A Distributed Garbage Collection Algorithm

Terence Critchlow

UUCS-92-11

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

July 30, 1992

Abstract

Concurrent Scheme extends the Scheme programming language, providing parallel program execution on a distributed network. The Concurrent Scheme environment requires a garbage collector to reclaim global objects; objects that exist in a portion of the global heap located on the node that created them. Because a global object may be referenced by several nodes, traditional garbage collection algorithms cannot be used. The garbage collector used must be able to reclaim global objects with minimal disturbance to the user program, and without the use of global state information. It must operate asynchronously, have a low network overhead, and be able to handle out-of-order messages. This thesis describes a distributed reference counting garbage collector appropriate for the reclamation of global objects in the Concurrent Scheme environment.

A DISTRIBUTED GARBAGE COLLECTION ALGORITHM

by

Terence J. Critchlow

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Masters of Science

Department of Computer Science

The University of Utah

August 1992

ABSTRACT

Concurrent Scheme extends the Scheme programming language, providing parallel program execution on a distributed network. The Concurrent Scheme environment requires a garbage collector to reclaim global objects; objects that exist in a portion of the global heap located on the node that created them. Because a global object may be referenced by several nodes, traditional garbage collection algorithms cannot be used. The garbage collector used must be able to reclaim global objects with minimal disturbance to the user program, and without the use of global state information. It must operate asynchronously, have a low network overhead, and be able to handle out-of-order messages. This thesis describes a distributed reference counting garbage collector appropriate for the reclamation of global objects in the Concurrent Scheme environment.

To Heidi

For Heidi

Because of Heidi

CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank Dr. Mark Swanson for all of his help during the past year. His patience in guiding me through the intricacies of Concurrent Utah Scheme was invaluable, and remarkable.

Thanks to Dr. Robert Kessler and Hewlett-Packard Laboratories for providing the equipment and funding for this research.

Finally, I would give my thanks to my wife, Heidi Preuss, for supporting me through everything, and for being a great proofreader.

CHAPTER 1

INTRODUCTION

Garbage collection has been used as a way to relieve programmers of the burden of memory deallocation and allow them to concentrate on the problem they are trying to solve. Unfortunately, garbage collection has been an expensive operation, often halting the user process for several seconds while collection is occurring. As a result only a few languages, such as Lisp and Scheme, use garbage collection techniques. Most languages place the responsibility of memory allocation and deallocation on the programmer. As programs become larger, memory deallocation becomes an increasingly complex problem. This increase is particularly true in parallel programs where programmers must worry about all possible interactions between processes. This has caused a renewed interest in garbage collection algorithms.

Problems with traditional, uniprocessor, garbage collectors have been overcome. These garbage collectors can operate in real time, using only a limited amount of memory. Unfortunately, the success of these algorithms does not transfer to a distributed computing environment. Distributed garbage collectors have to be concerned about issues such as node synchronization and network traffic overhead. In addition, many systems require the garbage collector to handle delayed and out of order messages.

The Concurrent Scheme environment requires a global garbage collector to address these issues. Concurrent Scheme is an extension of the Scheme programming language that provides parallel programming capabilities. Concurrent Scheme has introduced several `global objects` to implement this capability. Unfortunately, global objects cannot be reclaimed by conventional garbage collectors because they may be referenced by several nodes simultaneously. Reclaiming global objects requires an asynchronous garbage collector that can handle out of order messages while maintaining a low network overhead.

This work describes a distributed reference counting garbage collector that has been implemented in the Concurrent Scheme environment. The algorithm handles out of order messages and requires no internode synchronization. The asynchronous nature of the algorithm contributes to a network overhead of at most one message per reference. The initial results from the implementation are encouraging; the efficiency of the algorithm is within acceptable limits, and can be improved even further.

Other garbage collection algorithms are discussed in the next chapter. The Concurrent Scheme environment is described in detail in Chapter 3. Chapter 4 describes the distributed reference counting algorithm in detail. Chapter 5 provides an analysis of the algorithm, including timings obtained from the current implementation, and describes future work which is beyond the scope of this thesis.

CHAPTER 2

PREVIOUS WORK

Garbage collectors can be divided into three major types: marking, reference counting and generational. Each type of collector has advantages and disadvantages. Marking collectors are easy to implement and will remove all garbage from a system. The amount of work performed by a marking collector is not proportional to the amount of garbage: mark and sweep collectors do work proportional to the amount of memory in the system, and copying collectors do work proportional to the number of live objects in the system. Reference counting collectors are also easy to implement but do work proportional to the total number of references to an object. The problem with reference counting collectors is their conservative nature; without additional support they are incapable of reclaiming all the garbage in a system. In particular, circular references cannot be reclaimed by straightforward reference counting algorithms. Generational collectors examine fewer objects per collection than marking collectors but are still able to reclaim all the garbage. However, a generational garbage collector requires more computational overhead to reclaim all garbage in a system than a marking collector.

2.1 Marking Garbage Collectors

Marking collectors determine what is garbage by first determining what is being used. Marking collectors use a set of objects known as the root set to determine what is alive. By definition, all objects not reachable from the root set are garbage. The basic algorithm used to determine which objects are alive is similar to a graph traversal algorithm. The root set provides an initial set of unvisited, reachable objects. As long as there are reachable objects that have not been visited, visit a reachable object, mark the object as visited, and mark the objects reachable from that object as reachable. Any object not marked at the end of the iteration is garbage.

There are two basic types of marking collectors, copying collectors and mark and sweep collectors. The major difference between these types of collectors is what occurs when a node is marked as visited. In copying collectors the step of marking the object visited is a combination of copying the object and replacing it with a forwarding pointer. In mark and sweep collectors the process of marking the active object is followed by a collection step in which all unmarked objects are placed on a free list. Some mark and sweep collectors have a compaction phase in which all live objects are moved together; reducing page faults. Both types of collectors have been used successfully in a variety of applications.

2.1.1 Copying Collectors

Baker [4] developed one of the first incremental, real-time copying garbage collectors. In Baker's algorithm there are two disjoint heap spaces, TOSPACE and FROMSPACE. The user process, or mutator, allocates all cells from TOSPACE. FROMSPACE contains garbage and the data not yet copied by the garbage collector. When TOSPACE becomes full the roles of FROMSPACE and TOSPACE are reversed, or flipped. This role reversal is possible because, by the time the flip occurs, FROMSPACE contains only garbage. Immediately following a flip, the root set is placed in the new TOSPACE. Garbage collection begins after a flip. Every time a new cell is allocated, a known number of cells are copied from FROMSPACE to TOSPACE. These cells are found by searching through TOSPACE to find cells containing a reference to FROMSPACE. The cell being pointed to is copied and the reference is updated to the new cell location. When a cell is initially copied from FROMSPACE to TOSPACE a forwarding pointer is left behind so other references to the cell will refer to the same structure. All attempts to copy a previously copied cell return the forwarding pointer. When there are no pointers from TOSPACE to FROMSPACE the garbage collection has completed.

The number of cells copied during each allocation must be large enough to ensure all accessible cells are copied before TOSPACE becomes full. Every time the mutator references a cell not yet copied from FROMSPACE, the cell is copied to TOSPACE. This helps ensure that the garbage collection will complete before the mutator fills TOSPACE. The method of copying cells distributes references over a large area of memory. The result is poor data locality which impairs mutator performance. Baker's algorithm is a real-time algorithm only in the strictest sense. The amount of time it will take to perform any given operation is bounded by a constant, but the constant is very large. This makes the algorithm impractical to use in a genuine real-time system. However, Baker's algorithm provides a basis for more practical algorithms.

Dawson [11] adapts Baker's algorithm to improve localization of data. This is done in two ways. First, the root set is slowly moved into TOSPACE, not immediately placed there after a flip. Second, cells are copied to different locations in TOSPACE depending on the reason the copy occurred. If the copy was the result of the mutator attempting to perform an allocation, the cell is copied to one location. If the copy was the result of the mutator referencing an uncopied cell, the cell is copied to a different location. These minor modifications make a significant improvement in the efficiency of the mutator.

Appel, Ellis and Li [3] also modified Baker's algorithm to improve performance and allow for parallel execution. Their collector works on a page- at-a-time as opposed to a cell-at-a-time. When a flip occurs, all pages in TOSPACE are read-protected. The pages in FROMSPACE become TOSPACE. If there are any uncopied pages at the time of a flip, they are copied before the flip begins. The copying of an unknown number of pages means the algorithm is not truly real-time. However, the authors maintain that the amount of time used by this step is insignificant when compared to the overall operation of the program. If the mutator tries to reference an uncopied object, a page fault occurs. The page is then copied and unprotected. A concurrent thread is used to copy pages while the mutator is running. This reduces the number of page faults and dramatically improves the performance of the algorithm. The results obtained by the initial implementation of this algorithm, using a single mutator thread, are promising.

2.1.2 Mark and Sweep Collectors

Dijkstra, Lamport, Martin, Scholten and Steffens [14] developed the first concurrent mark and sweep garbage collector. In their algorithm the mutator and the collector run in parallel. The collector works by marking nodes with different colors. Initially, the root nodes are marked grey and all other nodes are marked white. When a grey node is visited, it is marked as black and all of the nodes reachable from it are marked grey. The authors use two invariants to show that if no grey nodes exist, then all the white nodes are garbage. When the collector has determined there are no grey nodes remaining it performs a sweep. The sweep consists of moving all white nodes to the free list and marking all other nodes as white. After the sweep has been completed, the collection starts over. For this to work properly, there must be cooperation between the mutator and the collector. All the cells the mutator allocates while a collection is occurring must be colored grey. If the mutator allocates nodes during the mark or sweep phase of the collector, the collector will not know about these nodes for the current collection. These nodes will not be mistaken as garbage because they have been marked as grey. The collector will consider these nodes on its next pass. Dijkstra's algorithm has practical value as seen by the number of similar algorithms used in distributed systems.

Hudak and Keller [17] have modified Dijkstra's algorithm to work on a distributed system with a high degree of parallelism. In Hudak's algorithm, all nodes are initially white. There is a distinguished node called the root node from which all elements of the root set are reachable. Garbage collection begins by selecting the root node and spawning a task for each reference contained in the root node. A count of the number of tasks spawned is associated with the root node. The marking phase progresses as subtasks are assigned nodes. If a node is a leaf node, or if the node has been previously marked, the task immediately returns. Otherwise, the task marks the node grey, spawns a subtask for each reference by the node and keeps count of the number of subtasks spawned. The task blocks, waiting for the spawned tasks to return. When all subtasks have completed the current node is marked black and the task returns. The mutator must cooperate in the marking phase by spawning or executing a marking task under certain circumstances. In most cases, the mutator is not affected by the collection. The marking phase is over when all subtasks spawned by the root task have returned.

