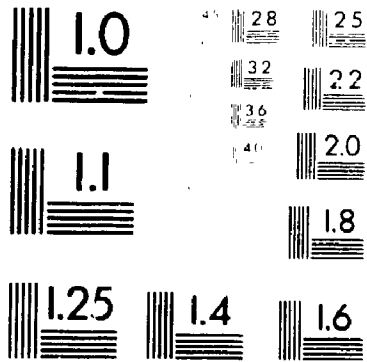


1



MICRO



**National Library
of Canada**

**Bibliothèque nationale
du Canada**

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Garbage Collection in a Multiprocessor Smalltalk System

By
John Duimovich, B.Sc.

This thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

School of Computer Science
Carleton University
Ottawa, Ontario

©copyright
1988, John Duimovich

12 January 1989

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.


ISBN 0-315-51159-1

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of this thesis

Garbage Collection in a Multiprocessor Smalltalk System

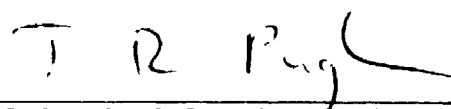
submitted by John Duimovich, B.Sc.

in partial fulfilment of
the requirements for the degree of
Master of Computer Science



Thesis Supervisor

Thesis Supervisor



Chairman, School of Computer Science
Carleton University
January 12 1989

Abstract

Garbage collection is an important feature of the Smalltalk programming environment. This thesis presents a multiprocessor garbage collection algorithm used in **Actra**, our multiprocessor Smalltalk system. The algorithm uses *Entry Tables* to determine interprocessor object reachability. The use of tables reduces interprocessor synchronization requirements allowing processors to garbage collect independently. This technique differs from other multiprocessor algorithms which require that all processors stop during garbage collection. A prototype of this collector has been implemented and is described in this thesis.

Acknowledgements

Thank you to my advisors Dave Thomas and Nicola Santoro for their guidance, patience and understanding. I would also like to thank the other Actra project researchers Wilf Lalonde and John Pugh for all their support through the years.

This work would not have been possible without the support of many people and organizations. Many thanks to Brian Barry and DREO, without their support this work would not have been performed. Additional thanks to Dy-4 systems which provided hardware, the National Research Council of Canada, and the many Actra project supporters. Special thanks to Digitalk for the use of their proprietary technology and to George Bosworth for his technical assistance.

I would also like to thank Mike Wilson for understanding all my raving moments. Thanks to Dave Thomson and Kent Johnson for keeping me sane by friendship alone. Thanks to Paul White for being a fun guy in my place while I finished this work as well as being my best thesis proof reader.

Thanks to Jenny for all her love and finally, thanks to my parents and family. I can honestly state for the record (my mother) that a thesis on garbage collection has nothing to do with Tuesdays garbage pickup.

Contents

1	Introduction	1
1.1	Garbage Collection	1
1.1.1	Here Come Multiprocessors	2
1.2	Garbage Collection: Some Definitions	2
1.3	The Garbage Collection Problem	4
1.3.1	Finalization	5
1.3.2	Forwarders	5
1.4	The Multiprocessor Garbage Collection Problem	6
1.5	Motivation	7
1.6	Contribution	7
1.7	Related Work	8
1.8	Thesis Overview	9
2	Uniprocessor Garbage Collection	10
2.1	Mark/Sweep	10
2.2	Reference Counting	12
2.3	Copying Collectors / Scavengers	14
2.3.1	Copying Collection Requirements	14
2.4	Baker Real Time SemiSpace	15
2.5	Vax-Smalltalk GC (Ballard)	19
2.5.1	Exit Tables	19
2.5.2	Object Table Entries (OTE) Recovery	20
2.6	Ungar's Generation Scavenger	21
2.7	Leiberman-Hewitt Lifetime Collector	21
2.7.1	Assumptions for Leibermann-Hewitt	22
2.8	Dijkstra Parallel Mark/Sweep	23
2.9	Summary	24
3	Multiprocessor Garbage Collection	26
3.1	Model and Definitions	26
3.2	Multiprocessor Issues	27
3.2.1	Object Movement	27
3.2.2	Synchronization	29
3.2.3	Garbage Collector/Mutator Interaction	29
3.2.4	Memory Contention	29
3.2.5	Language Issues	30
3.2.6	Smalltalk Language Issues	30
3.3	The Multi-Lisp Baker Scavenger	31
3.3.1	The Algorithm	31
3.3.2	Flip Synchronization	32

3.4	Synchronization Issues	32
3.4.1	Object Movement	33
3.4.2	Object Access	33
3.4.3	Scavenger/Scavenger Interaction	34
3.5	Jew's Multi-Generation Scavenging	35
3.5.1	An Overview of Jew's Algorithm	36
3.5.2	Using Remembered Sets	36
3.6	Smalltalk Synchronization Requirements	36
3.6.1	Object Movement: New Space Object Access Restrictions	38
3.6.2	Locking Requirement of Remembered Set Object Access	39
3.6.3	Remembered Set Accessing: Removal and Addition of Elements	40
3.7	Multi-Trash, The Multi-Scheme Garbage Collector	41
3.8	The Firefly Multiprocessor Garbage Collector	44
3.9	Multiprocessor Smalltalk	47
3.10	Summary	48
4	Entry Table Garbage Collector	49
4.1	Overview: Our Entry Table Garbage Collector	49
4.2	Why Entry Tables ?	52
4.2.1	Remote Object Updates Are Expensive	52
4.2.2	Reduced Synchronization	52
4.3	Definitions	53
4.3.1	Message Passing	53
4.4	The Scavenging Algorithm	54
4.5	Entry Tables and Remembered Sets	54
4.6	The Trade Offs Involved	55
4.7	Fast Entry Table Conversion	56
4.7.1	Lazy Conversion	56
4.7.2	Converting Between Entry Tables and Real Pointers	57
4.7.3	Converting between RPs and ETs	57
4.7.4	Maintaining the Shadow Space	58
4.8	Premature and Incorrect Tenuring	59
4.8.1	Reclaiming ET Cells	59
4.8.2	Entry Table Reclamation Without Extra Space	59
4.8.3	Object Tenuring and Removal Of Objects From The Entry Table	60
4.8.4	The 'Has.Scavenged' Flags	60
4.8.5	Reclaiming ET Cells To Old objects	61
4.8.6	Entry Table Reclamation Using Extra Space	61
4.9	Marked_By and Reached_By Bit Maintenance Rules	64
4.10	Cross Processor Object Access	65

4.11	Concurrency During Scavenging	66
4.12	Tightly Coupled Processors	67
4.13	Correctness Arguments	67
4.14	Summary	68
5	Actra: The Implementation	69
5.1	The Actra-Harmony System	69
5.1.1	Hardware	69
5.2	Harmony	70
5.2.1	Task Creation	70
5.2.2	Sending and Receiving Messages	70
5.3	The Entry Table Garbage Collector	71
5.4	The Entry Table Scavenger	71
5.5	Entry Table Manager: ET Creation and Deletion	73
5.5.1	Scanning the Entry Tables	75
5.5.2	Choosing Unique or Duplicate Cells	75
5.6	The Multiprocessor Interpreter	76
5.6.1	Interprocessor Message Sending	77
5.6.2	Task Structure	77
5.7	Reader and Writer Tasks	78
5.8	Concurrency During Scavenges	79
5.9	Tape and Glue: 'Putting It All Together'	79
5.9.1	Actors	79
5.9.2	Implementing Entry Table Cells as Forwarders	80
6	Conclusions	81
6.1	Overview	81
6.2	New Directions, Future Research	82
6.2.1	Hardware Solutions	83
6.3	The Last Word	84
A	Entry Table Source Code	89
A.1	Entry Table Scavenger	89
A.2	Reader/Writer Task	94
A.3	Examples From Class Actor	99
A.4	Example From Class Entry Table	100
B	Entry Table Recovery Example	101
B.1	Example: Reclaiming an Entry Table cell	101

List of Figures

1	Live/Dead Objects and Root set	3
2	Reference Counting Misses Circular Structures	13
3	Scavenging a Memory Region	14
4	Baker's Real-Time Algorithm	16
5	Object Access With Baker's Algorithm	17
6	Object Creation In Lisp Has Age Polarity	23
7	Using Remembered Sets To Keep Track Of External References . .	37
8	Memory Division In Multi-Trash	43
9	Memory Layout In The Firefly GC	45
10	Entry Tables Are Used Between Highly Mobile Spaces	51
11	Criteria For Adding To A Remembered Set	55
12	Entry Table To Real Pointer Conversion	57
13	Shadow Space Used For Entry Table Conversion	58
14	Reclaiming ET Cell To OLD During A Scavenge	62
15	Clearing Referenced_By Flags of Unreferenced ET Cells	63
16	The Entry Table Scavenger	72
17	Creating an Entry Table Cell	74
18	Entry Table Recovery Example: Initial State	101

Chapter One

1 Introduction

In this thesis we examine garbage collection algorithms. Specifically, the problem of garbage collection on a multiprocessor Smalltalk system is studied. We begin with a discussion of the motivation for this study — the reasons for garbage collecting in both uniprocessor and multiprocessor systems. An emphasis is placed on describing garbage collection in a multiprocessor Smalltalk system, however, the results are applicable to other garbage collected systems. We present an overview of various existing garbage collection strategies, their advantages and disadvantages and use this analysis to justify our design choices for the garbage collection algorithm presented. A prototype implementation is described in this thesis.

1.1 Garbage Collection

In modern programming systems, dynamic memory allocation and deallocation is provided through the use of system primitives such as *malloc* or *new* when allocating memory and the corresponding deallocation routine *free* for deallocation. The responsibility for the use of these memory management primitives is left wholly up to the programmer. In order to ease this burden, some systems such as Smalltalk [1], Lisp [2], and Mesa and Cedar [3] provide automatic stor-

age reclamation mechanisms called *garbage collection*. Early garbage collection algorithms were slow and as such considered a major bottleneck in an interactive environment such as a Lisp or Smalltalk system. More recently Baker [4], and Ungar [5] have presented fast garbage collection algorithms suitable for use in these exploratory programming environments.

1.1.1 Here Come Multiprocessors

As the cost of microprocessor technology goes down, it becomes more and more attractive to build multiprocessor systems. The literature contains many multiprocessor designs which claim mainframe performance from inexpensive microprocessors in multiprocessor configurations. Systems such as the BBN Butterfly, IBM RP3, Intel iSPC hypercubes, Motorola HYPERmodule, INMOS Transputer, the Connection Machine [6] have already been built. The reader is referred to [7] for a review of some commercially available parallel processor systems. Correspondingly, there has been a large increase in demand for languages to program these advanced computers.

The first efforts attempted to use parallel versions of standard languages, such as Fortran, and have had limited success in specialized applications [8]. Languages such as Lisp, (Connection Machine CMLisp [6], Multi-Lisp [9] and Scheme, (Multi-Scheme [10]) have all been implemented complete with garbage collectors. Current parallel Smalltalk projects like Actra [11], and Multiprocessor-Smalltalk [13] require high performance multiprocessor garbage collection strategies optimized for Smalltalk.

1.2 Garbage Collection: Some Definitions

In a garbage collecting system, memory is divided into *objects*. Objects may vary in size and contain either uninterpreted data or pointers to other objects. Objects which are still in use are called *live* objects, similarly, objects not in use are termed

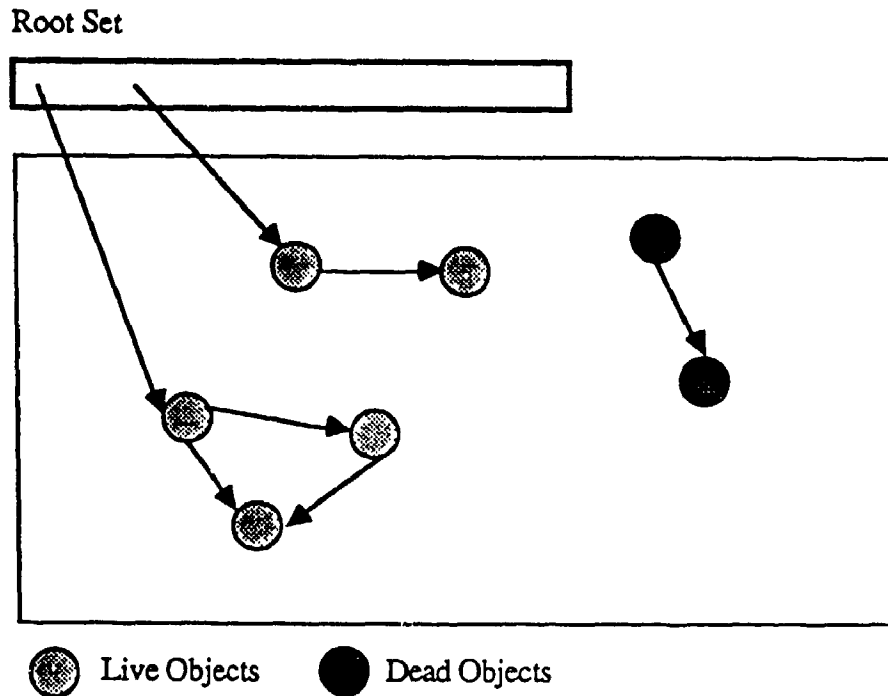


Figure 1: Live/Dead Objects and Root set

dead. The recovery of these dead objects is called *garbage collection (GC)*. Live objects are only those objects which are reachable, via some path, from a *root set* of objects, all other objects are dead. The root set is an *external*, well known collection of objects such as the run time stack which is used to trace out the set of live objects. The set of live objects may be represented as a directed graph with the objects as nodes and pointers to other objects as the directed edges in the graph. A *mutator* modifies the graph of live objects as it performs computations thus making some objects unreachable by its actions (i.e. the dead objects that must be recovered by the garbage collector).

1.3 The Garbage Collection Problem

The activity of garbage collection can be divided into three simple activities:

1. Finding live/dead objects;
2. Reclaiming the dead objects; and
3. Updating pointers to objects moved due to compaction or object movement.

Finding the graph of live objects involves tracing the reachable objects in the system from the root set. It is clear that to find such a graph, each live object must be inspected. This operation can be time consuming due to the large number of live objects in a typical system. Much of the research has focused on reducing the amount of work involved in determining reachability [14] [15] [16] [17]. Techniques such as reference counting attempt to reduce the disruptive pauses, common in some systems during garbage collection, by incrementally determining reachability using 'accounting techniques'. Other techniques, such as generation scavenging, combine statistical knowledge of memory usage with assistance from the mutator to reduce the amount of work required to determine the change in the reachability graph and thus the dead objects.¹ Other researchers propose method which use language specific knowledge to determine garbage collection requirements at compile-time instead of run-time [18].

