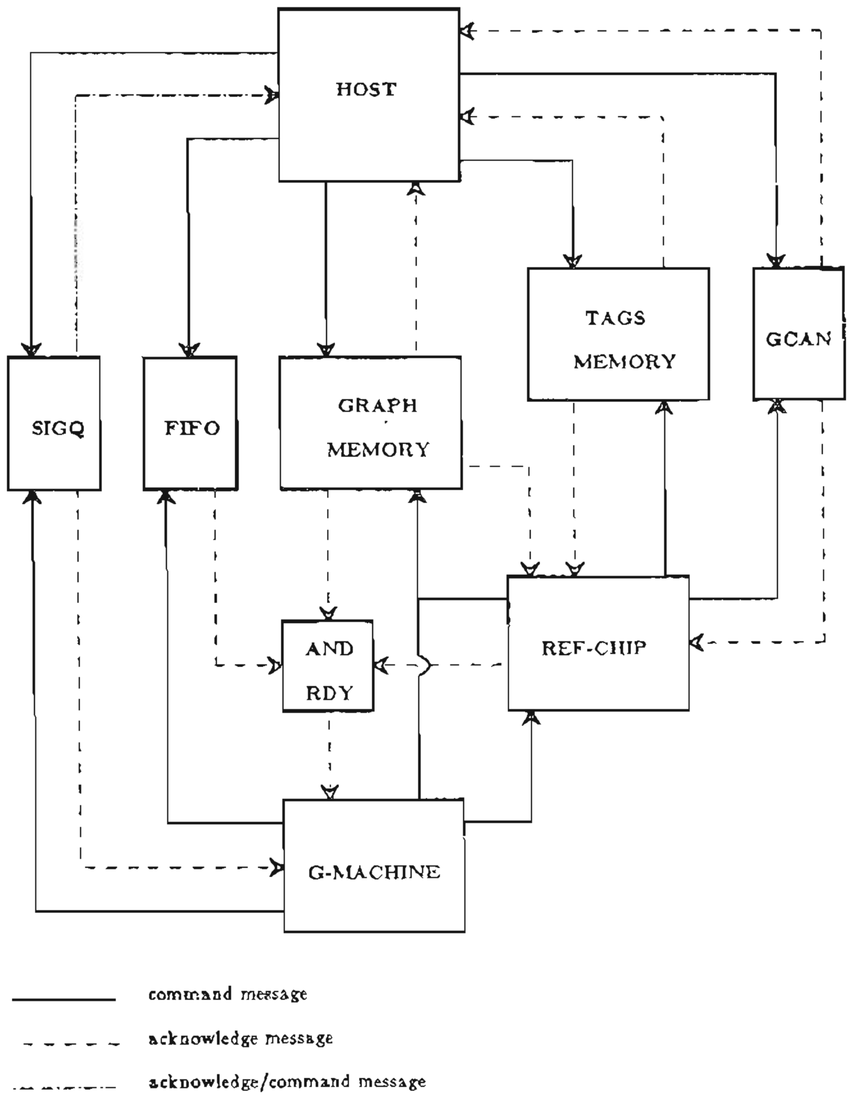

Figure B1. Functional simulation blocks.

The simulation consists of nine tasks, one for each of the functional blocks shown in Figure B1. Two additional memory tasks, one each for the Graph and Tags memories, can be added to simulate two memory modules. Two kinds of messages are passed between tasks representing data or commands emitted during processing, or status or acknowledge signals.

With the exception of the host task, all tasks in the simulation are "synchronous" with their processing time relative to a simulated 10 MHZ clock cycle. The host clock speed is simulated to be 16 MHZ. System clocks and the processing time of the hardware are simulated by suspending the tasks for regular intervals. Tasks are synchronized by the simulated clocks and the messages passed between tasks.

The remainder of this appendix describes each of the nine tasks and how they modeled the architecture. For simplicity, only one simulation model is described; the model which simulated the full garbage collection algorithm. Only small changes to this basic model are required to obtain the other models that are simulated.

The G-machine Task

The G-machine task is the "main" process of the simulation in that the length of the simulation is defined by the number of G-machine RISC instructions to be "executed". The instructions generated by the G-machine are *read, write, alloc, trash, call, return,* and *other*. All *other* instructions are assumed to be single cycle instructions. The frequency of each of these instructions is defined by a probability distribution and the "allocations per second", a simulation parameter.