The sweeping phase is broken into three subphases. During the task deletion phase, any mutator task that references garbage is deleted. This is done because, in a pure functional language, these tasks can not contribute to the final result of the program. After all irrelevant tasks are deleted, the garbage nodes are reclaimed. Finally, the remaining nodes are reinitialized. The authors acknowledge this algorithm requires a large amount of space but feel the added speed is worth the expense. This algorithm is an interesting attempt at constructing a real-time mark and sweep collector. Although Hudak's algorithm meets the definition of real-time stated in the paper, the algorithm is not real time by other standards.

Because the execution of a marking task by the mutator may take an arbitrary amount of time the collector is non-real-time in the sense of all operations being bounded by a constant. However, the algorithm is successful in creating a highly parallel garbage collector that is distributable and able to reclaim irrelevant tasks.

Juul [20] has adapted Dijkstra's algorithm for use in the Emerald system. As in Dijkstra's algorithm there must be cooperation between the collector and the user processes for the collector to work properly. In Juul's algorithm, this cooperation occurs in two ways. First, whenever a user process attempts to access a grey object a fault occurs, the object is marked black, and the object's references are marked grey. Second, an object is marked black when it is allocated. This ensures user processes only see black objects. The collection process is broken into two levels, local collection and global collection. In the global collector, the root objects are initially marked grey, all other objects are white. The collector selects a grey object. If the object is local to the processor it turns the object black, and turns the object's references grey. If the object is not local to the processor, it places the object in a grey set. At some point, the collector will send all of the objects in its grey set to the other processors. If the other processor is currently performing a global garbage collection, the information is incorporated immediately. If the processor is not in the process of global garbage collection it will begin a global garbage collection and use the objects provided as part of its root set. This allows global references to be found and noted. This also allows any processor to initiate a global collection. The marking phase is complete when there are no grey objects on any processor. The sweep phase of the global collection clears a global reference bit on all objects that are considered garbage. Local collection occurs when at least one processor is unavailable for global garbage collection and the current node has no grey objects.

There are three major differences between the local collection algorithm and the global collection algorithm. First, the local collector need not send the grey set to other processors. Second, the root set is expanded to include all objects that have been referenced by other processors. These objects can be distinguished because the global reference bit has been set. Third, the local collector actually frees memory used by the objects that it determines are garbage. This algorithm has not yet been completely implemented so results are not available. There are three major problems with this algorithm, all caused by the sharing of references by passing objects between processors. First, it creates a large amount of message overhead on the network. Second, it requires a notion of global state to determine when the marking phase has completed. Third, it requires synchronizing all processors in order to perform a global garbage collection.

Shapiro, Plainfossé and Gruber [27] [28] have developed an asynchronous, distributed, mark and sweep collector. In this algorithm, the network is divided into several uniquely identified spaces. Each space maintains an Object Directory Table (ODT) and an External Reference Table (ERT). The ODT contains potential references by other spaces to objects residing in the current space. The ERT contains the location of objects referenced in the current space residing in other spaces. When a reference is copied out of a space, an entry containing the reference and the destination space is created in the ODT. As a reference is copied into a space, the reference is replaced by a stub pointing to an ERT entry containing the reference and the space that sent the reference. If the stub is referenced, the value is requested from the space stored in the ERT. Local garbage collections occur on a per space basis, the global garbage collector is the result of cooperation between spaces. During a local garbage collection the ODT is included in the root set preventing any externally referenced object from being collected. At the end of the local collection, unmarked ERT objects are removed and a message is sent to the space containing the reference causing the associated ODT entry to be removed. To handle out of order messages, Shapiro implemented a time stamp protocol. This protocol allows out of order messages to be detected and prevents the garbage collector from reclaiming live objects. The major problem with this algorithm is the amount of network overhead generated by handling lost and out of order messages. Another, less limiting problem is the amount of memory used by the ODT and the EDT.

2.2 Reference Counting Garbage Collectors

Reference counting garbage collectors determine which objects are reclaimable by noting which objects are not referenced by other objects. An object not referenced by any other object cannot be used by the program and is garbage, an object that is referenced by another object may be used by the program if the object that references it is accessible by the program. To determine which objects are referenced, a reference count is associated with each object. When an object is first created, it has a reference count of one because it is referenced by the object creating it. When a reference to an object is added, the reference count is incremented. When a reference to an object is removed, the reference count is decremented. When the reference count reaches zero the object is garbage and can be reclaimed. Reference counting collectors have two major advantages over marking collectors. First, reference counting is inherently an incremental process, making real-time reference counting collectors easy to develop. Second, reference counting algorithms map to distributed systems very easily, because they do not require global synchronization and can be made to handle out of order messages.

The major problem with reference counting algorithms is cyclic references. If object A refers to object B and object B refers to object A neither A nor B will be reclaimed. Their reference counts will never drop below one due to the reference by the other object. Many algorithms have been developed to overcome this problem. The most common solution, used in [12], [23], and [34], is the use of an auxiliary, mark and scan collector to aide in the garbage collection process. The auxiliary collector is invoked infrequently, as a supplement to the reference counting collector. If the system being used is able to detect the creation of a circular reference, as the combinator machine used by Brownbridge [7] is, separate reference counts can be used for cyclic and noncyclic references. These reference counts can be used to reclaim cyclic structures by using an algorithm similar to Bobrow's [6] to determine when the structure is no longer referenced externally. In some cases, a special feature of the language allows an efficient algorithm for detecting circular references to be developed. Friedman's collector [15] exploits the fact that, in a purely functional language, circular references cannot be created by user programs. Because circular references can be created only in well-defined circumstances, a specialized mechanism for handling cycles efficiently can be developed.

Many reference counting algorithms have a problem with the size of the reference counts. If the algorithm requires the reference count to be stored with the object directly, the size of the reference count must be established. This limits the number of references to an object that can be accurately recorded by the algorithm. The algorithm must outline what action is to be taken when additional references are required. In some algorithms, additional references are never reclaimed, in others, additional memory is used to store the required information about these references. In either case, additional overhead is consumed dealing with this problem. Algorithms that do not require the reference count to be stored with the object avoid this problem by allowing the reference count to grow as required.

One of the first distributed reference counting systems was developed by Bevan [5]. In this algorithm, when an object is created it is given a reference count of maxvalue. The original reference to this object is given a reference weight of maxvalue. Maxvalue is an arbitrarily large power of two. When a copy of a reference is made the reference weight of the original reference is split equally between the old reference and the new reference. When a copy is made of a reference having a reference count of one, a special object, called an indirection cell, is introduced. The indirection cell points to the object the original reference pointed to and has a reference count of maxvalue. The original reference is modified to point to the indirection cell and is given a reference weight of maxvalue. The copy is then performed as normal. When a reference is deleted, a decrement message containing the reference weight is sent to the object the reference pointed to. This value is subtracted from the reference count of the associated cell. When the reference count of an indirection cell reaches zero, the cell sends a decrement message with value one to the cell it points to and deallocates itself. When the reference count of the object reaches zero it is garbage and is reclaimed. This algorithm provides an asynchronous collector having low message overhead and low computational overhead.

The major problem is the trade off between memory and use of indirection cells. If maxvalue is large, memory is wasted holding the reference weight. If maxvalue is small the number of indirection cells may become large causing increased message traffic and response time. For example, if there are several copies made of a single reference there will be a linked list of indirection cells. In order for the newest reference to access the object it must traverse this list. If the indirection cells are located on different nodes this traversal will introduce unnecessary messages and require more time than an access from the original reference. The collection algorithm is claimed to be a real time algorithm. However, referencing an object may take an indeterminate amount of time because an unknown number of indirection cells must be traversed implying the algorithm is not strictly real time. If the number of references per object is small this algorithm is real-time and works well. However, this algorithm is not scalable to a system where several nodes reference the same object.

Paul Watson and Ian Watson [33] modify a weighted reference counting algorithm credited to K-S Weng [35] but similar to Bevan's. In this algorithm, when a graph headed by an indirection cell is evaluated, the value is stored in the indirection cell so the indirection is removed. This improves the efficiency of the original algorithm because redundant values need not be recomputed. However, this type of technique requires special hardware, such as a graph based machine, and is not applicable for languages such as Scheme because different references may no longer be identical. These restrictions prevent this algorithm from use in a general environment.

DeTreville [12] uses a different approach to parallel distributed reference counting for the Modula-2+ garbage collector. In his algorithm, DeTreville differentiates between local references and shared references. The local references are determined by scanning the thread state during every garbage collection. The shared references are stored with the object and incremented and decremented in the normal way. When the shared reference count of an object is zero it is placed on a list called the zero count list (ZCL). At the end of the collection, the ZCL is scanned. Any object on the ZCL having no shared references and no local references is garbage and is reclaimed. If an object on the ZCL has a local reference but no shared references it remains on the ZCL and is not reclaimed. If an object on the ZCL has a shared reference count greater than zero it is removed from the list. An object on the ZCL may have a nonzero shared reference count because the mutator may have added a shared reference to an object that previously only had a local reference. DeTreville solved the problem of circular references by adding a mark and scan collector to the system. The mark and scan collector acts as a secondary collector. The major purpose of this collector is to reclaim circular references. Because the reference counting collector is supposed to collect most of the garbage the mark and scan collector is invoked infrequently. In addition, the reference counting collector may preempt the mark and scan collector. The reference counting collector was chosen as the primary collector for DeTreville's system because it is faster and more local than the mark and scan collector. The use of two different collectors provides a reasonable level of efficiency while still reclaiming all garbage in a system.

2.3 Generational Garbage Collection

A generational garbage collector is based on the premise that most garbage in a system is created by objects that have short life spans. If an object has existed for a long period of time it is unlikely to become garbage soon. Therefore, garbage collectors should spend less time collecting old objects and more time collecting new objects. Generational collectors achieve this through a process known as aging. As an object survives garbage collections, it slowly ages. Aging usually, but not necessarily, involves moving older objects to distinct areas so two different generations of objects do not share the same heap space. Collection is then performed on the younger generations on a more frequent basis than on the older generations. A generational garbage collector is any garbage collector that attempts to use aging techniques in conjunction with other garbage collection algorithms to improve the efficiency of the algorithm.