Reclaiming dead objects is typically performed using a linear scan of memory to link all dead objects in a free memory list. Alternatively, a region of memory can be declared dead and completely reclaimed by removing all live objects from the region.

The issue of object movement and updating pointers occurs due to the possibility that memory may require compaction due to fragmentation. A compaction

¹Reference counting and generation scavenging are discussed more fully in chapter two.

phase effectively 'renames' all the objects in the system and requires that all referencing objects be informed about the name change.

1.3.1 Finalization

When a live object is no longer referenced by the system it may be desirable to be notified of the 'death' of the object, or more accurately, the imminent death of the object. Notification that an object is about to die is called *finalization*. A simple use of finalization can be found in Smalltalk systems with *Object Tables* (OT). When an object is no longer needed its OT entry can be reclaimed. This can be easily accomplished with finalization by reclaiming the OT entry of each of the objects which appear in the finalization queue. This feature can also be useful in systems which mix garbage collection and explicit memory allocation/deallocation. The garbage collector upon locating a dead 'to be finalized' object can perform any explicit deallocation required by the non-garbage collecting system. An example of this would be automatic file closing by the system, if the programmer forgets to close a file. We are especially interested in this technique due to the application of finalization to distributed systems. Finalization could allow the garbage collector to inform other processors/machines about changes in the state of an object's reachability. For example, a processor can use finalization to inform other processors about the non-reachability of a particular object from that processor. A simple implementation (does not handle cycles) of finalization can be found in Multi-Scheme.

1.3.2 Forwarders

A *forwarder* is an object which is a proxy for an object stored elsewhere in memory. Forwarders can be used to efficiently implement language features such as active objects and object mutation. Forwarders are important in garbage collection systems in that they provide a fast mechanism to update references to moved

objects without requiring the usual scan of memory. This is called a lazy update mechanism because updates are performed only when required and possibly not at all if the object being forwarded becomes garbage. A description of forwarders and their implementation can be found in [19].

1.4 The Multiprocessor Garbage Collection Problem

The basic problem of multiprocessor garbage collection is finding live objects in the system and reclaiming dead objects. The additional complications introduced by multiprocessor systems are:

- Synchronization of cooperative work among processors;
- Information sharing between processors; and
- Division of resources (memory, shared devices).

In a uniprocessor system, the garbage collector may assume that the graph of live objects will remain constant during a garbage collection. In a multiprocessor system, this may not be true, requiring that processors synchronize their activities so that a processor does not interfere with the work of another.

Independent processors must share information. Information flow between processors and data validity must be maintained. A processor P1 must be aware that decisions based on the state of another processor P2 may be invalid due to a. v actions by P2 which change that state. A large portion of time may be spent insuring that all processors have consistent information. For example, a processor may trace the reachability of an object that P2 claims is reachable but before the trace is finished the object dies and the traced path may not be alive. More serious problems can occur when processors move objects and other processors try to use the object at the old location.

Memory is often divided between processors in multiprocessor systems. Determining the interspatial reachability of an object is more complicated since there

is a possibility that the path to that object may change while the determination is being made.

1.5 Motivation

A major motivating factor in the design of the multiprocessor garbage collection algorithm presented here has been the Actra project [11].

The goal of the Actra project is to design a multiprocessor Smalltalk system. An Actra multiprocessor workstation consists of one to ten MC68020 [33] processors connected on a VME bus. Interprocessor communication is via shared memory. These workstations are connected to other Actra workstations via Ethernet. The resulting system provides a local parallel computation environment, as well as a co-operative distributed object oriented system. Additionally, the system must be able to interact with the outside world via interprocess communication primitives. Multiprocessor communication between Smalltalk processors is done using the Actor model [41]. A powerful aspect of this design is that it allows Smalltalk Actors to communicate with other tasks in the system (even non-Smalltalk tasks), an important consideration in our design.

One of the requirements of Actra is that it have a simple, fast and reliable multiprocessor garbage collector. The Actra garbage collector must allow processors to be able to garbage collect independently of each other, as it is expected that some processors may be creating much more garbage than other processors. Specifically, a processor which is creating lots of garbage should not cause the rest of the system to pause while it garbage collects.

1.6 Contribution

In this thesis we present a garbage collector which allows processors to garbage collect independently of each other. The use of *Entry Tables* for highly mobile objects removes many of the synchronization problems associated with copying

types of garbage collectors. Additionally, we have reduced the costs for accessing objects which are not being shared. The garbage collector is designed as a shared memory collector but has the attribute that it can be used as a non-shared memory collector as well. The idea that collecting remote references based on information provided by the remote garbage collector is important. We allow processors to garbage collect at any time and use incremental reachability rules to allow local garbage collectors to reclaim table entries using information provided by remote garbage collectors.

In summary, the major issues addressed by our garbage collector include:

- Processors must be able to garbage collect independently;
- Garbage collections should be fast;
- Synchronization costs should be low if sharing not used;
- Processor coupling strength should be defined by the user, not the garbage collector implementer; and
- External interfaces to other languages must be allowed.

1.7 Related Work

There have been many multiprocessor systems and garbage collector algorithms designed and implemented [13] [9] [10] [42]. A survey of these systems and others are described in this thesis. It is interesting to note that all these systems share one similarity in their approach to multiprocessing — they each attempt to make all the processors cooperate as a single monolithic (hopefully faster) unit. As such, much effort is placed in load balancing, work distribution, and fair resource sharing but little effort is made in improving the programming model for multiprocessor applications. For example, in Millers Multi-Scheme [10], every processor polls a global work queue in an attempt to help the system finish the ‘current’ computation sooner. Similarly, when the system garbage collects, all the processors stop and help.

In contrast, the Actra system has been designed to help cooperating processors accomplish specific tasks by division of the application into modular communicating entities (Actors). The communication patterns between these Actors are used to divide processor resources in an application specific manner. For this reason, it is a requirement that all processors need not participate in garbage collection so that slower garbage creating processors may continue processing without interference.

1.8 Thesis Overview

The thesis is organized as follows. Chapter 2 reviews the background material and classical garbage collection techniques. Chapter 3 presents an in depth study of the various multiprocessor algorithms. Chapter 4 presents our garbage collection algorithm. Chapter 5 describes the implementation of the algorithm and an extension to it which allows its use in non-shared memory systems. The final chapter presents the conclusions of this thesis, an overview of the contribution, as well as suggestions for future work.

Chapter Two

2 Uniprocessor Garbage Collection

This chapter presents a review of some uniprocessor garbage collection algorithms. Specifically, we discuss the basic issues in the design of the various uniprocessor algorithms. A discussion of various important collection techniques is presented with an emphasis placed on informing the reader of the application of these techniques to multiprocessor garbage collection.

2.1 Mark/Sweep

The classical mark/sweep algorithm first described by McCarthy [2] is named for its two major steps:

1. The *mark* phase performs an exhaustive search, marking all reachable objects from the root set.
2. The *sweep* phase finds all the unmarked (dead) objects by a linear scan of all memory and returns these object to the free memory pool.

Since memory is reclaimed on a per object basis, memory fragmentation can occur, possibly requiring memory compaction. Memory compaction could be required in order to satisfy a memory request when enough memory is available but

is divided among smaller free objects. A separate memory sweep is required to update addresses and copy the objects when compacting memory. The simplest implementations of the algorithm require that the mutator be stopped during garbage collection while more sophisticated implementations allow garbage collection to be interleaved with the mutator in an incremental manner.

The time/space requirements of mark/sweep vary from $O(1)$ extra space and $O(n^2)$ time to $O(n)$ extra space and $O(n)$ time where n is the number of objects in the system. This is dependent on the marking technique used. (A review of the various marking techniques can be found in Knuth volume II [25]). For example, there are marking algorithms which require no extra space, such as the reverse pointer technique. This technique results in the mark phase temporarily destroying the object structure and thus makes the reverse pointer technique particularly unsuitable for parallel garbage collection algorithms as many objects are made temporarily unusable by the collector.

An advantage of mark/sweep is that by walking every object in the system, dead objects can be identified easily, thus allowing for finalization. This, coupled with the fact that the implementation is very straightforward, makes this technique attractive. The disadvantage of mark and sweep is that it is slow, and a compaction phase requires an extra memory scan. This poor performance is due to the fact that the collector walks all live objects in memory during the mark phase, and all objects (both live and dead) during the sweep. Extra memory may be needed to implement the mark phase efficiently. Another problem is that virtual memory systems perform poorly under mark/sweep because the sweep phase accesses every object in the system (both dead and live) causing the paging system to read in all paged out memory. Many Smalltalk systems use a mark/sweep garbage collection phase to compact the image ² as it is the only garbage collection algorithm which will collect all garbage in the system.

²The image is a snapshot of a running Smalltalk system saved on disk.

Many variations in marking mechanisms exist. For an overview of the various mark/sweep algorithms see Cohen [20].

2.2 Reference Counting

In a reference counting system, every object contains a counter, representing the number of objects pointing to it, and is referred to as its reference count (RC). The reference count is maintained by the mutator as follows. Every time the mutator stores an object pointer into another object it must increment the RC of the stored object and decrement the RC of the displaced object due to the lost reference. When the reference count of an object reaches 0, the object can be reclaimed. Reclaiming an object requires that all of the objects references must also have their respective RCs decremented. This continues recursively until all zero RC objects are reclaimed. This recursive freeing is a costly operation if a large linked structure becomes garbage.

The reference counting strategy requires $\log N$ bits where N is the maximum number of objects that may point to an object. This requirement can be reduced by using a *sticky overflow* technique. For sticky overflow, once the RC of an object exceeds the maximum reference count, the count may not be decremented causing these objects to become permanent. In order to reclaim these sticky objects another technique such as mark/sweep must be used. The use of these sticky counts can be shown to be feasible in systems such as Smalltalk in which most objects have a reference count of less than eight [29].

An advantage of reference counting is that objects are reclaimed incrementally when they become garbage so that the garbage collection is non-intrusive, and thus suitable for interactive environments. The system also reclaims each object *individually* allowing the system to easily implement finalization. This allows the system to easily reclaim object table entries (OTE's) and is important since the first Smalltalk systems used Object Tables.

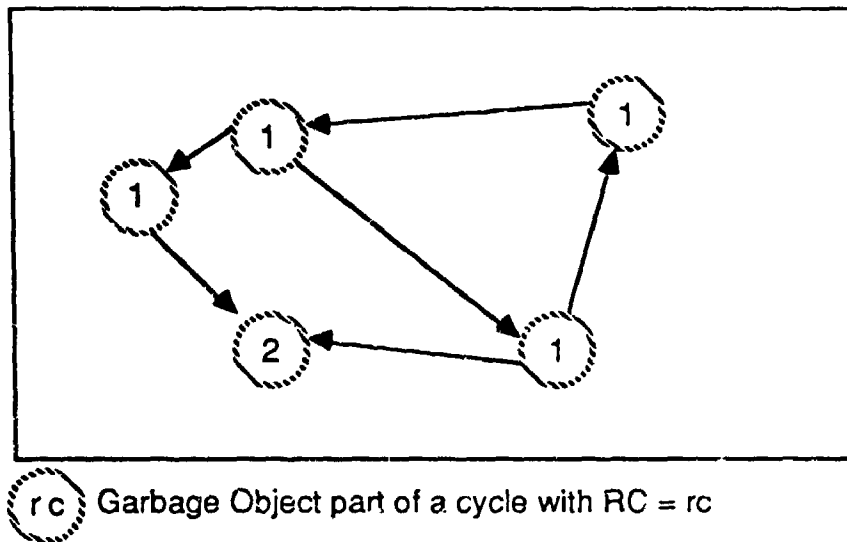


Figure 2: Reference Counting Misses Circular Structures

A major problem with reference counting is that it cannot reclaim circular structures (See figure 2). This is an unfortunate problem as it results in 'orphan' structures that must be reclaimed using other techniques such as mark/sweep. In systems that use this technique, the circular garbage problem can be minimized by the programmer explicitly unlinking the circular references³.

The performance of reference counting systems is poor due to the high accounting costs in maintaining the RC. Techniques for reducing the costs of maintaining the RC have been developed [15], which reduces the overall cost of reference counting but they do not reduce the expensive cost of recursive freeing of objects incurred when a processor wastes time walking 'garbage objects' rather than live objects.

³This type of code can be found as a historical artifact in the Smalltalk-80 Virtual Image.

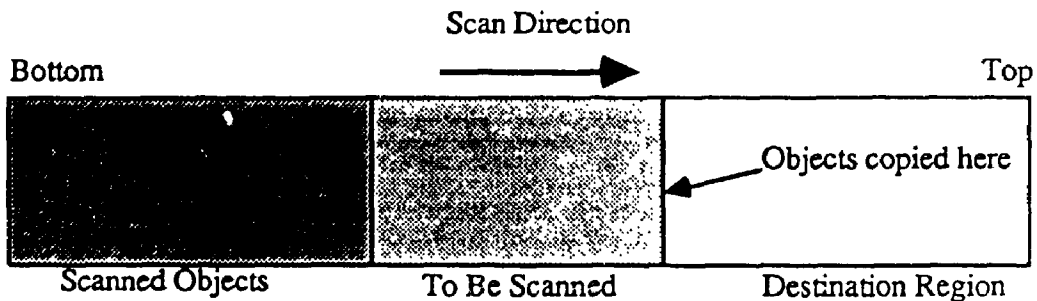


Figure 3: Scavenging a Memory Region

2.3 Copying Collectors / Scavengers

A broad range of collectors use *copying* to reclaim objects. In this technique, live objects are copied from the *source* space to the *destination* space and a *forwarding* object is left in the source object so other referents can find the copy. The destination space is scanned for references to objects in the source space which are then copied to the destination space. This technique is commonly referred to as *scavenging*.

Scavenging is the process of copying live objects out of a memory region. A scavenge begins by copying the root set of the source region into the destination region. The destination region is *walked* using a breadth first traversal (BFT). The BFT is accomplished by linearly scanning the destination space from low to high address. Objects are copied to the end of this destination region and thus are walked later. The scavenge is completed when the linear walk reaches the end of the destination region (See figure 3).

2.3.1 Copying Collection Requirements

The use of copying requires a memory space large enough to hold all the live objects in the source space. In the worst case, a memory region equal in size

to the source is required. When all live objects have been *evacuated* from the source region, the whole source region can be reclaimed. Note that the BFT can be accomplished without the use of additional memory, because the destination region is used as the unwalked object stack.

Copying can be done in a stop and copy manner or by interleaving scavenging with object allocation. In a stop and copy system, the mutator is halted while the garbage collector executes and then control is passed back to the mutator. An interleaving mechanism would copy a few objects every time an object is allocated thus resulting in an incremental behaviour. A forwarder is left behind so that all mutator references can find objects which have been forwarded. This interleaved collection/creation strategy can be made real-time by bounding the time allowed for object copying.