G-machine instructions to be processed by other units are sent as command messages at the end of its clock cycle, and are read by the receiving task at the beginning of the next clock cycle. The G-machine also maintains an internal state which holds the ready status of memory and the signal queue. The following pseudo-code describes the G-machine task algorithm:

```
begin /* G-Machine */
   wait one clock cycle
   check for memory acknowledge message
   check for signal queue acknowledge message
   generate command using allocation rate
   if command goes to host then
        /* call, return or alloc */
        while (signal queue is not ready) wait one cycle
        send command message to signal queue
        set state: signal queue not ready
   endif
   if command goes to memory then
        /* read, write, trash or alloc */
        while (memory is not ready) wait one cycle
        case (command)
           write:  send command message to ref-chip & graph memory
                set state:  memory not ready
           trash:  send command message to ref-chip
                set state:  memory not ready
           alloc:  send command message to fifo
                set state:  memory not ready
           read:   send command message to ref-chip & graph memory
                set state:  memory not ready
                while memory not ready
                   wait one cycle
                   check for acknowledge message from memory
                   update state
                end
        endcase
   endif
   go to begin
end /* G-Machine */
```

The conditions that place the G-machine task in a wait state are 1) when the G-machine has a command that goes to the host and the signal queue is not ready, 2) when the G-machine is reading from memory, and 3) the G-machine has a memory command and memory is not ready.

The And-Rdy Task

The and-rdy task reads the acknowledge messages coming from the fifo, graph memory and the ref-chip. The and-rdy task sends an acknowledge message to the G-machine which simulates the G-machine "memory ready" signal. The value of this "memory ready" signal is the logical and of the acknowledge messages the and-rdy task has received.

The Ref-Chip Task

The ref-chip task simulates the custom VLSI reference counting chip which is described in detail in Appendix A. Since the ref-chip interfaces with both the G-machine and memory tasks, both phases of the 10 MHZ clock cycle are simulated (the G-machine sends messages during $\phi 2$, the memory tasks and queues during $\phi 1$). Like the G-machine task, the ref-chip task maintains internal states regarding the status of memory and the garbage can. Additional state information is stored in an array of two records where *current* and *next* are indices into the array. This array stores the G-machine commands and holds the state information regarding how much processing had been completed for each instruction. If the array is either full or empty, then current and next are equal. The following pseudo-code shows how the ref-chip is simulated:

```
signal_memory(command)
   begin
      case (command)
           readt:  send command message to tags memory
                set state: tags memory not ready
           write:  send command message to tags memory
                set state: tags memory not ready
           readg:  send command message to graph memory
                set state: graph memory not ready
      endcase
   end /* signal_memory */

get_next_instruction()
   begin
      /* data for next instruction (if it exists) already latched */
      reg[next].state = latch
      current = (current + 1) mod 2
      if reg[current].ins != trash then
         set state: ready = true
   end /* get_next_instruction */

begin /* ref-chip */
   /* $\phi$1 processing */
   wait one phase
   if command message from G-machine then
        case (command)
           write:  set state: graph memory not ready
                reg[next].ins = write
                reg[next].state = init
                next = (next + 1) mod 2
           trash:  set state: ready = false
                reg[next].ins = trash
                reg[next].state = init
                next = (next + 1) mod 2
           read:   set state: graph memory not ready
        endcase
      if reg[current].ins and reg[next].ins are valid then
           set state: ready = false
   endif
   if ready state has changed then
        send acknowledge message to and-rdy

   /* $\phi$2 processing */
   wait one phase
   check for tags memory acknowledge message
   check for graph memory acknowledge message
   check for garbage can acknowledge message
```

```
case (reg[current].ins)
    write:
        case (reg[current].state)
            init: /* one cycle before data can be latched */
                reg[current].state = latch
            latch:
                if data is a pointer then
                    /* read reference count */
                    if tags memory is ready then
                        signal_memory(readt)
                        reg[current].state = readt
                    else
                        reg[current].state = wait
                    endif
                else
                    /* data is not a pointer */
                    get_next_instruction()
                endif
            wait:
                if tags memory is ready then
                    signal_memory(readt)
                    reg[current].state = readt
                endif
            readt:
                if tags memory is ready then
                    /* send write signal with address */
                    /* increment data */
                    signal_memory(write)
                    reg[current].state = incr
                endif
            incr: /* data is sent to memory */
                /* ready for next instruction */
                get_next_instruction()
        endcase
    trash:
        case (reg[current].state)
            init: /* one cycle before data can be latched */
                reg[current].state = latch
            latch: /* read node from graph memory */
                if graph memory is ready then
                    signal_memory(readg)
                    ins[current].state = readg
                endif
```

```
                    readg: /* latch data and test is-pointer bit */
                      if graph memory ready then
                           /* G-machine can now use graph memory */
                           set state: ready = true
                           if data is a pointer then
                              /* read the reference count */
                              if tags memory is ready then
                                   signal_memory(readt)
                                   reg[current].state = readt
                              else
                                   reg[current].state = wait
                                endif
                           else get_next_instruction()
                      endif
                    wait:
                         if tags memory is ready then
                            signal_memory(readt)
                            ins[current].state = readt
                         endif
                    readt:
                         if tags memory is ready then
                            /* decrement data */
                            signal_memory(write)
                            reg[current].state = decr
                         endif
                    decr:  /* data is decremented & sent to memory */
                         reg[current].state = enq

                    enq:
                         if garbage can is ready then
                            send enqueue signal to garbage can
                            set state: garbage can is not ready
                            get_next_instruction()
                         endif
                 endcase
         endcase
         go to begin
      end
```