Lieberman and Hewitt [22] combine a generational collector with a garbage collector based on a modification of Baker's algorithm. The heap is divided into several small regions that can be garbage collected independently. All of the objects within a region are the same generation. When a region is collected, all currently referenced objects are copied to a new region. Determining which objects are still alive requires scavenging newer generations for pointers into the region being collected. Pointers from older regions into newer regions are kept in a table associated with the newer region making scavenging regions older than the current region unnecessary. Scanning the table is quicker than scavenging the older regions because pointers from older generations to newer ones are uncommon. When all live objects have been copied the original region is freed. When an old region is collected, several regions must be scavenged because most regions are younger than the current region. This makes collecting older regions very expensive. Collecting a newer region requires very little scavenging because most regions are older than the current region. This means collecting newer regions is relatively cheap. Unfortunately, no results are available for this collector although the authors speculate the algorithm has the potential for good performance.

Moon [25] also uses Baker's algorithm to develop a generational garbage collector for the Symbolics 3600 computer. Moon's algorithm differs from Lieberman's in three major ways. First, objects of different ages may occupy the same memory area. Second, the user may specify the age of an object. This allows objects that are known to be static to be removed from consideration during garbage collections. Third, there are only three ages an object may be: ephemeral, dynamic or static. The age of an object, and the level of an ephemeral object, is encoded in the objects address. Ephemeral objects are objects that are expected to have short life spans. There are several levels of ephemeral objects; by default newly created objects exist on level one. As a level fills, objects are moved to the next level. Ephemeral objects of the highest level are aged into dynamic objects. A special table is used to maintain the references to ephemeral objects by objects of other ages. This table is used to improve the efficiency of ephemeral garbage collections. Garbage collection of dynamic objects is performed less frequently, and is more expensive, than for ephemeral objects. Static objects are assumed to live forever. They can be reclaimed by an explicit garbage collection call, but will never be reclaimed by normal system operation. If the entire system state is saved, all objects are garbage collected before the system is written to disk. When a saved state is restarted, all objects are considered static. The use of specialized Lisp hardware, in particular tagged memory, has made this algorithm very efficient, compared to other collectors.

Goldberg [16] combined a generational garbage collector with a reference counting algorithm. The result is a collector that will work in a distributed environment. In this algorithm, an object keeps track of its copy count and its generation. When a new object is created it has generation zero and copy count zero. The object also has an associated ledger, stored on the processor where the object was created. The ledger is an array that contains information about the number of outstanding references for each generation. The ledger is initialized to have a value of one for generation zero, and zero for all other generations. When a copy of the object is made, the object's copy count is incremented. The copy of the object has a generation one greater than the copied object. When an object has been discarded, the ledger is sent a message containing the generation, g, and the copy count, c, of the discarded object. The ledger then subtracts one from the value stored at index g, and adds c to the value stored at index $g{+}1$. When all indexes of the ledger are zero, the object can be reclaimed. The use of the ledger allows for the possibility of out of order messages. The message overhead of this algorithm is exactly one message per reference. This low overhead is the major benefit of this algorithm.

There are two major problems with this algorithm. The first problem is the amount of space used to implement the algorithm. There is a small cost associated with each reference of the object and a large cost associated with the ledger. The second problem is the amount of computational overhead introduced by this algorithm. Each time an object is created, the copy count, generation and ledger must be initialized. This is true even if the object is never referenced again. There is also additional overhead involved in the copying and deletion of objects. The algorithm used to discard individual references is not described, presumably some other garbage collection algorithm is used.

CHAPTER 3

THE CONCURRENT SCHEME ENVIRONMENT

The Concurrent Scheme (CS) language was developed by Swanson at the University of Utah [21, 31, 30]. The current implementation of CS, Concurrent Utah Scheme (CUS), is supported on three platforms: Mayfly [9, 10], HP300 and HP800. The Mayfly architecture is an experimental parallel architecture possessing several unique features. Each node in the Mayfly architecture is connected to other nodes to form a processing surface, or hexagonal mesh. The processing surfaces are connected together to form the network. Messages are passed from node to node using a dynamic routing scheme that guarantees the messages will be delivered to their destination, but may arrive out of order. The Mayfly is designed to be a highly scalable architecture; the current system is capable of scaling to over 500,000 nodes. The HP300 and HP800 architectures are simply networks of homogeneous workstations.

The motivation behind CS was to design a language scalable to a highly parallel system; to achieve this, two basic goals were formalized. First, communication between nodes must be asynchronous. There are many reasons for this, the most compelling being the amount of network traffic generated by synchronization messages. The network is a critical resource and a potential bottleneck. Second, no concept of global state may exist. If a node were responsible for maintaining the state of the network, it would become a bottleneck: the state of the system cannot change faster than the node can keep track of it. These goals are based on the observation that any bottleneck will reduce scalability.

In order to achieve user specified parallelism, Swanson extended the traditional Scheme language to include domains, threads, placeholders and ports. Copying semantics were introduced to maintain consistency between domains. The semantics ensures all references within a domain are consistent and cannot be modified by other domains. These extensions to the base language are discussed in Section 3.1. In Section 3.2 a detailed description of the implementation is given. This description will form the basis for an explanation of the global garbage collection algorithm.

3.1 Extensions to Scheme

When the CUS environment is initialized, the number of nodes to be used in the network, n, is specified. Each node in the network is assigned a unique identification number from the range $[0, n-1]$. The Mayfly system assigns nodes from the available processors. The other platforms require a special file containing names of potential network nodes. The machines specified in the file are tried in sequential order until the required number of machines are allocated. Each node executes an identical copy of the environment.

CUS domains have spatial characteristics. All computations occur within domains and all variables live within domains. Local garbage collections occur on a per domain basis using a copying collector. Domains form the basic unit of mutual exclusion; only one thread may be executing in a domain at a time. This ensures that the programmer need not worry about locking protocols for local variables. Domains may be created on any node by specifying the desired node number when the domain is created. If the node number, m , exceeds the number of nodes in the system, n, the domain is created on node (m mod n).

Threads are the instrument of parallelism. When a thread is created it is given a procedure to execute. If the procedure is a closure [1] the thread executes in the domain in which the closure was created. If the procedure is not a closure, a domain in which to execute the procedure must also be given. The execution of a thread occurs on the node containing the required domain. When there are several active threads on a single node the resources are time shared to give pseudo-parallelism. True parallelism occurs when threads execute on different nodes at the same time. A placeholder may be used to return the value of the thread at completion.

Placeholders represent the promise of a potential value at some point in the future. After creation they can be manipulated like most other types. However, if an operation occurs for which the value of the placeholder must be known, the thread requiring the information blocks until the placeholder is determined. A thread may be forced to block until a placeholder is determined by using an operator that returns the value of the placeholder. The value of the placeholder is `determined` by a function call. This function requires the placeholder to be determined and the value to which it is to be determined. Once a placeholder has been determined, threads waiting for the value are placed on the active queue. A placeholder may be determined only once.

Ports are a built-in implementation of first-in, first-out queues. When a port is created the user specifies the number of objects the port will buffer. The program may then add and remove objects from the port one at a time. If the port is empty and a thread tries to read from the port the thread will block until an object is sent to the port. If a thread tries to send an object to a full port the thread will block until an object is read from the port.

3.2 Implementation Details

The CUS system was designed for the Mayfly parallel computer architecture. In this architecture a node contains two processors. One processor, the `mp`, is used primarily for systems support operations. It handles the sending and receiving of messages and performs scheduling operations. Threads running on this processor cannot be interrupted. Thus the `mp` provides a convenient location for atomic operations to be performed. The other processor, the `ep`, is used primarily for user processes. All of the high-level operations execute on this processor. The `ep` is time-shared between all active threads. System calls exist to switch the current thread from one processor to the other. The thread is blocked while the switch is taking place. If there is a thread running on the desired processor the migrating thread is added to a queue and resumed in turn, otherwise it is executed immediately. It is possible to have processes executing on both the `mp` and the `ep` at the same time allowing true parallelism on a single node. The architectural uniqueness of the Mayfly is simulated by low-level software on other CUS platforms. This simulation allows the advantages of the `mp` to be utilized even when the processor does not exist.

All communication between domains takes place via message passing. When a domain communicates with another domain the information to be shared is sent to an export function. The information is completely dereferenced removing all pointers into the current heap. When all references have been resolved the information is copied into a message packet. If the message is too large to be sent in one packet multiple packets are used. The information is sent to the requested node by the `mp`. The message is received by a thread running on the remote node's `mp` and copied into the appropriate domain's heap. Once the copy has been completed the thread switches to the `ep` to process the information.

Domains, threads, placeholders and ports are represented as structures that exist outside the scope of any domain's heap. This is necessary because the copying protocol must not affect the uniqueness of these objects. Consider the sequence of expressions in Figure 3.1. If placeholders existed within domains, when placeholder

```
(set! aa (make-placeholder)) ;; in domain A
(set! bb (touch aa))         ;; in domain B
(set! cc (touch aa))         ;; in domain C
(determine aa (list '1 '2))  ;; in domain A
(set! dd (touch aa))         ;; in domain D
```

Figure 3.1. Potential Copying Protocol Problem

`aa` was exported to domain B a copy of the entire placeholder structure would be sent. Similarly, when `aa` is sent to domain C a different copy of the structure would be sent. Each of the three copies of the placeholder would be equal but not identical. When `aa` is determined in domain A only the structure local to domain A would be determined. The copies of the placeholder in domains B and C would not be determined because they are different. At this point `bb` and `cc` would be equal to each other but not to `aa`. When `aa` is copied to domain D it has already been determined and its value will be assigned to `dd`. To avoid this inconsistency and to maintain the copying protocol Swanson introduced immediate objects. Immediate objects are one word tagged objects that contain all information required to locate the structures they represent.

Swanson uses an export table to organize the global objects. The export table is an array of export objects, shown in Figure 3.2, that contain pointers to the real global objects. When a global object is created, the actual structure is allocated out of the global heap. This object is then assigned to an entry in the export table. Based on the entry assigned, the node where the object was allocated and the type of global object created, an immediate object is generated and returned. This immediate object is used by all higher level functions. The immediate object is unique because there is exactly one export table per node and all global objects created by a node are referenced by the export table on that node. Because the

```
(defstruct (export)
    object    ;; the object being exported; nil–¿invalid
    link      ;; ptr to next export for owning domain or next available
    owner     ;; pointer to owning domain
)
```

Figure 3.2. Export Object

immediate object is one word the copying protocol does not have any indirections to follow when the object is exported. When detailed information about the object is required, for example the value of a placeholder, the immediate object is dissected to reveal the location of the real object. The real object is accessed directly and the required information is returned. Because all copies of the immediate object point to the same location, there cannot be any consistency problems.