The major advantages of this type of algorithm are that memory compaction is automatic, circular structures are reclaimed and only live objects are walked. Unfortunately, scavengers have additional hidden housekeeping costs which are incurred by the mutator. These costs are in addition to the costs of copying and include indirection through forwarding pointers, and maintaining remembered sets or entry-exit tables. Memory usage is not optimal as a proportion of memory must be left unused to allow room for copying live objects. For example, in the following section we discuss the Baker Semi-Space algorithm which leaves unused at all times one half of total memory.

A review of the important scavenger algorithms is presented here. For an additional discussion of these algorithms see [23].

2.4 Baker Real Time SemiSpace

The Baker Semi-Space collector is a copying type collector. The memory is divided into two semispaces called *oldSpace* and *newSpace*. *newSpace* is further subdivided into three regions which are coloured black, gray, and white. Objects

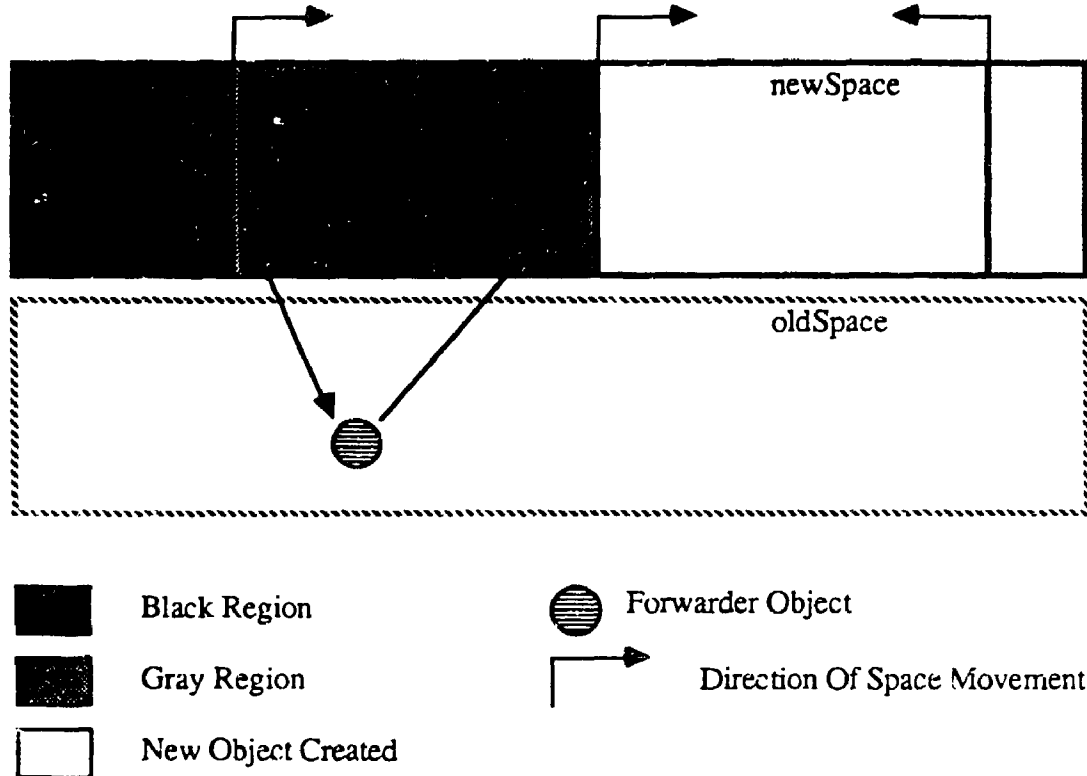


Figure 4: Baker's Real-Time Algorithm

in **newSpace** derive their colour from the space in which they reside, objects in **oldSpace** are uncoloured. All objects are allocated in **newSpace** with the following invariants maintained:

- Black objects may contain references to objects in the black, gray or white region. That is, black objects may only point to objects in **newSpace**;
- Gray objects may point anywhere (**newSpace** or **oldSpace**); and
- The white region contains newly allocated objects.

The algorithm works as follows. A gray object is made black by insuring all its references are into **newSpace** (gray, black, or white). This work is performed by a scavenger which walks from the low end to the top of **newSpace** converting the

```

fetchField (Object,index)
" fetch the field of the object at index "
IF IS_BLACK(Object) THEN
    return (Object [index]);      "return directly "
ENDIF
" must test and see if the object should be updated "
IF IS_IN_NEW_SPACE(Object [index]) THEN
    return (Object [index]);
ELSE
    " copy the object to newSpace OR return
    the forwarded (already copied) object "
    return (COPY_OR_RETURN_FORWARD(Object [index]));
ENDIF

```

Figure 5: Object Access With Baker's Algorithm

gray objects into black objects by copying any oldSpace references into newSpace. Once all the references have been copied then the scavenger colours the object black and moves on to the next gray object. Objects that are copied into newSpace leave behind a forwarding pointer so that other referencers may find the copy. Figure 4 shows the memory layout of Baker's algorithm.

Due to the presence of forwarding objects, the mutator must test whether the object has been forwarded and fetch memory from the correct location. External pointers may only reference into newSpace. Thus, every memory fetch from a gray object may require that the object referenced be copied from oldSpace into newSpace. It is copied to the end of the gray region for scanning later as it may itself contain references into oldSpace. Figure 5 shows the tests required when fetching from heap memory. Object access is an expensive operation hence hardware may be required for acceptable speed (This operation has been microcoded in some Lisp Machines).

When all oldSpace objects are copied to newSpace and newSpace is full a *flip* is performed. A flip exchanges newSpace with oldSpace and copies the root set

(machine registers and stack references) into newSpace. Additional objects are copied to newSpace by scavenging and by the fetchField operation shown above. If the root set of objects is large (for example an external stack), then the flip can be relatively expensive compared to an object allocation. Note that the old space was automatically compacted as it was copied to newSpace.

The Baker algorithm interleaves garbage collection with object creation. By bounding how much work the scavenger is allowed to do per object collection (eg. 10 microseconds time) the algorithm achieves real-time performance. The limitation to this real-time claim is that a flip may take a long time (much longer than a memory allocation). Large objects also pose a performance problem because the collector may not be able to copy the whole object in less than the bounded time. It should be pointed out that this algorithm was really designed for Lisp where large objects are rare and as such ad-hoc solutions are feasible (large objects can be kept in a separate space). Solutions to the large object copy problem include:

1. A split object representation that allows an object to be partially copied;
and
2. A separate space for large objects, this space being garbage collected less often.

Both solutions are complicated and result in additional performance penalties. Another issue encountered with this algorithm is the selection of the amount of time allocated to the scavenger during each allocation. If this time is too small then the object allocation space may collide with the gray region. This would leave no memory in newSpace to copy the rest of the objects in oldSpace and an 'out of memory' condition would occur. If the time given is too large, then real-time performance bounds will be correspondingly lower.

2.5 Vax-Smalltalk GC (Ballard)

The Vax-Smalltalk [22] garbage collector is based on Baker's algorithm. The basic ideas exploited in this collector are:

- the majority of newly created objects die soon after creation (for example, Points); and
- the majority of objects which survive past the high 'infant' mortality phase are static (semi-permanent) for long periods of time (for example, CompiledMethods).

Based on these assumptions, memory is divided into two spaces. A large separate space known as the Static Region is reserved for long lived objects and a smaller space known as the Baker space is used for object creation. The smaller space is garbage collected using Baker's algorithm. If an object survives more than sixty-three flips then it is *tenured* and moved to the Static Region. The advantage of this algorithm is that the long lived objects in the system are not repeatedly copied back and forth between the two flip spaces. Instead only the smaller region is garbage collected frequently. The fact that most newly created objects die soon after creation helps reduce the number of objects that must be copied every flip.

2.5.1 Exit Tables

Additional information is needed by the Ballard collector to insure that all references from the Static Region into the Baker space have been traversed when the flip occurs. To accomplish this an *exit table* of references from the Static Region to the Baker spaces is kept by the mutator as a root for garbage collection. This is required because the whole Static Region is not scanned for references into the Baker spaces due to time limitations. Instead, the mutator must perform some

'bookkeeping' at runtime and keep track of all the objects in the Static Region which point into the Baker space. This is accomplished by checking every object store; every time a new space object is stored into a Static Region object, the Static Region object is saved in an Exit Table which is scanned every flip to insure that all new space pointers are updated.

2.5.2 Object Table Entries (OTE) Recovery

Since Ballard's Vax-Smalltalk is an object table based system, the update of moved objects is fast, testing of forwarding bits is not required when referencing objects. However, the use of an object table requires the additional complexity that the OT itself must be garbage collected. Ballard's system has incremental collection of the OT entries and uses bits maintained by the scavenger to determine change of reachability from one scavenge to the next. For this purpose, the garbage collector flips are called odd or even. Each entry in the OT has an extra two bits (odd/even). Newly created objects are given the same colour as the current flip. Whenever the scavenger copies an object it sets the bit in that object for its current colour. Thus, all new objects and copied objects have the current flip bit set. During each flip, the OT is scanned and the opposite bit from the current flip colour is *cleared*. After two flips, an OT which is garbage will have both bits clear, and can be reclaimed. This requires a scan of the *whole* object table to be performed *every* flip, making flips very expensive (long pause during a collection). If a full scan is not completed by flip time it must either be finished or no OT entries can be reclaimed until two complete successive sweeps have been made. The solution to the problem of a potentially large pause during flips was to interleave object creation with the scanning of the OT so that the full OT is scanned by the time a flip is required.

2.6 Ungar's Generation Scavenger

Generation Scavenging [5] is based on the ideas presented in Ballard's Vax-Smalltalk and by Leiberman and Hewitt [40]. The major differences in Ungar's scavenger are that all pointers are direct, there is no object table, and the scavenges are performed in a stop and copy fashion. This means that forwarding is not required resulting in a 30 percent gain in system speed over OT based systems. The collector is fast; pauses are only a fraction of a second and require no hardware support. Other advantages include the reclamation of circular structures and efficient performance on systems with virtual memory. The performance claims are impressive, it uses approximately 3 percent of the CPU time in the Berkeley Smalltalk-80 implementation.

Ungar's and Ballard's algorithms are based on the same assumption that many objects are long lived and should not be copied every flip. Ungar uses *remembered sets* (Ballard calls them exit tables) to keep track of all the objects in old space that contain references to objects in newSpace. This remembered set is used as a root set and also to update pointers during scavenges. There is the same hidden cost as in Ballard's algorithm in maintaining remembered sets. Every store into an oldSpace object must test whether the object being stored is in newSpace. In order to minimize the number of tests required when storing objects, a large effort has been made to reduce this cost. One such optimization is to create the execution contexts in newSpace so that storing into method locals does not require testing.

2.7 Leiberman-Hewitt Lifetime Collector

The Leiberman-Hewitt [40] garbage collector is also a copying collector which uses statistical knowledge to reduce the amount of memory scanned by the garbage collector. The algorithm attempts to reduce the amount of memory wasted in Baker's algorithm by dividing the memory spaces into *generations* based on age.

The age of a generation is an indicator of the youngest object in that generation. The youngest generations are scavenged most often due to the high mortality rate of newly created objects, and correspondingly older generations are scavenged less frequently. Older generations which are found to be partially empty can be merged into a single generation with the age being that of the youngest generation of the merged group.

Objects in one space may only point directly to objects in spaces with the same age or older. Objects that point from an older space to a younger space must go through an *Entry Table*. This Entry Table pointer is an indirection pointer *into* a space. Entry Tables are used to reduce the number of objects that must be walked when scavenging.

Memory reclamation proceeds as follows. When space is needed, a young region is *condemned* and the objects in this memory region are evacuated out of the space thus freeing the space. Evacuation is performed by scavenging all the regions *younger* than the condemned space. These younger regions are the only regions which may point directly into the condemned region. The entry tables are used to find any *older* objects that reference into the region without having to scavenge that older regions themselves. Once all live objects in a space have been copied out, the region is empty. Forwarding pointers are left behind so that execution can continue during a scavenge. Hardware forwarding is essential here for performance (same requirement as the Baker Scavenger).

2.7.1 Assumptions for Leibermann-Hewitt

The basic assumption in the Leibermann-Hewitt algorithm is that most new objects point to older objects. This is true for functional languages, such as Lisp (for which this garbage collector was designed) because the destructive operations like `set-car!`, (`at:put:` in Smalltalk) are not used very often. When objects are created in Lisp (`cons a b`) the parameters to `cons` must exist before the `cons` cell

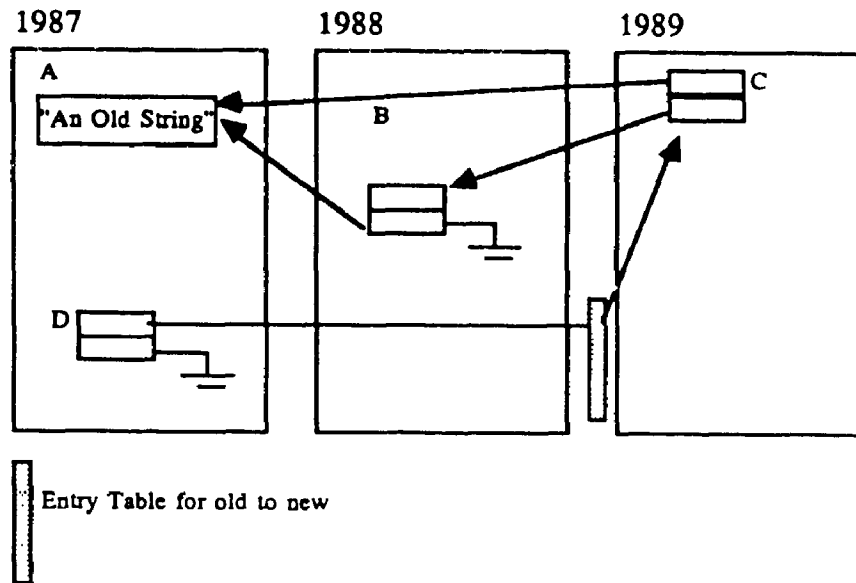


Figure 6: Object Creation In Lisp Has Age Polarity

can be created. Figure 6 show four objects A,B,C, and D. Objects B and C may point directly at A because they are older. Object D is restricted from pointing to object C because C is younger. Thus, if space 1988 were to be evacuated, space 1989 and any entry table into 1988 must be scanned to find all references.

In Smalltalk, the assumption that new objects point mostly to older objects is invalid due to the pervasive use of destructive operations, such as assignment. As a result, objects often contain a mix of both older and younger objects.

2.8 Dijkstra Parallel Mark/Sweep

The Dijkstra parallel collector [12] uses a two processor model, a mutator and a collector. All objects are marked either black, gray, or white representing the following states.