The ref-chip emits "ready" and "not ready" acknowledge messages which are "anded" with the ready status of the fifo and memory tasks by the and-rdy task. Therefore, a "not ready" status of the ref-chip is seen by the G-machine task as "memory not ready" and prohibits the G-machine task from executing any alloc, read, write, or trash instructions. Like the actual hardware, there are three situations when this can occur. The first when the ref-chip receives a trash command from the G-machine. When this happens, the ref-chip does not send a "ready" acknowledge message until it has completed its graph memory access.

The second situation where the ref-chip would send a "not ready" acknowledge message is when the ref-chip is processing a write or a trash and has received another write or trash from the G-machine. In that case, both current and next elements of the array are full, and the G-machine must not send another command message until the ref-chip is able to receive it.

The third situation occurs when there is a bottleneck in the garbage collection process. If the host task is unable to keep up with the garbage collection demands, the garbage can queue fills up. When that occurs, the ref-chip can no longer enqueue any new garbage nodes and therefore process any new instructions it receives from the G-machine.

Not included in the simulation is the logic related to read-modify-write collisions with the host processor. The ability to detect collisions only affects the performance of the ref-chip when a collision occurs. With one memory module the chance of a collision occurring is assumed to be about one in a million (each module holds one million graph nodes). Therefore, it was concluded that adding the logic to the simulation would not significantly affect the data.

The Tags and Graph Memory Tasks

All memory processes simulate the behavior of the dual-ported memory and are based on the specifications of the Intel Advanced Dynamic RAM Controller i8207 [Adv84] and Texas Instruments 256k dynamic random-access memories [Tex84]. The main functions of the i8207 are to arbitrate memory requests between two ports

(PortA and PortB), provide control for dynamic RAMs, generate refresh control given a refresh period, and provide acknowledge signals. The simulation model has the same functionality with the exception of providing explicit DRAM control. DRAM access is modeled as a number of cycles before the next memory request can be serviced.

Arbitration

Each memory task has three ports from which memory requests can originate: 1) PortA, the synchronous port, receives command messages from the G-machine and the Ref-chip, 2) PortB, the asynchronous port, receives command messages from the Host, and 3) PortC is the internal port where refresh requests are generated. These ports are checked for messages every simulated cycle. Reflecting the specifications of the i8207, arbitration between outstanding requests is called as early as two cycles after the start of a memory cycle. Arbitration is repeated every cycle thereafter until a new memory cycle begins. The i8207 arbitration algorithm used in the simulation, either *PortA priority* or *Most Recently Used*, can be specified as a simulation parameter.

For the i8207, the main function of arbitration is selection of the next port (and thus the next instruction). The simulation includes this functionality and also calculates the number of cycles required to complete the instruction at that point. The i8207 requires four cycles to complete a memory cycle, but if the memory requests are for different banks, memory cycles can be overlapped one cycle. For each memory task, each of the four DRAM chips have an equal probability of being accessed. If two sequential memory requests are for the same DRAM chip, the memory cycle requires four simulation cycles. Otherwise, three cycles are required.