Closures pose an anomaly to the copying protocol. Although no domain is allowed to contain a direct pointer into another domain a closure by definition is a pointer into a domain. Therefore whenever a closure is exported a pointer is created from one domain to another domain. To resolve this problem, Swanson created immediate closures to represent exported closures. When a closure is to be exported, an entry is reserved in the export table. The object in the export table points to the closure within the exporting domain. This does not violate the copying protocol because the export table exists outside of all domains. An immediate closure object is then created and exported in place of the closure. When an immediate closure is copied into a domain, or `imported`, it is immediately wrapped in a special closure called a gateway. A gateway is a closure containing an immediate closure object in its local state. Gateways are easily distinguishable from other closures because they have a special form. When a gateway is executed, the associated immediate object is used to reveal the location of the real closure that is executed in the proper environment.

The `root domain` is used to store global variables. It is different from other domains because a copy of the root domain is kept on every node in the network. This allows for quick access to global variables. However, there is a great deal of overhead associated with keeping all copies of this domain consistent. Any time a global variable is added or modified the change must be broadcast to all nodes in the network. For this reason Swanson discourages the use of globals.

The domain level garbage collector is a copying collector. The roots of the domain are taken from the domain's current environment, the current stack, and the thread currently running in the domain. Additional roots are provided by threads waiting to enter the domain, threads whose execution stack includes the domain, and closures that have been exported from the domain. The root domain includes the symbol table in its root set. The collector allocates a new heap for the domain from the global heap, copies over all objects accessible from the root set and returns the old heap to the global heap.

CHAPTER 4

GLOBAL GARBAGE COLLECTOR FOR CUS

In the current implementation of CUS, global objects are not collected. This results in the size of the export table increasing monotonically. At some point, the export table, and associated objects, consumes all available heap space and threads are unable to execute within domains. This problem can be avoided by the use of a garbage collector for global objects. It is possible, but unlikely, for the export table to overflow even when all of the garbage is collected. A distributed reference counting algorithm is presented that will solve the global garbage collection problem in CUS.

The requirements for the collector are outlined in Section 4.1. Section 4.2 describes an obvious solution to the problem which, although incorrect, does provide useful insight. Section 4.3 describes the algorithm in detail. Section 4.4 provides three examples of the algorithm's execution. Section 4.5 proves the algorithm to be correct.

4.1 Goals

Several problems must be overcome to satisfy the requirements placed on a global garbage collector in CUS. Garbage collection algorithms that require a large number of messages to be passed between nodes are unsuitable for this environment. The resulting increase in network traffic would slow the network an unacceptable amount. The collector must operate asynchronously. Any attempt to synchronize the garbage collector would result in problems with network traffic and dramatically reduce the computational efficiency of the entire system. The amount of work required by the collector should not depend on the number of nodes in the system. Dependency on the network size will reduce the scalability of the collector. Messages arriving out of order should not cause the collector to perform incorrectly or to incur any additional computational overhead. The collector should not stop user threads for an extended period of time to perform a collection. This implies either the collector should be executed in parallel with the user threads or the collector should be a real time algorithm.

4.2 An Obvious Solution

If a reference counting garbage collector is to be used to collect global objects in CUS, an obvious algorithm would store an objects reference count with the object. When a reference to the object is created, an increment message is sent to the node on which the object resides. When a reference to an object is removed, a decrement message is sent to the object. Unfortunately, this algorithm will not work. Consider the following sequence of actions on a three node network:

- Node 1 creates a global object, `A`; `A` has a reference count of one.
- A reference to `A` is created on node 2; `A` has a reference count of two.
- A reference to `A` is created on node 3. An increment message is sent to node 1, but the increment will not occur until the message arrives.
- Node 3 reclaims its reference to `A` and sends a decrement message to node 1. `A`'s reference count is still one, because neither the increment or the decrement message has arrived.

- The decrement message arrives at node 1. `A`'s reference count becomes zero and `A` is reclaimed.

- The increment message arrives at node 1, causing an error because `A` does not exist.

In a reference counting system, an increment must occur before the corresponding decrement occurs in order to guarantee correctness. This property does not hold in the obvious solution because a race condition exists between increment and decrement messages when in-order delivery is not guaranteed. One solution to this problem is to introduce synchronization messages. When a node sends an increment message, it waits for the destination node to send a reply confirming it received the increment. Unfortunately, the node cannot continue its normal operation until this reply arrives, introducing an unacceptable amount of waste into the system. Another, more practical solution, is to force all increments to occur locally. If an increment occurs locally, it can be guaranteed to occur before the corresponding decrement, eliminating the race condition without additional network overhead.

4.3 The Distributed Garbage Collector

This section describes the algorithm developed to solve the problem of global garbage collection in CUS. Section 4.3.1 explains the data structures used to implement the algorithm described in Section 4.3.2. Section 4.3.3 describes a revised algorithm, which, although equivalent to the initial algorithm, may be easier to understand.

4.3.1 The Data Structures

The implementation of the algorithm required the introduction of a new data structure to the system. This structure, the import table, is an array of import table entry objects, shown in Figure 4.1. There is a unique import table residing on every node in the system. This table contains one entry for every node in the network. The purpose of this table is to hold all immediate objects imported to the current node. A variable length array of import objects is stored with each table entry. The array associated with import table index i on node n contains all immediate objects imported by node n that reside on node i. The time required to locate an element in the used list is proportional to the number of elements currently imported from the node. Insertion and deletion of import objects can be performed in constant time once the list has been searched.

The import objects described in Figure 4.2 are used to hold all information associated with an imported object. This information is retrieved by a linear search

```
(defstruct (import-table-entry
                     (:constructor create-import-entry)
                     (:print-function print-import-entry))
    size        ; current size of the array being pointed to
    free        ; pointer to the head of the free list
    used        ; pointer to the head of the used list
    array       ; the array of objects imported from here
)
```

Figure 4.1. Import Table Entry Object

```
(defstruct (import
                      (:constructor create-import)
                      (:print-function print-import))
    object      ; the item that we have imported
    link        ; pointer to the next element
    contact     ; the node that we originally got the object from
    exported    ; has this object been exported from this node
    ref-count   ; the number of references to this import
)
```

Figure 4.2. Import Object

for the immediate object through the appropriate import table entry array's `used` list. The `reference count` represents the number of domains on this node that know about the object. The `contact` field is the number of the node that initially sent the object to the current node. The `exported` field is used to determine if the current node has exported the immediate object. This field is used only if the immediate object does not reside on the node. The `link` field is used to maintain either the `used` list or the `free` list for the current import table entry.

In addition to the introduction of the import table, two modifications were made to existing data structures. The first was the modification of the export object structure to include a reference count. This reference count represents the number of copies of an immediate object sent from this node not yet accounted for. The other modification was the addition of the import field to the domain structure of Figure 4.3. This field is used to determine which immediate objects the domain

```
(defstruct (domain
                           (:constructor create-domain)
                           (:print-function print-domain))
    name              ; just a symbolic name for debugging
    link              ; ptr to next domain on node
    entry-count       ; count of nested entries by the cur thread
    thread            ; thread occupying the domain
    waiting           ; list of threads waiting to get in
    touchers          ; threads that TOUCHed with wait-outside=T
    delay-queues      ; list of delay queues associated with the domain
    heap-base         ; ptr to base of current heap
    heap-limit        ; ptr to upper bound of current heap
    heap-next         ; ptr to next available location in cur heap
    exports           ; ptr to list of exported closures and phs
    phs               ; placeholders imported to this domain
    imports           ; list of imported objects
    external          ; exportable id of this domain
)
```

Figure 4.3. Domain Object

references. The import field references a list of paired objects. The first element in the pair is used as a mark for the local garbage collector. The mark allows the local collector to determine when there are no references to the immediate object within the domain. The second element in the pair is a pointer to an import object.

4.3.2 The Initial Algorithm

Figures 4.4 and 4.5 provide a general outline of the proposed distributed reference counting algorithm. Figure 4.4 describes the modifications required in the export and garbage collection functions. Figure 4.5 describes two additional functions required to implement the garbage collection algorithm. The algorithm is described in detail below.

The garbage collection algorithm is a distributed reference counting algorithm. The collector knows an object is garbage when the export table reference count for the object reaches zero. When an object becomes garbage, the collector frees the memory used to contain the global object and adds the export object to the export table's free list.

When an immediate object is sent to another domain, the local export table reference count must be incremented. If the object resides on the current node, the increment can be performed immediately. If the object resides on another node, a local export object exists only if the object has been previously exported from this node. In order to determine if the object has been previously exported the import table is searched for the associated import object. If a local export table entry for the object exists, the exported field of the import object will contain the index into the export table where the entry is located. The reference count for this entry is incremented. If a local export object does not exist, an export object is allocated. The export table reference count for this entry is incremented. The exported field of the import object is set to the index into the export table where the export

```
(defun export-an-object (object)
  (if (local object)
      (increment-export-count (export-table-loc object))
      (let ((export-entry ; determine if prev exported
                          (export-status (import-object object))))
         (if export-entry
             (increment-export-count export-entry)
             (let ((export-entry  ; have to create export entry
                              (allocate-export-table-entry object)))
               (update-export-status export-entry object)
               (increment-export-count export-entry)))
          (increment-import-count object))))


(defun local-collection ()
  (for-the-entire-heap
     (if (immediate-object current-object)
         (mark-and-copy  current-object)
         (copy-and-return current-object)))
  (for-each-element-in-the-import-list
    (if (marked object)
        (copy-to-new-import-list object)
        (progn (decrement-import-count object)
               (if (eq (import-count object) 0)
                 (send-decrement-message-to-contact object))))))
```

Figure 4.4. Required Modifications to Existing Functions

object is located. The import reference count is incremented in order to ensure the object will not be removed from this node while the export table still references it.

When a decrement object message is received by a node, the export object associated with the immediate object is found. The reference count associated with the export object is decremented. If the reference count of the object is zero, the object is garbage and can be removed. If the object resides on the current node, the object field of the export object points to the global object which is deleted and the associated memory is deallocated. The immediate object is forgotten and may be reused. The export object is cleared and added to the export table free list. If the object does not reside on the current node, the import table is searched to find

```
(defun decrement (object)
  (let ((export-entry (find-export object)))
    (decrement-export-count export-entry)
    (if (eq (export-count export-entry) 0)
      (progn  ; have to free export table entry
        (if (local object)
            (reclaim object)
            (progn (decrement-import-count object)
                   (when (eq (import-count object) 0)
                     (progn
                       (send-decrement-message-to-contact object)
                       (reclaim-import object)))))
        (reclaim-export-object export-entry)))))

(defun import-object (object domain sender)
  (if (previously-imported-to-node object)
      (progn  ; already in import table, may be in domain
        (send-decrement-message-to-sender object sender)
        (if (not (previously-imported-to-domain object domain))
            (add-to-import-list object domain)))
      (progn (add-to-import-table object sender)
             (add-to-import-list object domain))))
```

Figure 4.5. New Functions Required by Collector

the associated import object. The import reference count is decremented and the exported field is set to NIL. The export object is cleared and added to the export table free list.