- Black objects are reachable and have been walked;

- Gray objects are reachable but have not been walked; and
- White objects are reachable from gray objects or are garbage.

The mutator *shades* objects that it uses. Shading turns white objects into gray notifying the garbage collector that it must walk that object. The collector sweeps memory repeatedly: when no gray objects are found during a sweep (this means that no additional objects need to be scanned) all the white objects are garbage. The advantage in this collector is that synchronization needed between the mutator and the collector is minimized. No explicit synchronization is needed between the collector and the mutator for garbage collection actions however, synchronization is needed for memory allocation.

Unfortunately, the algorithm is very difficult to implement and prove correct. The worst case behavior of this algorithm is very poor because marking 'on the fly' allows garbage objects to remain uncollected until two sweep phases later. This results in very poor performance to systems where garbage is created very quickly or the garbage collection processor is slower than the mutator. Smalltalk systems often use intermediate results and then throw them away immediately. These intermediate results must be protected by the interpreter by shading, preventing that object from being collected during the next sweep because it was reachable at some time during the mark phase. During the next phase, it will be collected. Performance analysis of this algorithm has shows this type of cyclic behaviour.

2.9 Summary

In this chapter we have presented a survey of many common garbage collection techniques. The approaches presented range from the brute force techniques such as mark/sweep, to incremental collectors such as reference counting and the copying collectors.

The traditional garbage collection techniques such as mark/sweep and refer-

ence counting suffer in that the algorithms process all objects, both live and dead, in the system. In systems which create large amounts of garbage quickly this has been shown to be inefficient. For this reason, these techniques are not considered the most appropriate for Smalltalk garbage collection.

A more suitable approach has been found in the copying collectors which improve performance because only live objects are processed. We believe that the copying algorithms are more promising for multiprocessor Smalltalk garbage collection designs. Other researchers have thought so also, and some of these algorithms have been extended to work on multiprocessor systems. In the following chapter we describe these extended algorithms and review their suitability for garbage collecting Smalltalk systems.

Chapter Three

3 Multiprocessor Garbage Collection

This chapter reviews several current multiprocessor algorithms which have influenced the design and implementation of our Entry Table algorithm. The algorithms are discussed in depth, so it is assumed that the reader is familiar with the algorithms presented in Chapter two. The applicability of each algorithm to Smalltalk systems is explored.⁴

We first present the multiprocessor garbage collection model and the various definitions used in describing the algorithms.

3.1 Model and Definitions

Consider a Multiple Instruction Multiple Data (MIMD) multiprocessor system containing two or more processor elements (PE). These PEs share one large address space. This memory is divided into smaller segments which are assigned to the processors. The assigned memory is termed *local* to that processor and implies ownership (even though other processors may access it directly). Usually this *local* memory is truly local in the sense that it is on the same physical circuit board as the processor and may have a faster access time or preferred access

⁴Caveat: Many of the problems described in this chapter are not problems with the GC algorithms themselves but on how they interact with the mutator.

rights. *Remote* memory refers to any memory which is not local to a particular processor (i.e. some other processor's local memory). Each of the processors may have one or more *mutators (interpreters)* and a garbage collector running on them. The term *mutator* refers to any process which is performing computation on the objects in memory. *Interpreter* is used to refer specifically to the Smalltalk interpreter. *Scavenger* is the term used to describe the scavenging collector for *local garbage collection (LGC)*. A synchronized *Global Garbage Collection (GGC)* will be used to describe a mark/sweep global collector with all processors participating as a unit. For our purposes, the difference between a LGC and GGC is that the global garbage collection requires all processors to synchronize during the complete collection phase.

3.2 Multiprocessor Issues

Before presenting the multiprocessor algorithms, we review some of the additional requirements of a multiprocessor garbage collection algorithm versus a uniprocessor algorithm. As the problems are described, simple solutions are presented. Further solutions and their ramifications are discussed in the context of the various algorithms described in this chapter.

3.2.1 Object Movement

Many of the techniques discussed in chapter two use object copying in their garbage collection strategy. A multiprocessor algorithm using these techniques (Baker's Real-Time, Ungar's Generation Scavenger) must insure that the object movement is atomic and that all processors are informed about objects which have moved. Preserving atomicity is important since the garbage collection process must be transparent to the user. Updating pointer information across all the other processors is essential for system integrity. If the system uses an object table, then updating pointers is unnecessary (they don't change) however atomic

object movement is still a problem (a processor cannot access an object while it is being moved). Potential solutions to the atomic object movement problem include:

1. Stopping all processors every garbage collect;
2. Locking the object on every access; and
3. Pure message sending.

The first option is the simplest solution, however potential concurrency gains are reduced. Stopping all processors trades concurrency speedup versus simplicity of the implementation, and reduced runtime synchronization costs. Many systems [13] [10] use this approach because it is a straight forward conversion from uniprocessor to multiprocessor implementations. This approach would be suitable if all processors have the same garbage creation rate and need to garbage collect at approximately the same time.

The second option offers a finer grained solution, every object access must lock out other processors. Object locking is an expensive choice because the system pays the price for sharing even when sharing is not used. A processor which accesses an object which may never be shared, will still require that object to be locked when accessed. Locking can be performed on a larger scale (i.e. memory region) to reduce the overall cost of locking; this could allow unrestricted access to a group of objects after acquiring the memory region thus removing the need for locks being used on objects inside this region.

The third alternative, pure message sending, is a mechanism which restricts other processors from directly accessing objects on other processors. To access an object, a message must be sent from the remote processor to the processor which owns the object. Total control over object access is given to the owning processor, thus removing the possibility of interference from other processors. This option

reduces some of the advantages of shared memory systems but has the advantage that object access and computation follow a well defined protocol and thus are easier to understand.

3.2.2 Synchronization

Multiple access by several processors requires some form of serialization or mutual exclusion (MUTEX). A review of mutual exclusion algorithms can be found in [31]. The frequency and type of access required determine the mechanism that can be used to synchronize processors. Operations that require MUTEX access of short duration can use simple *spin locks*. Spin locks waste processor cycles but may be faster in some cases where access time is short. Using spin locks may cause memory contention problems, as discussed below. Mutual exclusion of longer duration can use process based techniques like *semaphores*.

3.2.3 Garbage Collector/Mutator Interaction

A major concern in Actra is the synchronization due to the interaction between the garbage collectors and mutators (on separate processors). This synchronization is encountered when a processor is garbage collecting and requires access to another processor's resources. It is important that the scavenger not interfere with the effects of the remote interpreter. As an example, a garbage collector which is updating a field in object (because of an object movement) at the same time as a interpreter is using that object must not cause any work done by the interpreter to be undone. This is related to the atomic object movement requirement discussed above.

3.2.4 Memory Contention

Shared resources which require mutual exclusive access are potential performance bottlenecks. For example a single 'memory free' pointer could cause most object

allocations to wait for exclusive access to the pointer. One solution is to divide memory into chunks so that mutual exclusive access is not required as often. A processor acquire a chunk and then may freely allocate objects in that chunk. Memory can become fragmented using this approach depending on the chunk size. Memory usage may not be optimal because some processors may use less of their allocated chunks than others.

3.2.5 Language Issues

The interaction of the language with the memory system is also very important. The language for which the garbage collector is designed can make a large difference in performance. An example of this is object size. Languages like Lisp create many small sized objects, usually cons cells, which contain two pointers (eight bytes).

In a system like Smalltalk, the objects are larger, averaging fifty bytes. What this means in terms of the garbage collector is that moving objects in a Smalltalk system takes longer on average than in a Lisp⁵ system. The result may be that when moving objects, the wait for object access would be longer in Smalltalk leading to a less efficient system if for example a spin lock mechanism was used when waiting for an object to be copied.

3.2.6 Smalltalk Language Issues

The Smalltalk interpreter does not have concurrency at the bytecode level. Multiprocessing in Smalltalk still requires the use of traditional mutual exclusion techniques. Two Smalltalk processes (on different processors) may conflict when using a shared object. It is important, as discussed above, that the garbage collector not violate the bytecode atomicity. This is a difficult problem if multiple scavengers are allowed to update objects in Smalltalk space without stopping the

⁵Production Lisp systems have similar problems because they provide vectors and strings.

Smalltalk interpreter.

3.3 The Multi-Lisp Baker Scavenger

The first algorithm presented is Halstead's Multi-Lisp [9] scavenger running on the Concert multiprocessor system. The Concert multiprocessor system consists of 4-8 processor (MC68000) clusters sharing a sixteen megabyte address space. Up to eight clusters may be connected together using a 'RingBus' arbitrator thus allowing a maximum of 32-64 processors. Interprocessor communications is performed through shared memory. Local memory access times are two to four times faster than access to remote memory. Additionally, each processor may also have some private local memory (shadow memory), which is not addressable from other processors in the system. This allows more than sixteen megabytes in the total system with up to sixteen megabyte shared.

3.3.1 The Algorithm

The Multi-Lisp parallel garbage collection algorithm uses simple extensions to Baker's semi-space algorithm. Each processor has its local space divided into two equal halves and runs a Baker scavenger on those spaces. Recall the invariants for Baker's algorithm presented in chapter 2.

- Black objects may only point into new space;
- Gray objects may point anywhere; and
- White objects are new.

With multiple processors and multiple spaces, the invariants have been extended as follows.

- Black objects may only point to new space. This new space may be on *any* processor; and

- Gray objects may point to any space on any processor.

Each processor interleaves object creation with scavenging and must perform a flip when its new space is full. Scavenging on the processors can be done independently of the other processors but space flips must be synchronized across all processors. Object access by the mutators must test the forward bit as well as an additional lock bit which indicates that an object is being moved. When any particular processor wants to do a flip it must wait until the other processors agree to do the flip. This guarantees that none of the other processor's gray regions contain pointers into that processor's old space. In the worst case, a processor may be required to furiously blacken *all* of its new region (by removing the gray objects) so that it is prepared to flip.

3.3.2 Flip Synchronization

The main disadvantage of Halstead's algorithm is that all processors must perform their flips at the same time. This requires that they synchronize every flip. In fairness, flips are performed fairly infrequently (relative to the object creation rate) so the actual synchronization costs are actually not excessive. The real cost is that a garbage collect flip is expensive. Real time performance may be lost during a space flip. A flip involves copying all external references to the new space (this includes the run time stack, all external registers, and tables). Forcing all the other processors to undergo flips more frequently than necessary or waiting until they are ready to flip is both inelegant as well as inefficient. A processor which is creating lots of garbage will cause the other processors, which may not be creating as much garbage, to undergo flips too frequently.

3.4 Synchronization Issues

We examine the synchronization issues for this GC. We first describe the mutator/gc interaction in terms of a Lisp mutator. We then present the requirements

if this algorithm is used in a Smalltalk system. The interaction between multiple scavengers is also described.

3.4.1 Object Movement

When the Lisp interpreter accesses an object, it must be sure that the object will not move during that access. Halstead solves this problem by using a lock bit in every object. This is used to lock the object during access to prevent a scavenger from trying to copy the object to newSpace. The scavenger must test and set the bit before every object move.

3.4.2 Object Access

When accessing a field of an object, the Lisp interpreter must test if the object has been forwarded or is being copied. In addition, the mutator must set a bit indicating that the object is being accessed so that the garbage collector does not overwrite a field in that object. This would occur when a garbage collector is *blackening* a gray object. Any pointers into old space are updated to be into new space. It is possible for a mutator which is only reading a field of an object to not require this synchronization. This is because any scavengers updating that field may only write the *new forwarded address* into that field. Whether the mutator fetches the old reference or the new forwarded reference is unimportant since they are really the same object. Synchronization is only really needed when a mutator is writing into an object. This results in a race condition where the last writer wins (i.e. last update is the one that stays). This can be used to advantage in Lisp systems, because destructive operations are used less frequently and thus the costs may be reduced if MUTEX access is only used during writing. This is not the case in Smalltalk as assignment is a frequent operation.

3.4.3 Scavenger/Scavenger Interaction

When a scavenger is blackening an object which contains a remote reference, it has a few options.

1. copy an object to the remote processor's space;
2. request the remote processor to copy the object; or
3. copy the remote object to local space

The first choice, copying the remote object to the other processors space, would be the logical thing to do assuming the garbage collector is not responsible for object placement or 'load balancing'. Using this strategy, two processors must synchronize with each other so that both do not copy to the same space. This would require MUTEX access to the destination pointer and could result in a memory hot spot with multiple processors in contention for that pointer. A major problem with this strategy is that allocating local objects require MUTEX access to the memory free pointer thus slowing down every local memory allocation. In addition to this, the processors would have to run some sort of termination detection protocol so that another processor does not 'sneak' another gray object into a processors space after that processor thinks it has finished. This adds cost to every flip when the processors are waiting for everyone to agree to perform the flip. This type of termination agreement is not difficult and may be implemented using a simple shared memory location.

Using second option, if a processor is not allowed to copy into another processors space (to avoid the problems detailed above) then it must wait until the other processor copies an object to its new space. This may take much too long and invalidate the real time advantages provided by this collector.

The third option which is in fact the one used by the implementers is that each processor copies the object to its own space. The justification for this is that the 'locality of reference' will result in a performance gain due to faster

local references. This is not really true. The processor on which the object *v* will reside *most often* is the processor that has the reference 'earliest' in its memory (lowest in physical address space), *not* the processor which uses the object the most. This also could lead to the behaviour that an object is 'thrashed' by repeatedly being copied between competing processors. This strategy may also conflict with any load balancing which the system may perform. Another more serious problem is that the local processor may over-allocate its local memory by virtue of referencing too many remote objects. The implementers attempt to solve this problem by using an *extra global* memory space in which a processor may temporarily *borrow* some memory until the next flip. This extra memory must be available at all times and is unusable by the system except when a memory overflow occurs. This is particularly unattractive as the system already has a memory usage of less than fifty percent due to the use of Baker semi-spaces. If we were to apply this strategy to a Smalltalk system, a processor could conceivably become home to a large majority of the objects in the system. For example by referencing the Smalltalk system dictionary, a processor may copy the whole image to its local memory.

3.5 Jew's Multi-Generation Scavenging

Multi-Generation Scavenging [23] is an algorithm based on Ungar's Generation Scavenger extended for use on multiprocessor systems. This algorithm was designed for the Actra multiprocessor Smalltalk system but was never implemented. The Multi-Generation Algorithm was designed for the same hardware as our Entry Table algorithm and thus is of considerable interest here. The basic approach is to extend Ungar's Generation Scavenger to handle multiple processors by having additional remembered sets (RS) keep track of interprocessor space references. One remembered set is used to keep track of the interprocessor references between every pair of processors.