In order to accurately simulate the memory access time of the i8207, the simulation also includes the port MUX switching time. When a different port is selected, an additional two cycles are required by the i8207 to emit the MUX control for the ports. These two cycles can be overlapped with the execution of a previous instruction. The simulation includes this switching time in obtaining the number of cycles required to complete a memory access.

Refresh

The DRAMS modeled in the simulation have a long refresh period of four milliseconds. Given a 10% leeway, a refresh cycle is requested every 132 simulation cycles.

Memory Ready Signals

The G-machine/Graph memory interface is a no-wait system as described in [Rig84]. The PortA ready signal is the EAACK signal (early acknowledge) which is sent as an acknowledge message one cycle after the start of a memory cycle. Since PortB is the asynchronous port, the LAACK signal (late acknowledge) is simulated.. This signal occurs two cycles after the start of a memory cycle.

Memory Task Pseudo Code

```
begin /* Memory */
   /* memory outputs at end of $\phi 1$ */
   wait one phase
   loop
      wait one cycle
      if time for a refresh then
            add refresh command to request queue
      if  PortA message then
            add PortA command to request queue
      if PortB message then
            add PortB command to request queue
      if not in a memory cycle and there is a request then
            arbitrate requests
            get next instruction
      endif

      if in a memory cycle
            case (cycle)
                  0:  cycle = cycle + 1
                  1:  if instruction from PortA then
                              send ready message to PortA
                      cycle = cycle + 1
                  2:  if instruction from PortB then
                              send ready message to PortB
                      cycle = cycle + 1
                      arbitrate requests
                  default:
                      arbitrate requests
                      if done with this memory cycle then
                              get next instruction
                      else
                              cycle = cycle + 1
                      endif
            endcase
      endif
      go to loop
end /* Memory */
```

The Queue Tasks

The FIFO, Signal Queue, and the Garbage Can are all first-in first-out cascadable memories. The simulation model is based on the specifications for Monolithic Memories FIFO [Fir84]. All queues are 16 words deep and a 10 MHZ shift-out/shift-in rate is simulated. While the actual hardware queue is 64 words deep, a depth of

16 was used in the simulation so that stable simulation data could be obtained in a fewer number of simulation cycles. Simulations were conducted for queue lengths of 16, 32, and 64 in order to insure that performance of the design was independent of queue length.

Only simulation of the Signal Queue includes the queueing of actual data (the G-machine instructions buffered by memory for the host). Enqueueing and dequeueing of data for the FIFO and the Garbage Can is simulated by incrementing and decrementing the number of nodes in the queue. While all three queue processes are slightly different, the basic functionality is the same:

```
begin /* Queue */
    /* queue outputs at end of $\phi 1$ */
    wait one phase
    loop
        wait one cycle
        if output_ready = true then
            send acknowledge message
        if input_ready = true then
            send acknowledge message

        output_ready = false
        input_ready = false
        if there is a dequeue command message then
            if length = 64 then input_ready = true
            length = length - 1
            if length > 0 then output_ready = true
        endif
        if there is a enqueue command message then
            length = length + 1
            if length != 64 then input_ready = true
            if length = 1 then output_ready = true
        endif
        go to loop
end /* Queue */
```

The Host Task

The host task simulates a micro-processor (the Motorola MC68020) whose clock speed is 16 MHZ. The workload of the host task is defined by the type of instructions that it obtains from the Signal Queue, and the number of nodes it dequeues from the Garbage Can. An internal buffer which stores the persistent nodes whose reference counts are decremented during collection is also simulated. Instructions from the Signal Queue are processed first unless the Garbage Can or the internal buffer of garbage nodes is full.

Processing of G-machine instructions and the collection of garbage is simulated by parameterizing these functions as a series of memory accesses. Conditional branches to be taken during processing are determined by using a random variables to access probability distributions. Simulation parameters such as the size of the sub-graphs, collectibility of a node, number of memory accesses required to allocate a node, are determined in this manner.

The host processor stores some data related to garbage collection in its own private memory. All memory accesses to its own memory require three cycles. In addition, no assumptions are made about the ability of the host to combine memory accesses to its own memory and Graph or Tags memories. When two or more data items are required that reside in different memories, then two separate memory accesses are simulated.