As an immediate object is imported it is added to the local import table. In order to determine if the object has been imported to the current node before, the import table entry array is searched. If the object has been previously imported an associated import object will exist. In this case, a decrement message is sent to the node that sent the immediate object to the current node, and the current domain's import list is searched to determine if the object has been previously imported to the domain. If the immediate object has not been imported to the current domain before, the import reference count is incremented and the import object is added to the current domain's import list. If the immediate object has not been imported to the node before, an import object is allocated from the appropriate import table entry array to store the object. The contact field is set to the node that sent the message containing the immediate object. The reference count is initialized to one and the exported field is cleared. The import object is added to the domain's import list.

The local garbage collection is responsible for decrementing the import reference count. When an immediate object is copied by the collector the corresponding pair in the import list is updated. The current domain's import list is searched until the desired pair is found and marked. When the garbage collection is complete, each pair in the domain's import list is examined. If the pair was marked by the garbage collection, the mark is cleared and the pair is copied into the domain's new heap. If the pair was not marked, the import reference count is decremented and the pair is removed from the domain's import list. When an import object's reference count reaches zero, the import thread is scheduled. The import thread is a special thread that searches the import table for import objects having a zero reference count. When such an object is found, the thread sends a decrement message to the contact, clears the import and adds the import to the import table entry's free list. The use of an additional thread allows the user program to be resumed with minimal delay by increasing parallelism and asynchronicity between the garbage collector and the user process.

The export reference count of an object represents the number of nodes that may contain references to the object and the import reference count of an object represents the number of domains on the current node that have references to the object. Unfortunately, there are other references that must be considered. Indirect references may be created when a global object references another global object. For example: an immediate closure references the domain in which it resides; an active thread references itself; a thread references the domain in which it is currently executing, and any domain through which it must return to successfully complete execution. The reference counts must take indirect references into account. Because there is no general way to determine when an indirect reference is created, special cases are used for each type of indirect reference. Fortunately, only the types of indirect references mentioned above are possible. When an operation creates or destroys an indirect reference, the export reference count is manipulated appropriately.

4.3.3 The Revised Algorithm

In the revised algorithm, only the import reference count is used; making this algorithm easier to understand than the initial version, though no more efficient. The data structures are the same as for the initial algorithm except for the export table which does not need to maintain a reference count. The initial algorithm has been implemented in CUS and is the algorithm discussed throughout the rest of the thesis. The modified algorithm is presented only as an aid to understanding the initial algorithm. The modified algorithm is as follows:

When an object is initially imported to a node, it is added to the local import table. The object's contact is set to be the node that sent the reference to the object.

When an object is to be sent from one domain to another, the reference count for the object is incremented before the message is sent. When a message containing a reference to an object that has been previously imported is received, a decrement message is sent to the node that has sent the reference. If the object has not been imported to the requested domain before, the local reference count is incremented and the object is added to the domain's list of imports.

When a decrement message is received by a node, the reference count for the object is decremented. Once the reference count for an object reaches zero, the import object can be reclaimed. If the actual object resides on the current node, it can be reclaimed. Otherwise, a decrement message is sent to the object's contact node.

The local garbage collection is responsible for determining when a domain ceases to reference an object. When an object is copied by the collector the corresponding entry in the current domain's import list is marked. When the garbage collection is complete, each entry in the domain's import list is examined. If the entry was marked by the garbage collection, the mark is cleared and the entry remains on the import list. If the entry was not marked, the reference count is decremented and the entry is removed from the domain's import list.

4.4 Examples

4.4.1 Example 1

This example uses a network with one node, node zero, containing one domain, domain A.

Consider the creation of a placeholder in domain A. The placeholder structure is allocated from the global heap, and an export object is allocated to point to it. An immediate object is generated to represent the placeholder and is imported into domain A. An import object is allocated from the array pointed to by import table entry zero. The import object is assigned a copy of the immediate object and a reference count of one. The contact for the import is the current node, node zero, because that is where the immediate placeholder came from. A pointer to the import object is added to domain A's import list.

When domain A loses its reference to the placeholder, the reference can be reclaimed. During the next local garbage collection, domain A's import list entry for the placeholder will not be marked. This causes the entry to be removed from the import list and the import reference count to be decremented. Because no other domains have a reference to the placeholder, the import thread is scheduled to reclaim the object. The import thread clears the import object's fields and sends a message to the contact node to decrement the placeholder's export reference count. When the message is received, the decrement causes the export table entry's reference count to become zero. The placeholder structure is freed, the export object's fields are cleared, and the export object is added to the export free list.

4.4.2 Example 2

For this example, consider the two node network shown in Figure 4.6. Assume there are threads running on both nodes. Domain A resides on node zero and is referenced on nodes zero and one. Domain B resides on node one and is referenced only on node one. The number associated with an export table entry represents the export reference count for the object. The numbers associated with an import table entry are the reference count and the contact node respectively. Consider the following sequence of actions with respect to this network.

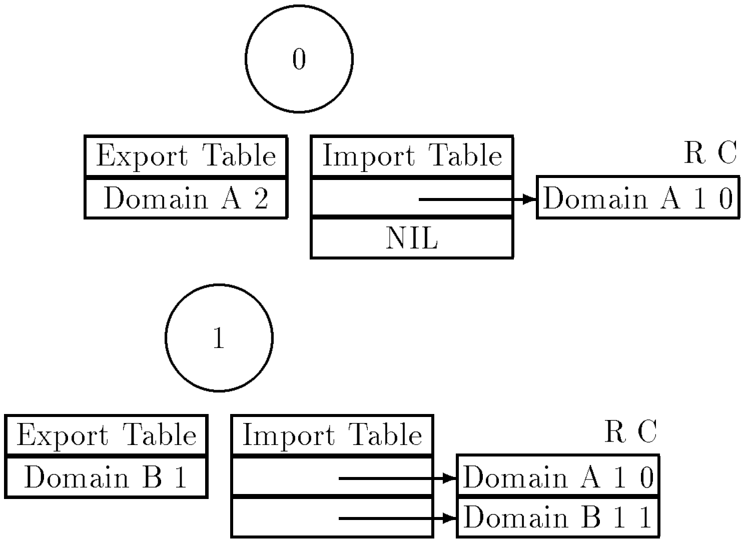

Figure 4.6. An Initial Network Configuration

Domain B exports a closure to domain A. An export table entry is created on node one, and an immediate closure is generated. The export table entry is assigned a reference count of one. The export table reference count for domain B is incremented because there is an indirect reference to the domain by the exported closure.

When the immediate closure is received by node zero, an import table entry is created for it. The import object has a reference count of one and a contact of one. The immediate closure is wrapped in a gateway and copied into domain A. The state of the network at this point is shown in Figure 4.7.

Domain B sends another copy of the closure to domain A. The local export reference count for the closure is incremented before the closure is sent. When the message is received, a decrement message is immediately sent to node one because the closure has already been imported by node zero. This causes the export reference count on node one to be decremented. The import reference count on node zero is not incremented because the closure has already been imported to domain A.

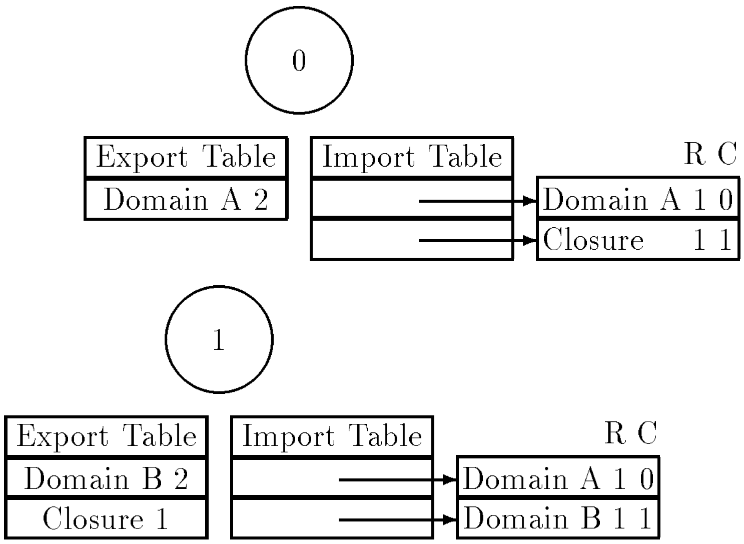

Figure 4.7. After Domain B Creates a Closure

Domain A discards its reference to the closure and performs a local garbage collection. The import reference count is decremented and, because the reference count is zero, a decrement message is sent to node one. When this message is received, the export reference count for the closure is decremented. The export table entry is cleared and the export reference count for domain B is decremented. This returns the network to the initial state shown in Figure 4.6.