3.5.1 An Overview of Jew's Algorithm

The key idea in this algorithm is the use of external Remembered Sets. When a processor performs a local scavenge it must walk the external remembered sets and update any references into its space. These RS are used as both a root set as well as an update list. The other processors need not be concerned that a scavenge is occurring and in effect can keep processing. This design was chosen because it seems to require little external processor intervention when a processor performs a local GC. We first present a more detailed description of this algorithm and then analyse the synchronization requirements of this algorithm when used in a Smalltalk system.

3.5.2 Using Remembered Sets

In the Multi-Scavenger, a Remembered Set is kept between every pair of processors (see Figure 7). $RS(X, Y)$ is used to denote the RS on processor X which contains the objects pointing into Y's new space. Remembered Sets must be kept for Old to New references as well as for New to New references. Elements are added to this set as follows. When the mutator on processor A performs a store into O, of a non-local object R which is in processor's B new space, it adds O into $RS(A, B)$. A scavenger on processor B will walk $RS(A, B)$ when scavenging. This walk will find object O in $RS(A, B)$ and update the new address of R. If R is tenured and object O does not contain any other references to B then the scavenger may remove O from $RS(A, B)$.

3.6 Smalltalk Synchronization Requirements

The multi-scavenger was designed for use in a multiprocessor environment. We discuss the synchronization required by this garbage collector in terms of its interaction with the Smalltalk interpreter.

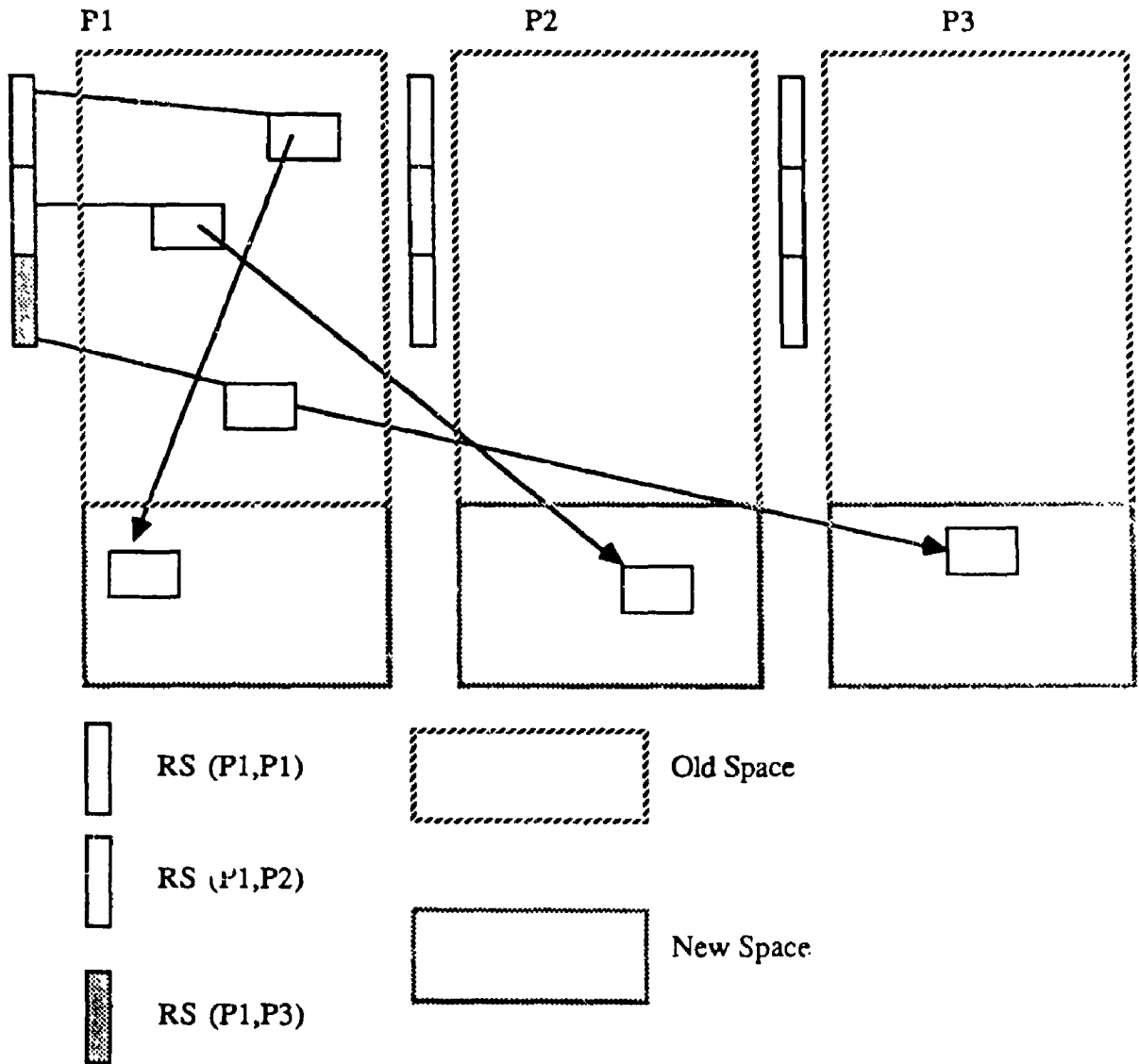


Figure 7: Using Remembered Sets To Keep Track Of External References

3.6.1 Object Movement: New Space Object Access Restrictions

The multi-scavenger algorithm allows other processors to continue execution while a processor garbage collects. Due to object movement in new space however, access to this space must be in a mutual exclusive fashion. Two types of cases occur:

- the local scavenger requires MUTEX access to all objects in new space during a flip; and
- cross processor new space access must prevent the accessed processor from scavenging.

The first case occurs when a local processor is scavenging. Other processor's may not access any objects in the scavenging processor's new space while that scavenger is copying objects.

The second case occurs when a processor is accessing another processor's new space. The accessing processor must lock the accessed new space against a garbage collection while it is accessing the object. This unfortunately requires a MUTEX operation for every new space access when performing cross processor references.

Let us examine the synchronization requirements of these cases. It is clear that the local processor need only lock the new space when it starts to scavenge. Thus, any other processor wishing to access that new space must check the state of the scavenger lock every object access. Additionally, the scavenger must wait for any processor currently accessing its new space to finish. This means that when any external processor references into a processor's local memory it prevents the local processor from garbage collecting, but not other processors that are just accessing. This can easily be implemented using a lock for processor pair. To lock out all new space accesses, the scavenger need only acquire all the

locks into its space. This type of synchronization is closely related to the multiple reader/single writer problem and can be called the multiple reader/single mover problem. Unfortunately, with this type of synchronization other processors may lock out the local collector for long periods of time. (Sometimes longer than it takes to scavenge). During a remote garbage collect, a processor could use special accessing techniques which are 'scavenger aware' however this solution requires additional synchronization between the interpreter and the garbage collector.

3.6.2 Locking Requirement of Remembered Set Object Access

A garbage collect cannot occur during the execution of a bytecode if it can affect its execution. In the multi-scavenger, a remote processor garbage collect may update a field of an object that is currently being used by some other processor. For example, processor A contains shared object S. Field 2 of object S contains a reference to a new object in B's new space. Thus, S is the $RS(A, B)$. The following steps occur.

1. Processor B is scavenging and is scanning $RS(A, B)$. The scavenger in processor B finds the reference in object S (field 2) and starts to copy it to the future survivor space.
2. Processor A performs storeInstVar 2 into object S. It stores an object (local to A).
3. Processor B has finished the copy and updates the field in S. (overwriting the previously stored local object from step 2).⁶

The above sequence of events demonstrate a mutator/garbage collector interaction resulting in a conflict. Note that this is different than two mutators both sharing object S and storing into the same field. This case is solvable by language

⁶Note that storing the future address before copying does not avoid this problem because the mutator must still wait for the copy to finish before using the object.

synchronization constructs such as semaphores and is independent of the garbage collection strategy. The mutator/gc conflict can only be solved by locking access to any Remembered Set objects while any processor is scavenging. This is the major problem with the algorithm. Essentially, the current *working set* of objects (all object in newSpace and the oldSpace remembered set) become inaccessible during any other processor's scavenge. This has the effect of halting work on any processor with objects containing external references when the owner of one of the external references scavenges.

As noted above, the multi-scavenger may update a field incorrectly if synchronization is not used to access Remembered Set Objects. This is because bytecodes are specified as atomic and that unless synchronization is provided by a remote scavenger the atomicity of the bytecode execution may be violated. This could cause a store to be 'undone' by another processor which is garbage collecting. Additional synchronization is needed when accessing the RS itself. This is described in the next section.

3.6.3 Remembered Set Accessing: Removal and Addition of Elements

As a processor continues execution, it may add elements to its Remembered Set. If a remote processor is scavenging when elements are added to the local RS, then the remote processor will have to be informed since it may have already walked that particular RS and thus may miss updating that object. Clearly some form of synchronization is required between the scavenger and any processor that adds to its RS. One solution to this could be to lock all the RS sets on all the processors and thus block any access to new objects. As each RS is walked, access to it may be given. Note that all the Remembered Sets into the scavenging processor's space must be locked at first. This is to prevent the possibility of a processor fetching a pointer that has not been updated from an unlocked RS object and storing into an space in which the RS has been walked. This communication

requirement reduces the potential performance of the remote interpreter since it must inform a remote scavenger (if any are active) when adding elements to its RS. Careful design of the RS update protocol can alleviate this problem but a synchronization step is needed regardless.

3.7 Multi-Trash, The Multi-Scheme Garbage Collector

The Multi-Trash [26] garbage collector is a parallel garbage collector designed for Multi-Scheme. Multi-Scheme is a parallel version of MIT Scheme running on the Butterfly multiprocessor ⁷. The Butterfly multiprocessor consists of nodes with MC68000 microprocessors and one megabyte of memory. The processors are connected via an omega network otherwise known as a butterfly network.

The Multi-Scheme system consists of a Scheme interpreter running on each of the processors. Each of the interpreters polls a global work queue for tasks to perform. The interpreters have access to one global heap which is further divided into sub-heaps. The sub-heaps are divided among the processors allowing each processor to allocate storage without having to gain exclusive access to a global free memory pointer. This reduces memory contention and potential memory 'hot spots'. The sub-heaps are fixed regions of memory which are further divided into partitions which are used to equalize memory distribution among the processors as described below.

The garbage collection technique is a stop and copy version of the Baker algorithm. When memory is exhausted on a processor all mutators are stopped. The processor which has run out of memory becomes the *master*. The master first scans all external roots. Each processor then scans their local stacks and their portion of the *Constant Space*. Constant Space is a region of memory containing objects which are permanent or *very* long lived. The portion of Constant Space scanned is determined at initialization and is modified every time Constant Space

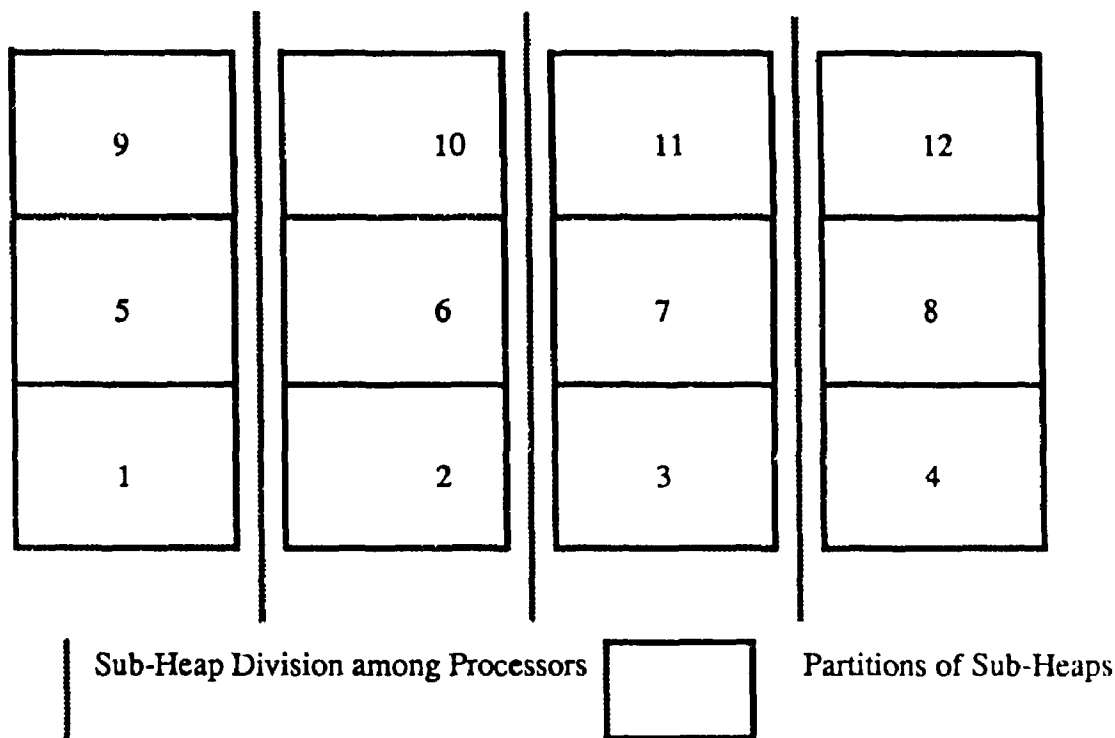
⁷Butterfly is a registered trademark of Bolt, Beranek, and Newman Laboratories.

grows. The heap is then scanned by each processor. In order to equalize the amount of work done by each of the processors, the heap is divided into equal partitions. When a processor is scanning one of these partitions, it stores the copied objects into a free partition (there will always be a free partition since memory is divided in half). When a scan fills a partition it is placed in the *ToBeScanned* list for other processors to possibly scan. All processors watch the *ToBeScanned* list until there are no longer any partitions left and garbage collection is done.

One of the requirements of this collector is that it balance the memory in each of the processor's sub-heaps so that after a flip a processor does not find its sub-heap full and trigger another GC. The technique used is a card dealing algorithm. When a processor requires another sub-heap partition to copy objects into, the partition it receives is from the next sub-heap that would receive a card if dealing a deck of cards amongst the processors. This insures that the number of partitions allocated from each processor's sub-heap differs by at most one.

A problem with this memory division is that memory may be wasted due to fragmentation in each of the sub-heap partitions. When copying to a partition, if an object does not fit into a partition, another partition must be allocated. The amount of fragmentation is dependant on the sub-heap size and largest object size. Due to the heap style allocation used, any holes in the partitions are unusable. See figure 8 for the memory division used in Multi-Trash.