Since the Graph and Tags memories are asynchronous to the host, the host task includes delays that would be introduced by a bus controller [Int84] when these memories are accessed. The bus controller has an internal clock that is half that of the micro-processor, and commands are accepted only during $\phi 1$ of that clock. On a memory access the host task uses a random variable to determine the clock phase of the bus controller. The bus controller also requires that memory signals be valid for two cycles, and this delay is simulated as well.

```
signal_memory(command)
   check for tags memory acknowledge message
   check for graph memory acknowledge message
   if bus phase = $\phi 2$ then
        wait one cycle
   /* bus controller will signal memory after one cycle */
   wait one cycle
   case (command)
        write:  send command message to tags memory
           set state: tags memory not ready
           wait two cycles /* send data to memory */
        readt:  send command message to tags memory
           set state: tags memory not ready
           while (tags memory not ready) wait one cycle
           update state
           wait one cycle /* to latch data*/
        readg:  send read message to graph memory
           set state: graph memory not ready
           while (graph memory not ready) wait one cycle
           update state
           wait one cycle /* to latch data*/
   endcase
end /* signal_memory */

collect_immediate()
   foreach data field of node /* there are two */
        /* read data from graph memory */
        signal_memory(readg)
        if data is a pointer then
           /* decrement reference count */
           signal_memory(readt)
           wait one cycle
           signal_memory(write)
           if node is persistent then
                if reference count != 0 then
                   add to internal garbage can
                else
                   collect_immediate()
                endif
           endif
        endif
   end /* foreach */
    wait three cycles /* clear allocated flag in own memory */
end /*collect_immediate */
```

```
collect_persistent()
   begin
        /* read reference count */
        signal_memory(readt)
        generate sub-graph size
        if size = 0 then
           if immediately collectible then
              /* 80 percent probability */
              collect_immediate()
        else
           traverse_sub_graph()
        endif
   end /* collect_persistent */

traverse_sub_graph()
   begin
        /* Traversal 0 */
        foreach node in subgraph
           signal_memory(readg)   /* read left data field */
           wait three cycles      /* read local reference count */
           wait three cycles      /* clear local reference count */
           signal_memory(write)   /* clear recently visited bit */
           signal_memory(readg)   /* read right data field */
        end /*foreach */

        /* Traversal 1 */
        foreach node in subgraph not recently visited
           signal_memory(readg)   /* read left data field */
           signal_memory(readt)   /* read recently visited bit */
           wait three cycles      /* read local reference count */
           wait three cycles      /* write local reference count */
           signal_memory(readg)   /* read right data field */
        end /*foreach */

        /* Traversal 2 */
        foreach node in subgraph
           signal_memory(readg)   /* read left data field */
           signal_memory(readt)   /* read reference count */
           wait three cycles      /* read local reference count */
           wait three cycles      /* write collectible bit */
           signal_memory(readg)   /* read right data field */
        end /*foreach */
        /* Traversal 3 */
        foreach node in subgraph
           signal_memory(readg)   /* read left data field */
           wait three cycles      /* read allocated & collectible bits */
           wait three cycles      /* clear allocated flag */
           signal_memory(write)   /* clear reference count */
           signal_memory(readg)   /* read right data field */
        end /*foreach */
        if an uncollectible node in sub-graph then
           signal_memory(readt)   /* read reference count */
           signal_memory(write)   /* reference count & persistence bit */
        endif
    end /* traverse_sub_graph */

begin /* Host */
    /* priority scheme */
    if garbage can is full then
        priority = external_gcan
    else
        if internal garbage can is full then
           priority = internal_gcan
        else
           if signal queue is ready then
              priority = signal_queue
           else
                 if garbage can ready then
                    priority = external_gcan
                 else
                    if internal garbage can is ready then
                          priority = internal_gcan
                    else priority = cycle
    case (priority)
        signal_queue:
           send dequeue command message
           wait two cycles /* time to get data */
           case (instruction)
                 alloc:
                    increment number of nodes in context
                    /* access own memory to find next node to allocate */
                    /* 1 to 4 cycles */
                    wait $x$ cycles
                    signal_memory(write) /* clear P and R bits */
                    send *enqueue* message to FIFO
                    wait two cycles /* time to send data */
```

```
                call:
                     push new context frame on stack
                     wait one cycle
                return:
                     foreach node in current context
                        /* read allocated flag */
                          wait three cycles
                          /* read persistent bit, recently visited bit,
                             & reference count */
                        signal_memory(readt)
                        generate sub-graph size
                        if size = 0 then
                             if immediately collectible then
                                  /* 95 percent probability */
                                  collect_immediate()
                        else
                             traverse_sub_graph()
                        endif
                     end /* foreach */
                     pop current context off stack
             endcase
          external_gcan:
             /* get a node from the Garbage Can *
             send dequeue command to garbage can
             /* wait to receive data */
             wait two cycles
             collect_persistent()
          internal_gcan:
             get node from internal garbage can
             collect_persistent()
          cycle:
             wait one cycle
     endcase
     go to begin
  end /* Host */
```