4.4.3 Example 3

Consider the network of three nodes shown in Figure 4.8. Assume there are threads running on all three nodes. The export and import entries for the threads are not shown because they are irrelevant for this example. Each import table contains three entries, one for each node in the system. Domain A exists on node

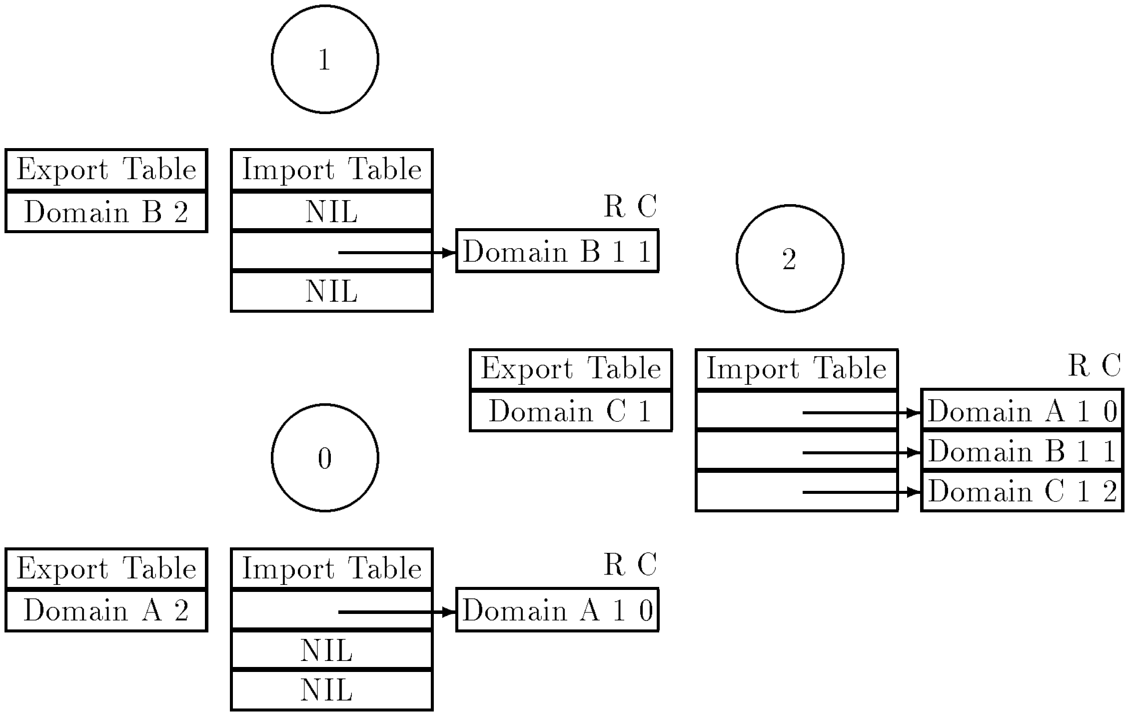

Figure 4.8. An Initial Network Configuration

zero and is referenced on nodes zero and two. Domain B exists on node one and is referenced on nodes one and two. Domain C exists on node two and is referenced only on node two. The number associated with an export table entry is the export reference count for the object. Both domain A and domain B are referenced on two nodes so their export reference count is two. Domain C is referenced on only one node, node two, so it has an export reference count of one. The numbers associated with an import object are the import reference count and the contact respectively. Each domain has been imported from the node on which it resides. Consider the following sequence of actions with respect to this network.

Domain C creates a placeholder and sends it, and a reference to domain B, to domain A. The new network state appears in Figure 4.9. An import object exists for the placeholder on node two because the placeholder was imported into the domain

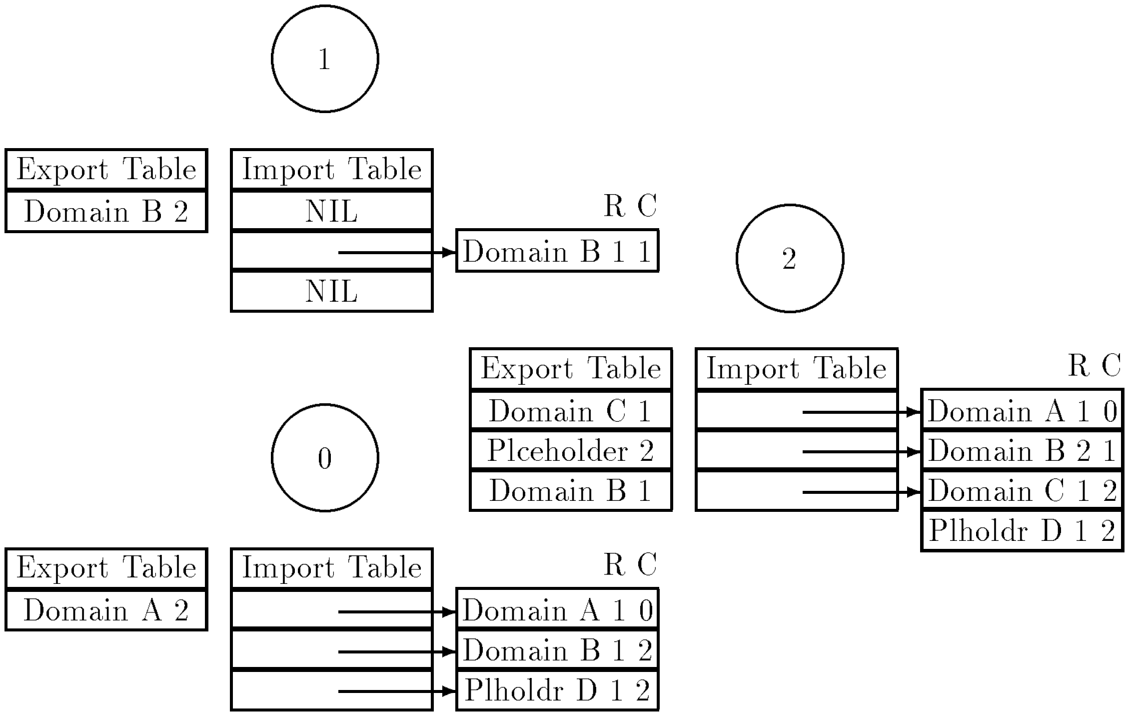

Figure 4.9. Network After Placeholder Sent to Domain A

that created it. The export entry for the placeholder on node two has a reference count of two because the placeholder was exported to both node zero and node two. The export table entry for domain B on node two was added because the domain was exported from a node other than the node on which it resides. This export caused the import reference count for domain B on node two to be incremented. Node zero has modified its import table to reflect the immediate objects imported into domain A. Note that the import objects are stored with the import table entry indexed by the node where the objects reside, not by the node that sent the object.

Domain C removes its reference to the placeholder and has a garbage collection. This causes the import reference count for the placeholder to be decremented. Because the import reference count becomes zero, the import can be cleared and added to the free list. The export reference count for the placeholder is decremented.

Domain A sends the placeholder to domain B and forgets its reference to domain B, changing the network state to that shown in Figure 4.10. An export object is allocated on node one for the placeholder because it does not reside there. The export reference count is set to one. The import reference count for the placeholder is incremented. Node one creates an import object for the placeholder imported to domain B.

Domain A performs a garbage collection causing the import reference count for domain B on node zero to be decremented. The import object is then cleared and added to the free list because its reference count reached zero. A decrement message is sent to node two, the contact for the domain. When node two receives the decrement message the export reference count for domain B is decremented. The import reference count for domain B is decremented and the export object is cleared and added to the free list. The current state of the network is shown in Figure 4.11.

Domain A loses the placeholder reference and performs another garbage collec-

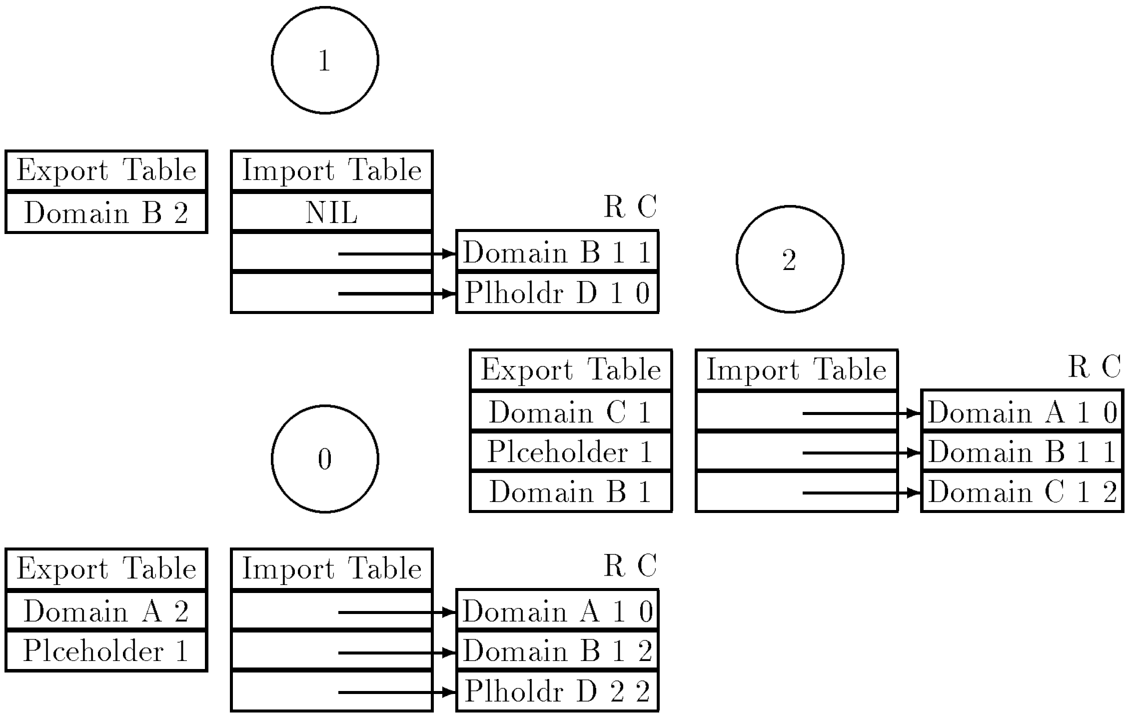

Figure 4.10. Network After Placeholder Sent to Domain B

tion causing the placeholder's import reference count to be decremented. Domain B loses its reference to the placeholder and performs a garbage collection. The placeholder's import reference count on node one is reduced to zero. The import object on node one is cleared and added to the free list. A decrement message is sent to node zero. When node zero receives this message the placeholder's export reference count is decremented. Because the export reference count is zero the export object is deallocated and the import reference count is decremented. The import object is cleared and a decrement message is sent to node two. When node two receives this decrement message the export reference count for the placeholder is decremented. The export object is cleared and the memory used to store the placeholder is deallocated. This returns the network to the initial configuration seen in Figure 4.8.