One of the interesting things about this algorithm is how it avoids the atomic object movement problem as well as object access by stopping all mutators during a GC. This strategy is an example of a difference in philosophy in designing multiprocessor algorithms. The strategy here is to use all processors as efficiently as possible to quickly finish an unpleasant task, (garbage collection). Our algorithm allows independent garbage collection on any processor without affecting the other processors. The difference can be traced to the use to which we are



Numbers indicate order of allocation during a flip

Figure 8: Memory Division In Multi-Trash

putting our processors. The MIT machine is to be used as a monolithic Scheme engine used to run Scheme on every processor as quickly as possible. Our implementation is geared to permit each processor to be used for different types of tasks (including non-Actra processes) and allow independent processing as much as possible, allowing the user to decide interprocessor communication patterns.

There are some problems with the implementation. The first is that by using a stop and copy version of Baker the collector has major pauses (2-60 seconds) during collection. This would be unsuitable for a Smalltalk environment. Additionally, the technique of sub-heap splitting into partitions incurs additional memory costs due to memory fragmentation. The requirement that all processors stop during a garbage collect has been shown to be inefficient when some processors are greedy. This is clearly demonstrated from the example that Read-Eval-Print⁸ has a ninety percent memory usage on the processor which is running

⁸Read-Eval-Print is the main Lisp evaluation function.

it and fifteen percent memory usage on the rest of the processors. Due to the implementation of the MUTEX handling, memory is limited to twenty four bits of the address (top byte of a thirty-two bit address is used for Broken Heart MUTEX marker⁹).

3.8 The Firefly Multiprocessor Garbage Collector

The real-time concurrent algorithm presented by Ellis, Li, and Appel [42] is an implementation of Baker's real-time algorithm. The implementation uses the memory management unit (MMU) hardware to implement synchronization. For an overview of current MMU technology the reader is directed to [34] [32]. The primary idea is to use the MMU to prevent access to regions of memory that have not been updated by the garbage collector. The advantage of this method is that the mutators need not concern themselves about synchronizing with the garbage collector, the hardware insures synchronization. Whenever a mutator attempts to access an object which has not been updated, the hardware prevents access. The offending mutator is halted until the object becomes accessible (after being updated). The mutator can then be restarted at the fault address. Memory access is restricted as follows.

- The mutator sees only new space pointers in its registers;
- White objects contain only new space pointers (White, Gray, or Black);
- Gray objects reside in locked memory pages; and
- Black objects are freely accessible.

When access is attempted on a locked page (i.e. old space or a gray object), a processor trap occurs. The trap handling code scans the locked page insuring that all the pointers in that space point into new space (the page is made black).

⁹A Broken Heart is an invalid pointer

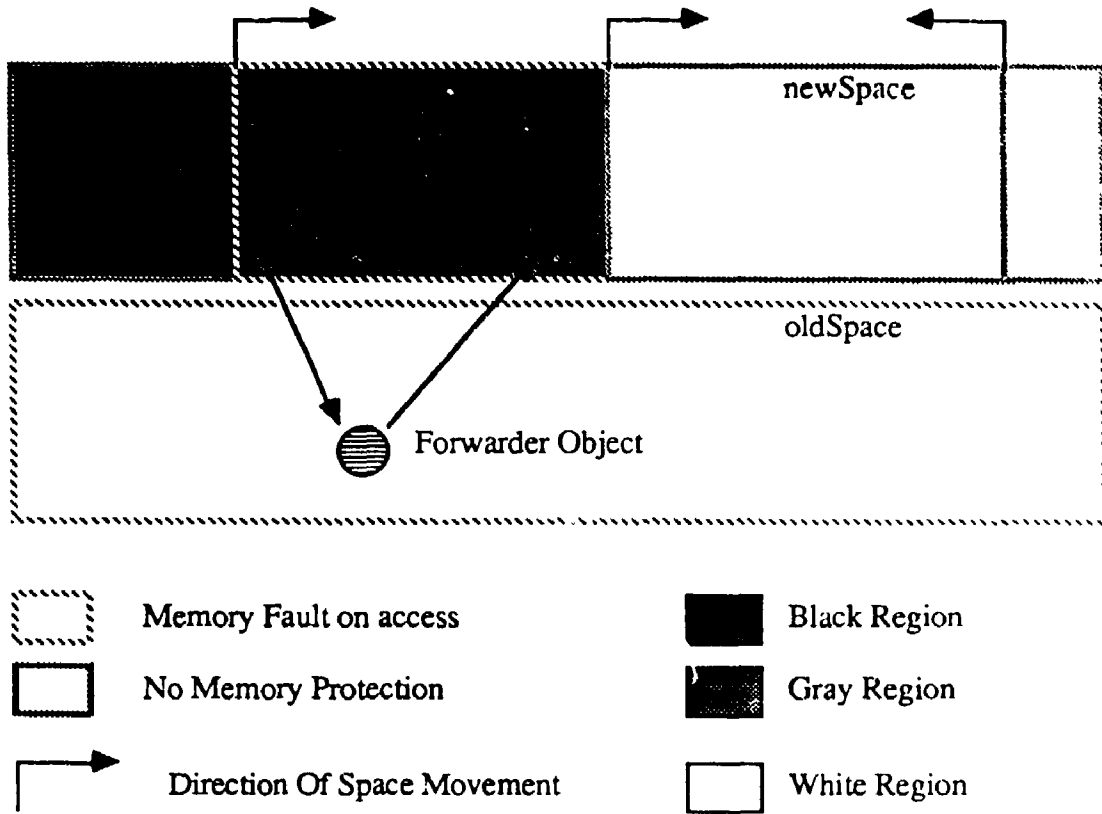


Figure 9: Memory Layout In The Firefly GC

The process which caused the trap can then be restarted. Since the mutator can only access black or white objects, it will only have new space pointers in its registers. A separate process scans the new space from low to high and converts gray pages to black pages. This is equivalent to the interleaving mechanism in the uniprocessor garbage collector.

As with the Multi-Scheme garbage collector, all mutator processes must be stopped during a flip. Since a flip may be expensive, the authors claim to have developed techniques for reducing the latency of flips. Mutator stacks can be considered as part of heap space and thus processed incrementally by the scavenger. Each stopped mutator must also have its registers updated. A lazy update mechanism is described. The authors state that the program counter of each stopped thread can be modified to point to a special routine which will update the registers when that mutator thread is resumed. The authors neglect to mention that if this technique is used then *every process must run* before the next mutator flip. If this does not occur then the forwarding pointers for the objects will be lost on the next flip. Using this trick will only defer the update of any pointers until the next flip (for threads that don't run as often as memory flips).¹⁰

Memory usage is also not optimal. For example, consider an unscanned page (gray) with only 1 object in it. If the mutator accesses this object, the trap will scan this one object and unlock the page. This makes this page unusable as memory for copying old space objects since access is unrestricted. Thus, the extra memory in that page would be left unused or added to a free buffer pool for allocation as new objects. This unfortunately is slower than heap style allocation used by the Baker algorithm in the first place. Theoretically, the amount of wasted space due to fragmentation would be $\text{NumberOfPages} * (\text{PageSize} - \text{MinObjectSize})$.

¹⁰It is tempting to call this a credit card optimization

The handling of derived pointers by this mutator is also inefficient. *Derived pointers* are pointers which reference into an object. Many compilers generate code which involves the use of derivative pointers for speed. For example, a loop referencing every object in a contiguous memory block may be more efficient if a derived pointer is used to sequence through the object instead of a base-offset addressing mode. When a pointer of this type causes a trap, the trap handling code must determine to which page the object belongs. A reverse linear scan of the page descriptors is required to determine which page starts with an object and then a forward scan to determine to which object the pointer belongs. While being sufficiently fast for small page sizes and small objects, large page sizes and large object sizes would make this linear scan too slow. The scavenging of active processes allows the garbage collector to falsely keep garbage objects alive. The scanning of stack frames may scavenge a stack frame after the process has returned from that frame. This only delays the reclamation of these objects by one scavenge and thus is not overly inefficient.

3.9 Multiprocessor Smalltalk

The Multiprocessor Smalltalk described by Pallas [13] is actually a description of how to convert a uniprocessor system to a multiprocessor system. Many of the ideas presented in this design have been described previously in [11]. Most of the multiprocessor issues are ignored or trivially solved using locks. Garbage collection in this system is handled by halting all processors and proceeding normally using Ungar's uniprocessor algorithm. Even memory allocation is handled by MUTEX access to a single memory free pointer. This project is in its early stages and thus has not had the time to address any multiprocessor garbage collection issues. The results presented focus on the changes required in the Smalltalk-80 virtual image when running with true multiprocessors. An example of such a change would be the different implementation of semaphores for multiprocessors

versus uniprocessors.

3.10 Summary

We have presented a review of the multiprocessor garbage collection algorithms found in the literature. In summary the existing algorithms are unsuitable.

The various implementations of Baker's algorithm waste too much space, and require additional hardware for performance. The multi-scavenger is too complicated to be feasibly implemented due to its many synchronization points. In all of the above algorithms, the extra costs for multiprocessors are paid even when shared memory objects are not being used. We refer to this as paranoid behaviour (the mutators are constantly looking over their shoulders in case a lgc is in progress). Problems with using an MMU include the extra costs for the page mapping tables and address translation times.

The algorithms presented also do not provided support for *shadowed* memory systems. These are systems in which for various reasons (such as performance or address space limitations) have part of their memory private to the local processor and unreachable to other processors.

Chapter Four

4 Entry Table Garbage Collector

This chapter describes our shared memory multiprocessor garbage collection algorithm based on using Entry Tables. First we introduce the terminology used to describe this algorithm. We then describe the Entry Table (ET) algorithm and how to garbage collect using an ET. The mechanisms for converting between ET cells and real pointers is presented along with detailed descriptions for maintaining the ET. Criteria for ET cell creation and reclamation are described, as well as strategies for ET recovery with and without extra memory. Finally, correctness arguments and concurrency issues are discussed.

4.1 Overview: Our Entry Table Garbage Collector

Our algorithm uses Entry Tables to keep track of interprocessor references. The use of tables for interprocessor references removes the need to update remote pointers during a scavenge. This allows a processor to scavenge without stopping any other processors. Cross processor new pointer references must use Entry Tables requiring pointer conversion to be performed during interprocessor message sends. This requires fast conversion between ET cells and real pointers. To this end, we exploit the local scavengers use of memory to implement a technique for fast two way conversion between ET cells without additional memory cost.

A simple mechanism for ET recovery using reference and marking information allows the garbage collection of the ET cells themselves, this is an important consideration when using tables. The approach for reclaiming tables is a major contribution of this thesis. The idea that collecting remote references based on information provided by the remote garbage collector is important. We believe that it is not necessary for a local garbage collector to scan other processor's memory in search for these references. In allowing processors to garbage collect at any time, any scanning performed by the local garbage collector would most likely be scanning a large portion of garbage. The alternative is to make the remote garbage collector share this information when it is most up to date (i.e. immediately after a remote scavenge) and for the local scavenger to use this information when it next needs it (i.e. its next lgc).

A major advantage of our algorithm is the reduction in synchronization required between garbage collectors and mutators. Reduced synchronization allows a local garbage collection to proceed without stopping any of the other processors. This is especially desirable if the processors have markedly differing memory usage patterns. Allowing independent garbage collection reduces the penalty for having *greedy* processors [26] which are creating much more garbage than other processors in the system. The key idea in our collector is that highly mobile objects, like new space objects in a generation scavenging system, should not be referenced directly. As was seen in Chapter 3, it is very expensive to update all the remote references when moving objects. This is due to the synchronization overhead as well as the need to scan potential referencing objects in other processor's spaces. Instead of direct pointers, we use entry table cells and a message sending protocol to access objects in another processor's new space. This guarantees that other processors will not be able to interfere with a garbage collector running on a particular processor and vice versa.

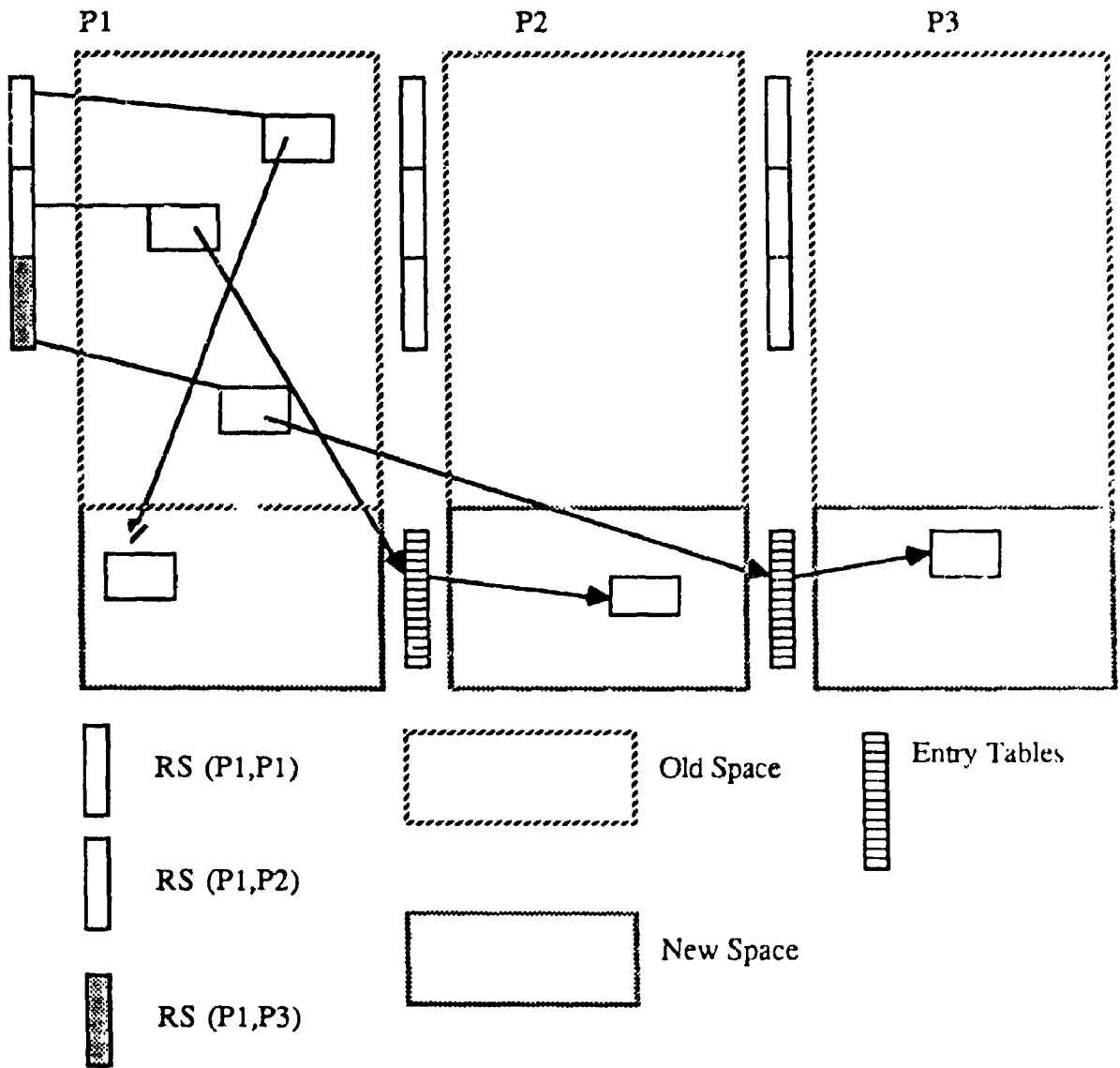


Figure 10: Entry Tables Are Used Between Highly Mobile Spaces

4.2 Why Entry Tables ?

The idea behind this algorithm is that updating the references on other processors is expensive due to the synchronization requirements for remote updates.