APPENDIX C: STATIC ANALYSIS OF RISC MICROSEQUENCES

A G-machine program is made up of a series of instructions which are called CISC instructions. These CISC instructions are expanded by the Instruction Fetch and Translation Unit (IFTU) of the G-machine into a sequence of one or more RISC instructions which are dispatched for execution by the Processor Control Unit. The instructions executed by the Graph Memory System (*read, write, trash, alloc, call* and *return*) are all RISC instructions. Therefore, the workload of the Graph Memory System is the number of times these instructions occur during the execution of a G-machine program.

A static analysis of the RISC sequences for each CISC instruction in [GAH86] was conducted in order to obtain this frequency. The table at the end of the appendix shows the number of memory instructions that occur in the RISC sequence for each CISC instruction of the G-machine. Also shown is the expected number of cycles required to execute the instructions in the sequence that are not memory instructions. All instructions execute in a single cycle with the exception of ALU operations, which require three.

Static analysis of the RISC microsequences was also used to obtain a crude estimate of the expected allocations rate per second of the G-machine. If each CISC instruction was executed only once, and each memory instruction took only one cycle, then allocations would occur in about 1.17 percent of the total cycles. Given a 10 MHZ clock cycle, that would be an allocations rate of 117K words per second. Therefore, it was concluded that the memory management scheme for the G-machine should perform well for allocation rates up to 125K words per second.

CISC Ins.	call	return	alloc	read	write	trash	Other(cycles)
add							3
addc							3
alloc			1				1
and							3
callglobfun	1						11
circ							1
copyp							1
copyv							1
decr							3
div							50
eval				5			60
fst				1			1
get_byte							1
get_fst				1			2
get_snd				1			2
incr							3
insert_byte							10
jcarry							1
jmp							0 (executed by IFTU)
jnot_neg							1
jneg							1
jnot_zero							1
jovr							1
jzero							1

CISC Ins.	call	return	alloc	read	write	trash	Other(cycles)
j_if_ptr							1
j_not_ptr							1
mk_app			1		2		7
mk_pr			1		2		7
mk_val			1		2		6
mk_v1_pr			1		2		6
mk_val_pr			1		2		5
mod							50
movep							1
movev							1
mul							30
neg							3
not							3
or							3
popp							1
popv							1
pop2							2
pop4							4
pop8							8
pop16							16
pushconst							1
pushliteral							1
ret		1		3			65
ret_int		1			2	2	14
rotp							1
rotv							1
shla							1
shra							1
shrl							1
signal							1
slidep							1
slidev							1
snd				1			1
sub							3

CISC Ins.	call	return	alloc	read	write	trash	Other(cycles)
subb							3
update				2	2	1	8
update_pr					2	1	5
update_v1				1	2	1	5
update_v_pr					2	1	5
update_poly				1	4	1	15
vrotp							2
TOTAL	1	2	6	16	24	7	454

BIOGRAPHICAL NOTE

The author was born the 3rd of January, 1955, in Denver, Colorado. She graduated from South High School in 1972 and moved to Portland, Oregon to attend Lewis & Clark College the following fall. There she received her Bachelor of Science degree in Economics in August, 1976.

Upon her graduation from college, she accepted the position of Statistician at Multnomah Foundation for Medical Care, a health research organization. There she had the opportunity to be affiliated with the development of an interactive data analysis package at Yale University. She also developed her own methodology for analysis of health care data and was asked to contribute a chapter on that topic to the book *PSRO: The Promise and the Perspective.*

In the fall of 1983 the author began her studies at Oregon Graduate Center when her son, Jacob, was two years old. She was a member of the G-machine research group, and during the last year or her studies, helped teach the Introductory VLSI and Advanced VLSI courses.

She has accepted a position at Intel Corporation as a VLSI component engineer. She is interested in architectures designed specifically for artificial intelligence processing and hopes to pursue further research in that field.