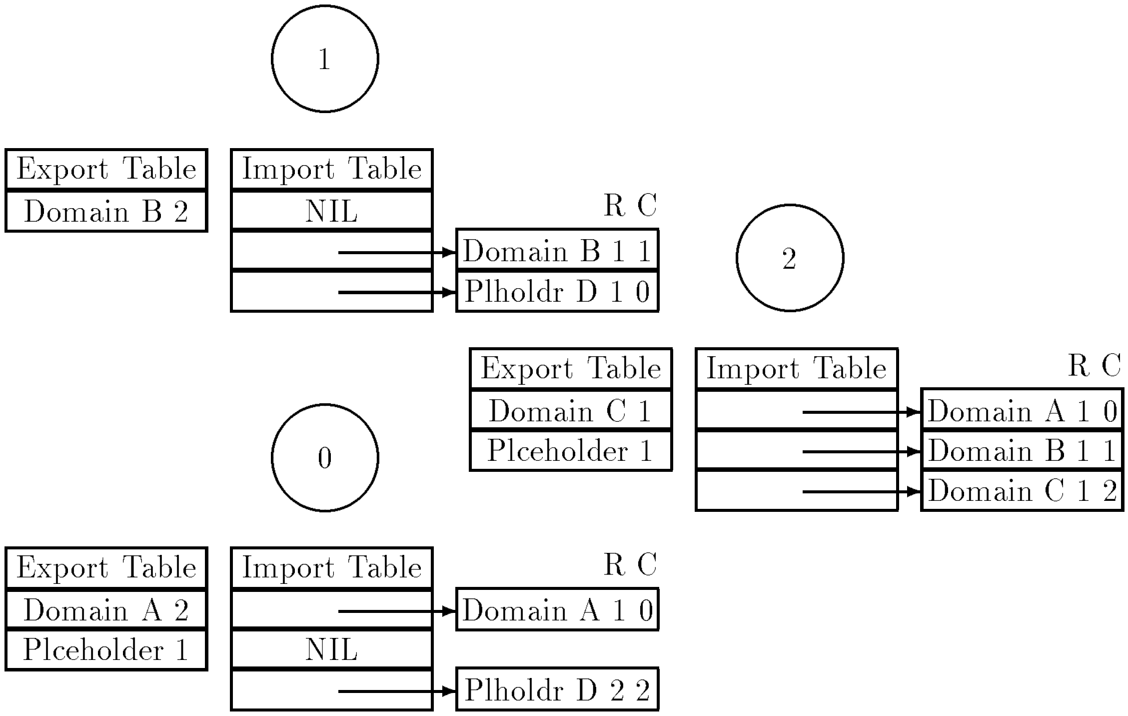

Figure 4.11. Network After Garbage Collection in Domain A

4.5 Proof of Correctness

Assumption 1 *The export table is contained within its own domain*

Assumption 2 *Determined placeholders are contained within their own domain*

Definition 1 `N` *is the number of nodes in the network*

Definition 2 `Import-count(i,o)` *is the import table reference count for object* `o` *on node* `i`

Definition 3 `Export-count(i,o)` *is the export table reference count for object* `o` *on node* `i`

Definition 4 `Mess(i,o)` *returns the number of messages sent from node* `i` *containing a reference to object* `o`

Definition 5 `Num(i,o)` *returns the number of domains on node* `i` *that reference object* `o`

Definition 6 `Ref(i,o)` *returns one if node* `i` *contains a reference to object* `o`*, and zero otherwise*

Definition 7 `Res(o)` *returns the number of the node on which object* `o` *resides*

Assertion 1 *Reference, and message, counts cannot become negative*

Assertion 2 *References to an object may be made by a domain or a message only*

Assertion 3 *An object* `o` *is garbage collected when Export-count(Res(o),o)* $\equiv 0$

Goal *The algorithm is correct if an object is collected if, and only if, there are no references to that object.*

Lemma 1 $\alpha(o) = \sum_{i=0}^{N-1} \textit{Import-count}(i,o) \equiv \sum_{i=0}^{N-1} Num(i,o)$

Proof:

When a message containing a reference to object `o` is received by node `i` and imported to domain `d`, the import reference count is incremented only if `o` is not currently referenced by `d`. Therefore, *Import-count(i,o)* $\geq Num(i,o)$. When a reference to an object `o` is removed from a domain `d`, the import reference count is decremented. Therefore, *Import-count(i,o)* $\leq Num(i,o)$. Therefore, *Import-count(i,o)* $\equiv Num(i,o)$.

Lemma 2 $\beta(o) = \sum_{i=0}^{N-1} \textit{Export-count(i,o)} \equiv \sum_{i=0}^{N-1} Mess(i,o) + \sum_{i=0}^{N-1} Ref(i,o)$

Proof:

Before a message containing a reference to object `o` is to be sent by node `i` to node `j`, the export reference count for `o` is incremented. The decrement message corresponding to this increment message will be sent either if `j` has a reference to `o`, or when $Num(j,o) \equiv 0$. If `j` had a reference to `o`, then there is always a message containing a reference to `o` in the network. If `j` did not have a reference to `o`, then the decrement will not occur until `j` no longer has a reference to `o`.

Theorem 1 *If an object is reclaimed, there are no references to it.*

Proof:

`o` is garbage collected $\Rightarrow \textit{Export-count(Res(o),o)} \equiv 0 \Rightarrow$

$Mess(Res(o),o) + Ref(Res(o),o) \equiv 0 \Rightarrow$

$Mess(Res(o),o) \equiv 0$ and $Ref(Res(o),o) \equiv 0$

$Ref(Res(o),o) \equiv 0 \Rightarrow \alpha(o) \equiv 0$

Theorem 2 *If there are no references to an object, then it is reclaimed*

Proof:

$\alpha(o) \equiv 0 \Rightarrow \sum_{i=0}^{N-1} Ref(i,o) \equiv 0$

$\sum_{i=0}^{N-1} Mess(i,o) + \sum_{i=0}^{N-1} Ref(i,o) \equiv 0 \Rightarrow$

$\beta(o) \equiv 0 \Rightarrow$ `o` is reclaimed

CHAPTER 5

ANALYSIS AND FUTURE WORK

5.1 Analysis of the Algorithm

The original version of the garbage collection algorithm has been successfully implemented in the Concurrent Utah Scheme environment. This implementation has been tested extensively, and manually validated to ensure all garbage objects, and no active objects, are collected. This validation was performed by observing local garbage collections to determine which objects were reclaimed, then examining the import and export tables to ensure the results were propagated correctly.

This algorithm has the ability to handle out-of-order messages. Out-of-order messages are problematic because they may cause an object to be reclaimed while it is still being referenced. In a reference counting algorithm, this problem arises when a decrement message arrives before the corresponding increment message has arrived. If an out-of-order decrement causes the reference count to reach zero, the object is mistakenly reclaimed. Because all increments occur locally, and atomically, in this algorithm, it is impossible for a decrement message to be received before the corresponding increment has occurred.

The asynchronous nature of this algorithm contributes to a network overhead of at most one message per reference. Every reference to a global object passed between domains requires a decrement. However, a reference passed between domains on the same node does not require that a message actually be sent. In this case, the decrement can be performed locally, by a thread operating in parallel with the user thread. Therefore, the number of decrement messages may actually be less than the number of interdomain references.

There are several features of this algorithm that enable it to scale to highly parallel systems. First, the ability to reclaim garbage without the use of global state information has a dramatic effect, because it ensures there will not be a node that becomes a garbage collection bottleneck. Second, the low communication overhead reduces the possibility of the network becoming a bottleneck. Finally, because reclaiming an object requires interaction between at most two nodes, the introduction of additional nodes into the system does not affect the efficiency of the collector.

There is very little computational overhead introduced by this collection algorithm due to its incremental nature. In addition, most of the work can be performed by threads running in parallel with the user process. The decrementing of export objects and the clearing of both import and export objects can be performed in parallel with the user process. This reduces the interruption of user processes by the collector and increases the utilization of the processor. Other operations such as importing an object and decrementing the import count of an object must be performed by the user process. However, these operations do not require much time, and occur while the user process is performing expensive operations, such as local garbage collection, so the increase in execution time is not noticeable.

5.1.1 Deficiencies of the Algorithm

The major problem with this algorithm is the inability to reclaim cyclic references. If closure A references closure B and closure B references closure A, the reference count for closures A and B will never be less than one. This means neither closure will be collected. The problem of cyclic references between global objects is unlikely to become significant. Although it is possible for a cyclic reference between global objects to be created it happens too infrequently to cause concern. It is important to note this problem does not affect the correctness of the algorithm. The algorithm describes a conservative garbage collector that will collect only objects known to be garbage. In the case of cyclic references the collector does not know the objects are garbage, so it will not collect them.

Unfortunately, the CS environment is too general to allow trivial detection or removal of cyclic references. If cyclic references become a problem, one of several, more complicated, techniques may be used to eliminate them. The common solution to the problem, using a marking collector as an auxiliary, is a possible solution. The auxiliary collector would be invoked only under certain circumstances, for example when there are a large number of global objects with low reference counts, and would reclaim all the garbage in the system. The problem with a marking collector is the high cost of forcing all the nodes in the system to perform a garbage collection at the same time. However, because the collector will be invoked infrequently, this overhead may be acceptable.

A less costly solution is used by Shapiro in his garbage collector. In this algorithm, if an object is not referenced on the node on which it resides, it is moved to a node that might contain a reference to it. The object may be moved to nodes that do not reference it, but, eventually, it will find a node that references it. Once the object is on the same node as its reference, it is easy to determine if a circular reference exists. The major cost in this algorithm is incurred by the movement of the object from one node to another, and will vary depending upon the object moved. For example, in most cases, it costs more to move a domain than a placeholder. This algorithm also has the unfortunate side effect of changing the degree of parallelism exhibited by a program. If domains become clustered on a single node, the majority of computation will be performed on that node, and the benefits gained by the distributed computing environment will be lost.

The algorithm requires local garbage collections to occur in order for global objects to be reclaimed. If these collections do not occur, it is possible for old references to objects to persist indefinitely. This prevents these objects from being reclaimed. In order to prevent this, it may be necessary to force local garbage collections to occur in domains that do not collect frequently. By forcing local garbage collections, it can be assured unused references to an object will eventually be removed.

The amount of memory used by the import and export tables may pose problems for user programs. Memory usage is high because individual table entries are large and many objects are used during program execution. The import and export objects do not need to be as large as they are. Memory can be saved by reducing each field in the object to the minimum required size. The size of the import table may be reduced by restructuring the table into a single array of imports containing a linked list for each node in the network. Even without the above modifications, the memory usage is not critical and is overshadowed by the algorithm's success at solving the problems posed by the CUS environment.

5.1.2 Possible Optimizations

Although the base algorithm has a very low network overhead, it is still possible to improve on it. One improvement is to reduce network overhead by piggy-backing decrement messages on other messages. This reduces the amount of information sent across the network by removing the header and trailer information contained in an individual message. Another improvement is to batch decrement messages to the same node. There are several different ways this could be done. A simple, but effective, solution is to store the number of decrement messages that should be sent to an object's contact with the object's import table entry. Then, when the import entry is freed, a single messages containing the number of times to decrement the reference count is sent to the contact node.[1] Another solution is to record the number of decrements associated with an object's contact, as above, but to change the contact whenever the object is received from a different node. This would be effective for those programs that exhibit locality of reference with respect to global objects.

Much of the computational overhead required by the algorithm could be eliminated by using a more efficient data structure. A significant amount of time is spent searching, adding to, and deleting from the import table. With additional research, it is likely an efficient hash function could be found. The use of a hash table would dramatically reduce the time required for import table manipulations.