4.2.1 Remote Object Updates Are Expensive

The analysis of the multiprocessor scavenger showed that allowing the remote update of pointers, local to a processor's space, can cause a multitude of synchronization problems. We propose an algorithm using entry tables to reference objects that reside in highly mobile memory spaces while allowing direct access to spaces in which objects are fixed. The use of tables between old space (objects move infrequently) and new space (high mobility) removes the requirement of remote updates when objects move. A tenuring policy is used to allow ET objects that survive for a period of time to be promoted to old space thus resulting in faster direct access (non-message sending access). The local garbage collection strategy is a form of generation scavenging due to its excellent performance in Smalltalk systems.

4.2.2 Reduced Synchronization

Entry Tables eliminate the need to update remote references. The garbage collector will never write directly into a remote processors memory thus reducing the synchronization required between processors. Updates are atomic because only the entry table cells must be updated and access must be performed by the local processor. All remote access to objects pointed to by these cells must be made by message passing. In effect, the ET cells are acting as forwarders. A technique for efficient implementations of forwarder type objects is presented in [19].

4.3 Definitions

Processors in the system are numbered $1 \dots N$. The memory space M_i , of a processor i , refers to both old and new spaces for this processor. $MOld_i$ and $MNew_i$ are used to differentiate the two subspaces. ET_i refers to the Entry Table referencing objects in $MNew_i$. When referring to the entry table cell for a particular object O the denotation $ET_i(O)$ is used. Entry Table cells may also be generally referred to as *handles*. The Remembered Set $RS(MOld_i, MNew_j)$ contains cross space references from $MOld_i$ to $MNew_j$ for $1 \leq i \leq N$ and $1 \leq j \leq N$. *Scavenger(i)* is the local garbage collector running on processor i .

4.3.1 Message Passing

Message passing is used to describe the mechanism of communication between two processors. Message passing in this system has been implemented in two ways. The first mechanism requires a Smalltalk message send be invoked on the owner processor. A transfer of control between processors takes place, and execution continues on the processor to which the object belongs. This type of message passing can be implemented as Actors [11] where the receiver Actor resides on the target processor. The other message passing mechanism is an optimized version of the above. The local processor does not relinquish control when executing a method on a remote object. Any access to the objects fields involves communication with the owner processor. This type of message sending can be implemented using high speed reader/writer tasks that are used as object accessors thus allowing a processor to fetch fields without invoking a full remote Smalltalk interpreter. It is important to realize that message passing requires the participation of both processors.

4.4 The Scavenging Algorithm

The Entry Table algorithm is a simple extension of the generation scavenging algorithm. The algorithm steps are

1. Local processing stops;
2. The scavenging processor first scans its Entry Table for all externally reachable objects. Unreferenced handles are reclaimed;
3. The local root set is scanned by the local scavenger;
4. The destination flip space is scavenged for all references into the previous new space;
5. The scavenger recreates the Entry Table shadow space; and
6. Local processing resumes.

4.5 Entry Tables and Remembered Sets

Recall that Jew's multiprocessor scavenger algorithm requires that processor i maintain $RS(MOld_i, MNew_j)$ which contains all oldSpace to newSpace references. Our algorithm retains this requirement and extends the membership criteria such that $RS(MOld_i, ET_j)$ for $i \neq j$ is also maintained by processor i . This information can be maintained in two separate remembered sets or combined as one. $RS(MOld_i, ET_j)$ allows processor i to quickly find all its external references from $MOld_i$. This information is used to garbage collect the Entry Tables themselves and thus is a requirement in our algorithm. Unlike the multi-scavenger, $RS(MNew_i, MNew_j)$ need not be maintained because external references originating from $MNew_i$ are found during a scavenge. This has the benefit that the additional work is not required by the mutator during local new space access.

```
IF IS_IN_MY_OLD_SPACE(destinationObject) AND
   IS_NOT_IN_ANY_OLD_SPACE (storedObject)
THEN
   add_to_remembered_set (destinationObject)
ENDIF
```

Figure 11: Criteria For Adding To A Remembered Set

Local scavenging is very simple — the local scavenger scans the same root set as in the uniprocessor algorithm with the Entry Table as an additional root set member. This keeps objects, which are not referenced locally, alive, by virtue of being in that processors Entry Table. This approach has been called *prevention* [43]. When a local scavenger encounters a remote entry table reference it marks the ET as reachable but no other processing is performed.

4.6 The Trade Offs Involved

We have removed some of the complexity of local garbage collection by disallowing remote memory references. The trade offs involved in using Entry Tables are:

- Entry Table maintenance must be performed (this includes creation and reclamation of the entry table cells); and
- remote new space objects are accessed using message passing, direct access is not allowed.

Before describing the maintenance of these tables we will describe the restrictions placed on memory access. The major requirement of our entry table algorithm is that processors may only directly access objects in their own new space or any old space. For other objects, a message send to the processor owning that new object is necessary. While this may seem expensive, the alternative involves using locks during object access. Note the assumption that a lightweight

process/message sending kernel is available¹¹. Examples of such are Thoth [51], Harmony [49], and the V kernel [51]. In effect, we are trading speed of remote reference (direct pointers) with an indirect reference (a message send). This is in fact the synchronization point.

4.7 Fast Entry Table Conversion

The use of entry tables requires fast conversion between object address and entry tables.

This conversion is required when executing:

- a message send to an object in another processor's new space; and
- message result returned from another processor.

Every interprocessor message send would require that the receiver object and its parameters go through this conversion process. The message sending mechanism must convert the receiver of the Smalltalk message and its parameters into entry table cells (if necessary) and any handles that reside on the receiving processor be converted back into real addresses. Upon return, the result must also be converted into an ET or local reference if it resides on the sending processor.

4.7.1 Lazy Conversion

There exists methods for reducing the creation rate of Entry Table cells. For example, Smalltalk may return results which are discarded immediately. If the receiving process is informed that the return result is not required then the return value need not be converted into an entry table. This strategy of *lazy conversion* could be used to reduce unnecessary ET creation.

¹¹when referring to processes, lightweight means that the process shares the same address space as other processes.

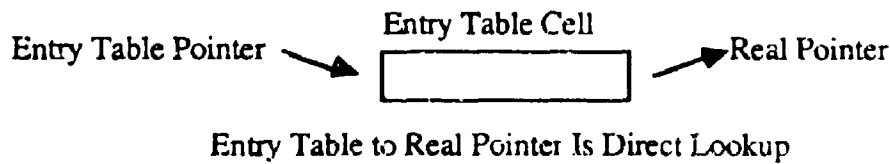


Figure 12: Entry Table To Real Pointer Conversion

4.7.2 Converting Between Entry Tables and Real Pointers

The forward conversion from ET to a real address is simple and straightforward. The address is simply found by de-referencing the entry table.

This is the same cost as an object table lookup that is used in many Smalltalk-80 systems. This process could be assisted by hardware (similar to Baker or Lieberman-Hewitt) implementations on some Lisp machines.

4.7.3 Converting between RPs and ETs

The conversion between real addresses and handles requires that it be determined whether or not an ET exists and if so return it, otherwise create a new ET. The simplest way of determining whether an ET exists is to increase an object's size and store the object's entry table with the object. Unfortunately this increases the object size of every object in the system. This growth in the size of an image is unacceptable because the expected number of ET's is small. The number of ET cells is expected to be small due to the tenuring policy.

Another possible solution used in other Smalltalk systems such as LOOM [29] are hash tables. Hash tables are feasible if the number of interprocessor references is low.

A third solution and the one chosen exploits the extra unused new space to

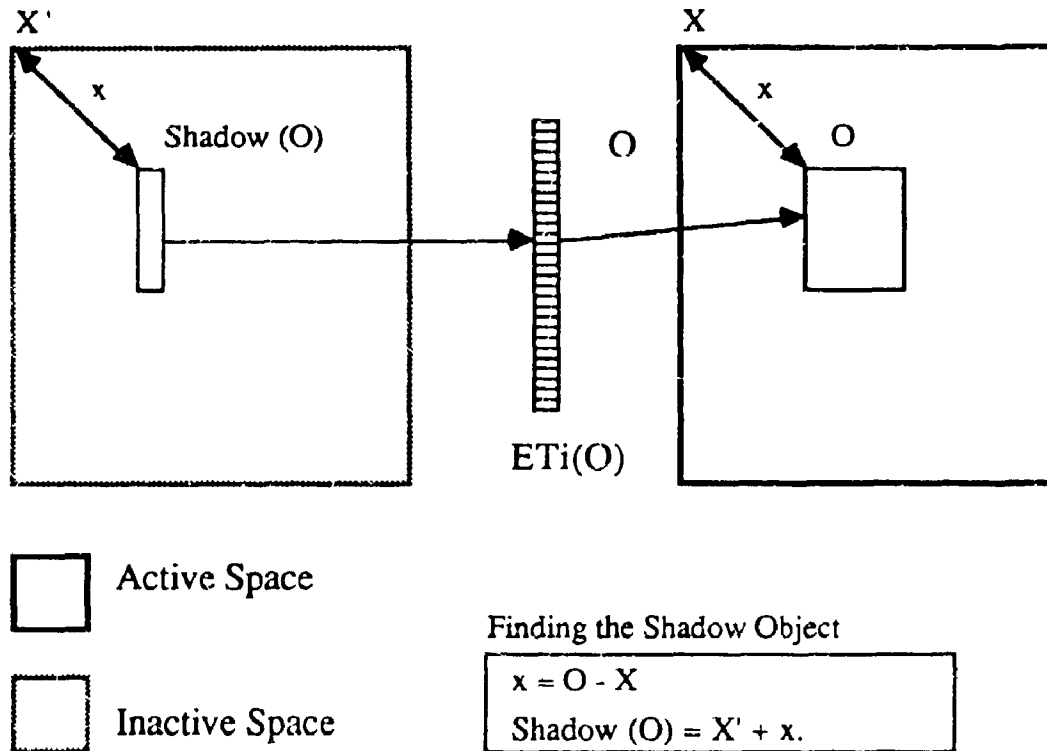


Figure 13: Shadow Space Used For Entry Table Conversion

shadow the active new space (see Figure 13). The ET slot for an object will be stored in the unused memory space at the same offset from the base of that space as the object is from the base of its space. The advantage is that this space is currently unused. In effect we get the extra space for free. A bit in each new space object is used to determine whether an Entry Table exists for this object.

4.7.4 Maintaining the Shadow Space

The additional work in this garbage collector is the maintenance of the entry tables. The first housekeeping task that the scavenger must perform is to rebuild the shadow space after a scavenge. This is necessary because the old shadow space is overwritten by the copied objects as the scavenger runs. Fortunately, the shadow space is easily recreated by scanning the ET after a scavenge and writing each object's ET value to the correct shadow locations. This scan can be

accomplished quickly and is dependent on the ET size for that processor.

4.8 Premature and Incorrect Tenuring

It is clear that by entering an object into the ET it will be tenured. This premature tenuring can cause the old space to fill too quickly and force the system to global garbage collect more often than necessary. This is undesirable because global garbage collects are time consuming and require the participation of all processors. The same problem is encountered when using Remembered Sets - storing a new object into a tenured object guarantees the tenuring of the new object, even if the old object becomes garbage. Similarly, creation of an Entry Table cell would guarantee tenuring if the Entry Table itself was not garbage collected. A technique for reclaiming ET cells is needed to handle the rapid turnover of ET cells expected because of the short lifetimes of newly created objects.

4.8.1 Reclaiming ET Cells

Entry Table cells must also be garbage collected. This is similar to the OT recovery necessary in Object Table systems. An example of this was described in Ballards algorithm. We allow Entry Table cells to be reclaimed using two different techniques. One technique made use of the simple observation that tenured objects do not need Entry Tables. A second technique uses additional memory to keep interprocessor referencing information to reclaim entry table cells more efficiently.

4.8.2 Entry Table Reclamation Without Extra Space

The Entry Table for an object is not required after the object to which it points has been tenured. Tenuring of the objects will fill the ET cell with an old space address. When every processor that references that particular ET cell has scavenged then the local scavenger can reclaim that entry table. This is described in

detail below.

4.8.3 Object Tenuring and Removal Of Objects From The Entry Table

As the objects age in the new space some of them will be tenured into old space. As this happens their ETs will become unnecessary since we allow direct old space referencing. When the remote scavengers run they will update the ETs to be the old pointer.

How do we know that an ET can be reclaimed? After an object has been tenured, it is clear that if every other processor has scavenged then the ET is not needed any longer and can be reclaimed. This is due to the fact that the other scavengers will have updated their pointers to point to old space and thus, have no need for that handle since old space is directly addressable. Thus, after every other processor has scavenged at least once the ET may be swept and cleared of old space references. This can be considered a form of synchronization but with very loose conditions. Every other processor must scavenge *at least once since that element in the ET had been tenured*. This requirement can be satisfied by designing a mechanism to request a scavenge from a remote processor.

4.8.4 The 'Has_Scavenged' Flags

Every processor contains a bit pattern Has_Scavenged (HS) which is used for indicating which processors have scavenged. *Has_Scavenged_i* is set to 1 when processor *i* has scavenged in the period of time since the local processor last cleared the flags. These flags are maintained by each processor as it scavenges. A remote scavenger will set a bit in each of the other processors Has_Scavenged fields notifying that it has completed a scavenge and is not currently scavenging. During remote scavenges, this bit is cleared to inform the local processors that

a scavenge is in progress.¹² The `Has_Scavenged` bit pattern is set to zero by the owner processor and thus can be used to determine if a remote processor has scavenged during some period of time. The setting and clearing of these bits must be performed under MUTEX access.

4.8.5 Reclaiming ET Cells To Old objects

The `Has_Scavenged` flags can be used to determine when all the other processors have scavenged. Every time a processor locally scavenges, it determines which processors have scavenged since it previously checked. After using the `Has_Scavenged` field, the field can be cleared. It is not known which processors contain references to the ET, and thus the processor must wait for *all* the other processors to scavenge before it can reclaim the ET. A local variable *Who_Has_Scavenged* is maintained for each processor to be able to clear out tenured Entry Tables cells. Figure 14 shows an example of this mechanism for ET cell recovery.