During the execution of a program, it is possible for long `chains` of references to evolve. A `link` in the chain is formed when an object is no longer referenced on the current node, but is referenced by nodes which obtained their reference from the current node. Each link in the chain contains an import table entry for the object, which is only referenced by the export table. These links waste space because there are no references to the object on the node, only references the node has given to other nodes. It is possible to remove these excess references cheaply. Currently, when an object is dereferenced, a message is sent to the node on which the object resides requesting information about the object. The requested information is sent back as a new message. This existing communication could be used as a synchronization protocol for reducing chain size. If the node containing the object incremented its export reference count every time it received a request for information, and the node requesting the information updated its contact for the object when it received this information, chain size could be reduced with minimal additional cost. It is unlikely long chains will evolve in this environment

[1]This optimization has been implemented. See Section 5.1.3 for results.

because most nodes eventually dereference imported objects, and referencing an object prevents a node from entering a chain, except as the first element.

5.1.3 Timings and Measurements

A suite of six representative user programs was used to generate the test results discussed below. The test suite was executed using three different versions of CUS; a version of the orginal implementation without global gabrage collection capabilities, a version of the basic garbage collection algorithm, and an optimized version of the garbage collection algorithm. The test suite generates a total of 727 reclaimable global objects, all of which were reclaimed by both the basic and optimized versions of the garbage collection alogirthm.

Table 5.1 shows the timing results obtained for the algorithms' execution on a single node network. Both the base algorithm and the optimized versions perform well in this environment, increasing execution time by only 12% and 10% respectively. The optimizations do not appear to have a significant effect in a single node network.

However, on a multinode system, the optimized version drastically outperforms the base version. Table 5.2 shows that the base algorithm requires 144% more exectution time than the original implementation. This is over three times the

Table 5.1. Timings for Algorithms on a Single Node

	Original	Base Algorithm	Optimized Algorithm
Test 1	5,538	5,688	5,636
Test 2	6,698	6,790	6,653
Test 3	6,382	8,410	6,755
Test 4	1,870	1,911	1,936
Test 5	10,898	12,619	11,998
Test 6	5,587	6,163	7,626
Total	36,973	41,581	40,604
% Inc.	0	12	10

Table 5.2. Timings for Algorithms on Multiple Nodes

	Original	Base Algorithm	Optimized Algorithm
Test 1	6,116	8,815	6,956
Test 2	7,673	15,001	10,289
Test 3	6,722	7,198	6,808
Test 4	4,592	7,585	13,744
Test 5	13,552	16,852	16,427
Test 6	13,557	71,959	19,802
Total	52,212	127,410	74,032
% Inc.	0	144	41

increase of 41% required by the optimized version. A brief examination of Table 5.3 explains the drastic difference. The optimized version of the algorithm uses less than half the decrement messages required by the base algorithm. The difference in efficiency between the optimized and nonoptimized versions underscores the importance of low network overhead in a distributed garbage collection algorithm.

5.1.4 Comparison to Other Algorithms

Few garbage collection algorithms are appropriate for the Concurrent Scheme environment. The two most acceptable algorithms are Shapiro's algorithm and Bevan's algorithm.

Table 5.3. Message Counts for Algorithms on Multiple Nodes

	Original	Base Algorithm	Optimized Algorithm
Test 1	84	146	125
Test 2	75	135	114
Test 3	161	320	303
Test 4	222	445	335
Test 5	8803	20826	13693
Test 6	661	957	776
Total	10,006	22,829	15,346
Number Decr.	0	12,830	5,347

Although the two table approach used by Shapiro is similar to the initial version of the proposed algorithm, the resulting collectors are significantly different. Shapiro's approach uses only slightly more memory than the proposed algorithm, and reclaims all the garbage in the system; even cyclic references. The time stamp protocol used by Shapiro introduces additional operational, and network, overhead in an attempt to handle out-of-order messages. The result is a reduction in the overall efficiency of the algorithm. In a network where out of order messages cannot occur, the time stamp protocol can be removed; unfortunately, this is not the case in Concurrent Scheme. Because of the requirements of the time stamp protocol, Shapiro's algorithm is not as suited to the Concurrent Scheme environment as the proposed algorithm is.

Bevan's weighted reference counting algorithm appears to be an excellent solution to the problem of global garbage collection in Concurrent Scheme. There is one major problem with it, however: determining the size of the reference count. If the reference count is too small, a large number of indirection cells will be created, increasing network traffic to an unacceptable level. If the reference count is too large, memory is wasted. Adding complexity to this problem is the knowledge that the number of nodes used may greatly affect the number of references to an object. If the object is stored as a global variable, each node in the network will have at least one reference to it. However, the number of nodes in the network is not determined until run-time. Due to the wide range in the number of nodes in the network, from one to several hundred thousand, the choice of an appropropriate reference count size is almost impossible. In most situations, the proposed algorithm will require less memory, less network overhead, or less memory and network overhead than Bevan's algorithm. Therefore, it is more appropriate for the Concurrent Scheme environment than Bevan's weighted reference counting algorithm.

5.2 Future Work

There are several optimizations that have not yet been implemented. Some, in particular the restructuring of the import table, will definitely improve the performance of the algorithm. Others require further research to determine if the potential performance increase is worth the additional complexity. This research can be performed as the number, and complexity, of user programs increases.

Although this algorithm is currently implemented only in CUS, there is no reason why it could not be implemented in a wide variety of environments. For example, this algorithm may be used to implement the no-senders notification option for Mach ports [19]. In Mach, a task creates a port, to which it has read rights. It then distributes send rights to other tasks, which can also distribute send rights to other tasks. The no-senders option notifies the task with the read right when there are no tasks with send rights. Currently, this is done by passing a token message between each node in the system at least twice, a very inefficient method. An efficient garbage collection algorithm could be very useful in this situation. Further research should prove useful in determining other environments in which a distributed garbage collection algorithm would be beneficial.

REFERENCES

[1] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[2] Santosh G. Abraham and Janak H. Patel. Parallel garbage collection on a virtual memeory machine. *Parallel Processing*, 1987.

[3] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, 1988.

[4] H.G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[5] D. I. Bevan. Distributed garbage collection using reference counting. *PARLE*, 2:176–187, June 1987.

[6] Daniel G. Bobrow. Managing reentrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):270–273, 1980.

[7] D. R. Brownbridge. Cyclic reference counting for combinator machines. In *Functional Programming Languages and Computer Architecture*, pages 273–288, 1985.

[8] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–364, 1981.

[9] Al Davis. Mayfly: A general purpose, scalable, parallel processing architecture. Technical report, Hewlett-Packard Laboratories 3L, 1990.

[10] Al Davis. Mayfly: A general purpose, scalable, parallel processing architecture. *International Journal on Lisp and Symbolic Computation*, 5(1/2):7–47, May 1992.

[11] Jeffrey L. Dawson. Improved effectiveness from a real time Lisp garbage collector. In *ACM Symposium on Lisp and Functional Programming*, pages 159–167, 1982.

[12] John DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, Digital, November 1990.

[13] L. P. Deutsch and D. G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[14] Edsger W. Dijkstra, Leslie Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[15] Daniel P. Friedman and David S. Wise. Reference counts can handle the circular environments of mututal recursion. *Information Processing Letters*, 8(1):41–45, 1979.

[16] Benjamin Goldberg. Generational reference counting: A reduced communication distributed storage reclamation scheme. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 313–321, 1989.

[17] P. Hudak and R.M. Keller. Garbage collection and task deletion in distributed applicative processing systems. In *ACM Symposium on Lisp and Functional Programming*, pages 168–178, 1982.

[18] John Hughes. A distributed garbage collection algorithm. In *Functional Programming Languages and Computer Architecture*, pages 256–272, 1985.

[19] Joseph S. Barrera III. A fast Mach network IPC implementation. In *Proceedings of the USENIX Mach Symposium*, pages 1–11, 1991.

[20] Niels Christian Juul. A distributed faulting garbage collector for Emerald. In *OOPSLA Workshop on Garbage Collection*, 1990.

[21] R. Kessler, H. Carr, L. Stoller, and M. Swanson. Implementing Concurrent Scheme for the Mayfly distributed parallel processing system. *International Journal on Lisp and Symbolic Computation*, 5(1/2):73–93, May 1992.

[22] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetime of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[23] Alejandro D. Martínez and Rosita Wachenchauzer. Cyclic refence counting with local mark-scan. *Information Processing Letters*, 34(1):22–35, 1990.

[24] Shogo Matsui, Yoshinobu Kato, Shinsuke Teramura, Tomoyuki Tanaka, Nobuyuki Mohri, Atsushi Maeda, and Masakazu Nakanishi. SYNAPSE: A multi-microprocessor Lisp machine with parallel garbage collector. In *Parallel Algorithms and Architectures International Workshop*, pages 131–137, May 1987.

[25] David A. Moon. Garbage collection in a large Lisp system. In *ACM Symposium on Lisp and Functional Programming*, pages 235–246, 1984.

[26] F. Lockwood Morris. A time- and space-efficient garbage compaction algorithm. *Communications of the ACM*, 21(8):662–665, August 1978.

[27] Marc Shapiro. A fault-tolerant, scalable, low-overhead distributed garbage detection protocol. In *IEEE 10th Symposium on Reliable Distributed Systems*, 1991.

[28] Marc Shapiro, David Plainfossé, and Olivier Gruber. A garbage detection protocol for a realistic distributed object-support system. Technical Report 1320, Institut National de la Recherche en Iformatique et Automatique, November 1990.

[29] Heonshik Shin and Miroslaw Malek. Parallel garbage collection with associative tag. *IEEE Parallel Processing*, pages 369–375, 1985.

[30] M. R. Swanson. *Concurrent Scheme, A Language for Concurrent Symbolic Computing*. PhD thesis, Department of Computer Science, University of Utah, Salt Lake City, Utah 84112, January 1991.

[31] M. R. Swanson and R. R. Kessler. *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 200–234. Springer-Verlag, 1990.

[32] Douglas M. Wasahbaugh and Dennis Kafura. Incremental garbage collection of concurrent objects for real-time applications. *IEEE Real-Time Systems Symposium*, pages 21–30, 1990.

[33] Paul Watson and Ian Watson. An efficient garbage collection scheme for parallel computer architectures. *PARLE*, 2, June 1987.

[34] Joseph Weizenbaum. Recovery of reentrant list structures in SLIP. *Communications of the ACM*, 12(7):370–372, July 1969.

[35] K-S Weng. An abstract implementation for a generalized dataflow language. Technical Report 228, MIT Laboratory for Computer Science, 1979.