4.8.6 Entry Table Reclamation Using Extra Space

As we have seen, a processor cannot reclaim ET cells without some extra information. If every ET contained information on external referencers as well as reachability bits then a processor could reclaim ET cells containing `newSpace` objects during local scavenges.

In order to accomplish this, every entry table cell contains the following.

Real_Pointer This field of an ET contains the real object pointed to by the ET cell. This is used and maintained by the local processor for use in scavenging and for ET to real address conversion.

¹²It is unimportant that a remote scavenger may set and clear the `Has_Scavenged` bit of all the other processors many times before the local processor uses the bit field. The flag is set if a processor scavenged at least once and is not currently scavenging.

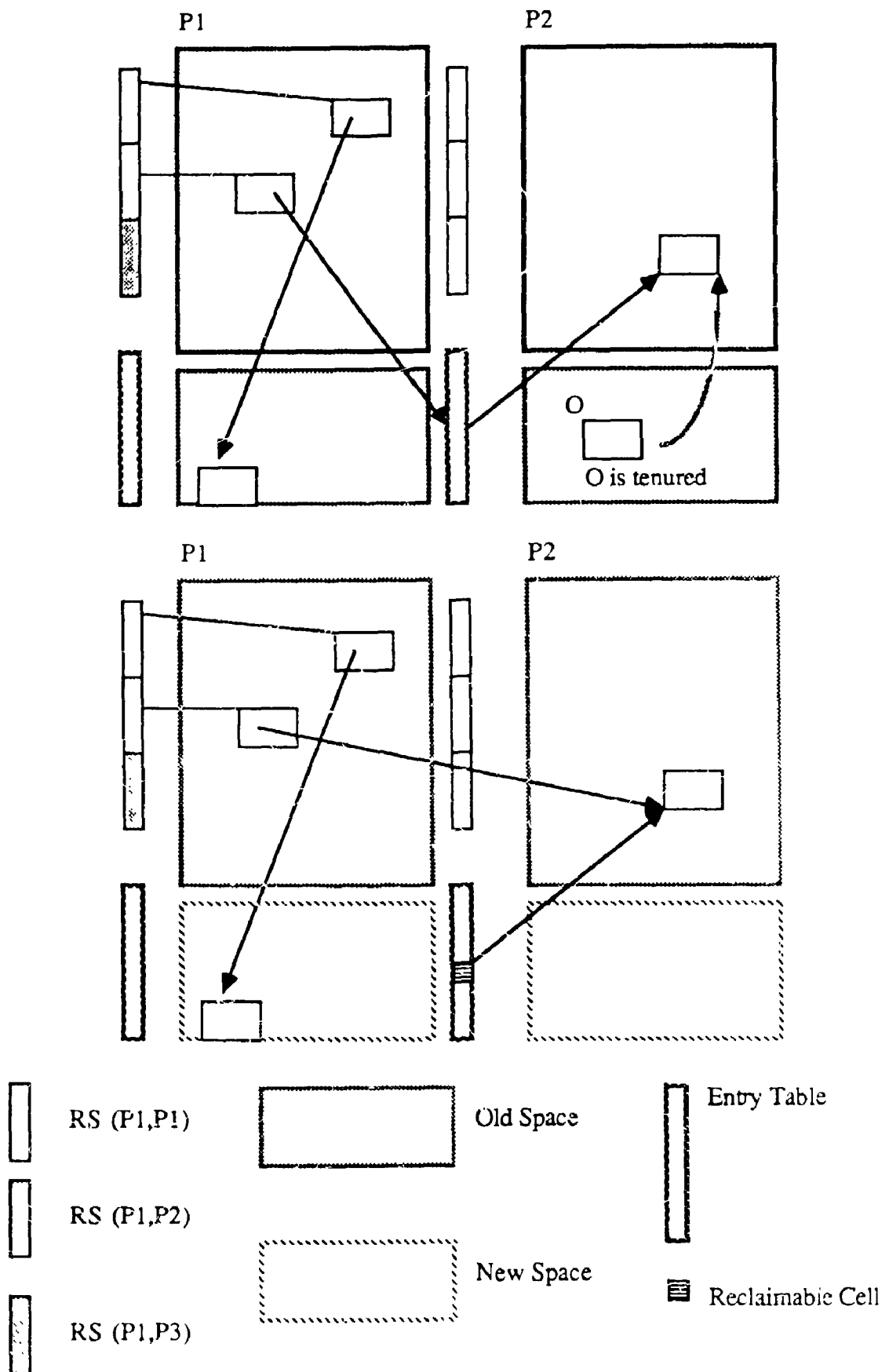


Figure 14: Reclaiming ET Cell To OLD During A Scavenge

```

IF REFERENCED_BY (ET processor_number) AND
  H# _SCAVENGED (processor_number) AND
  NOT_MARKED_BY (ET, processor_number)
THEN
  CLEAR_REFERENCE_BY (ET, processor_number);
ENDIF

```

Figure 15: Clearing Referenced By Flags of Unreferenced ET Cells.

Referenced_By (RB) The RB flags are stored in a bitfield of size N where N is the number of processors in the system. $RB(ET_i(O), P)$ bit is set to 1 when processor P references $ET_i(O)$. Due to arbitrary object deaths, this field only indicates which processors referenced $ET_i(O)$ at some time in the past.

Marked_By (MB) The MB flags is a bitfield of size N indicating that an ET cell was marked by a processor. $MB(ET_i(O), P)$ is set whenever processor P encounters that $ET_i(O)$ during a scavenge. This field represents the state of reachability from each processor at the time of each processors *last local garbage collect*. This could be considered a 'Delayed Marking Bit'.

The scavenger on processor P uses the Has_Scavenged flags to determine which processors have completed scavenges and thus have marked all reachable ET_P cells.

When the local scavenger on processor P walks the ET, it can determine that $ET_P(O)$ is reachable by the following rule. If a processor X has scavenged, and $RB(ET_P(O), X)$ is set to 1 then two cases are possible.

1. $MB(ET_P(O), X)$ is set. The object was reachable and is still reachable, so $ET_P(O)$ is not garbage.
2. $MB(ET_P(O), X)$ is not set. The object was reachable and is not now

reachable, so processor P can clear $RB(ET_P(O), X)$. (The object is not reachable from X)

It is clear that an ET may be reclaimed when all the Referenced_By flags are clear, thus, when processor P clears the RB bit such that $RB(ET_P(O), i) = 0$ for $1 \leq i \leq N$ then $ET_P(O)$ can be reclaimed. The criteria for clearing RB bits is presented in Figure 15.

An ET cell that has a Marked_By flag set and *not* a Referenced_By flag is an error (a processor cannot mark an object which it does not reference). Also note that if the Marked_By flag of an ET cell is set but the Has_Scavenged flag is not set, then this indicates that a remote processor is in the middle of a scavenge. In this case the Marked_By flag may *not* be cleared by the local scavenger. The Marked_By flag may be cleared *only if* the Has_Scavenged flag had been set by the same processor and the processor is not currently scavenging. We cover both cases with the Has_Scavenged flags because the flags are cleared during a remote processors scavenging. Any scavenger which has updated all references to an ET referencing old space will set the Mark_By flags because the ET is no longer reachable from that processor. Any time that two processors exchange an ET, the Marked_By must be set if the Has_Scavenged bit is set so that the ET lives until the next local scavenge.

4.9 Marked_By and Reached_By Bit Maintenance Rules

The following section describes the invariants which must be maintained so that the scavenger can reclaim ET cells correctly. A later section will use these invariants in correctness arguments.

Invariant 1 *An entry table will exist for a new space object if it is referenced remotely. (New Space objects cannot be referenced otherwise).*

Invariant 1 insures that all objects externally referenced will stay alive by prevention.

Invariant 2 *Referenced_By(P)* of object *O* must be set if the $ET_i(O)$ cell is directly reachable from processor *P*. If not reachable from a processor and *Referenced_By(P)*, then *Scavenger(P)* must clear the bit in the next scavenge.

The definition of *Referenced_By* is intended to insure that the *Referenced By* bit is cleared in finite time after the $ET_i(O)$ object becomes garbage. Note that the bit will be cleared by the *next* scavenge on processor *P*.

Invariant 3 *Marked_By(P)* for object *O* must be set if *Has_Scavenged(P)* is set and *O* is reachable from *P*.

Invariant 3 insures that every scavenge by processor *P* will mark every object reachable from its space. This allows the local processor to determine when an object has become unreachable and thus clear the *Referenced_By(P)* bit of *O*.

These three invariants result in the following rules for returning newly created ET cells from message sends. This is the only way new ET cells can be created.

1. Returning a newly created ET object from a message send must set the RB bit of the processor that is receiving the ET pointer (Invariant 2).
2. Returning a handle $ET_i(O)$ to processor *P* must set the $MB(ET_i(O), P)$ bit to true to maintain invariant 3. Thus, if the *Has_Scavenged* bit is set from the processor receiving the object then the *Marked_By* bit must be set (Invariant 3).

4.10 Cross Processor Object Access

The other case for ET creation occurs when storing and fetching objects from old objects in other processor's spaces. New object storing and fetching operations

are by message send only and thus are covered by the rules presented in the above section.

1. Every cross processor store of a new object must create a new ET cell to satisfy Invariant 1. This case is handled as a side effect of the Remembered Set membership test.
2. When a processor stores an ET cell into an object on another processor it must maintain the reachability bits, thus, when transferring an ET cell (by a store) the storing processor must set the Reached_By bit and the Marked_By bit if the Has_Scavenged bit is set for that processor.
3. Every cross processor fetch which return another processor's new object must create an ET. This case requires a message send to the owner processor to create the ET cell.

4 11 Concurrency During Scavenging

Access to a particular object in new space can be given as soon as the scavenger finishes walking that object. As the scavenger walks memory, its position in new space represents a 'high water mark' of allowable accessing. If the scavenger copies the ET references first, then Read/Write access to the ET objects can be given before the scavenger finishes. Access to the new space can be granted on an incremental basis allowing access up to the scavenge high water mark. Mutator execution on the scavenging processor however must be delayed until the scavenger finishes. Any ET cells created during a scavenge (due to a remote request) must have its shadow entry created with the rest of the Entry Table cells.

4.12 Tightly Coupled Processors

Some applications in multiprocessor systems require that two or more processors work closely together and share many objects. In this case, the costs of message sending for new space access via Entry Tables could be too high. In such situations, it is possible to maintain only one Entry Table for the group processors, requiring that all processors garbage collect together as a unit. This type of processor coupling can be performed dynamically allowing groups of processors to be tied together as a logical unit with one allocated processor number for the group. Processors external to the group must use the message passing protocol for remote access while processors internal to the group can use direct object access. This type of system could allow differing configurations to be created depending on the applications running in the system.

4.13 Correctness Arguments

This section contains the correctness arguments for the Entry Table garbage collector. We will show that the invariants above are necessary and sufficient to insure that ET cells are not prematurely garbage collected. In showing that ET cells are not prematurely garbage collected, it follows that the new space object will not be reclaimed prematurely.

Claim 1 *Objects referenced through Entry Tables are not reclaimed.*

It is clear that locally referenced objects are safe from garbage collection. ET cells provide a local reference protecting externally referenced objects. Thus, as long as an ET cell exists for an object it cannot be reclaimed.

Claim 2 *EntryTable_i will only be reclaimed when not referenced externally.*

By Invariant 1, *ET_i* exists. It is straightforward to note an ET is only reclaimed when its Referenced_By bits are cleared. By Invariant 2, an ET will have

the Referenced_By bits set if it is at all referenced externally. It directly follows that if the Referenced_By bits are clear the ET cell is not referenced externally and can be reclaimed.

4.14 Summary

In this chapter we described an algorithm which uses Entry Tables thus reducing many of the synchronization requirements of other algorithms. Some of the advantages in using Entry Tables include:

- processors can garbage collect independently of the other processors;
- synchronization costs are paid only when shared objects are used;
- accessing local objects requires no synchronization;
- external languages, such as C, can be interfaced with the garbage collector; and
- tightly coupled processors may garbage collect together if a large number of objects are shared.

Chapter Five

5 Actra: The Implementation

This chapter provides a description of the implementation of our Entry Table garbage collector described in the previous chapters. We have divided this chapter into three major sections. The first section describes Harmony and the test bed hardware used in our implementation. Secondly, we present an overview of the implementation of the garbage collector, extensions required to the general uniprocessor generation scavenger and the task structure used in our algorithm. The final section covers the vehicle for testing this garbage collector, our multiprocessor Smalltalk system, Actra.

5.1 The Actra-Harmony System

The following sections describe the Actra hardware and Harmony realtime kernel.

5.1.1 Hardware

Actra is implemented on a VME bus multiprocessor using commercially available MC680XX microprocessors and peripherals. The current hardware consists of 12.5 Mhz MC68020 processor cards each with 1 megabyte of memory [44]. Additional memory is provided over VME bus on memory-only peripheral cards.

The Harmony kernel is ROM based and requires 16K per processor. The host interface is implemented on an IBM AT with a VME bus interface card. The host is used to provide mouse and keyboard IO as well as a file system and graphics screen.

5.2 Harmony

Harmony is a multitasking, multiprocessor operating kernel for real-time control using lightweight tasks. Harmony provides a set of interprocess communication primitives (`_Send`, `_Receive`, `_Reply`), as well a process creation (`_Create`), and deletion (`_Suicide`, `_Destroy`) primitives. The message passing primitives are *blocking* meaning that a message `_Send` between processes blocks the sending process until the receiving process executes a `_Reply`. We present a brief description of each of the major Harmony primitives for reference when the reader is studying Appendix A. For an overview of Harmony see [50].

5.2.1 Task Creation

Task creation in Harmony is performed by the `_Create` function. `_Create` requires that a global task index be passed as a parameter. This index is used to search the `task_templates` of each processor and an instance of the process is created on the processor on which the task template is defined.

5.2.2 Sending and Receiving Messages

Message sending in Harmony is performed by the `_Send` function. `_Send` takes a message and process id and transfers the message buffer to the receiving processor. The message send blocks until the receiving processor performs a `_Reply`. The receiving processor must have executed a `_Receive` to receive the message. The important fact about Harmony messaging is that message sending and receiving are synchronous.

5.3 The Entry Table Garbage Collector

The Entry Table Garbage collector can be subdivided into the following subsections:

- The Entry Table scavenger.
- The Entry Table Manager (ET creation/deletion).
- Interprocessor message sending between tasks.
- Multiprocessor interpreter.

5.4 The Entry Table Scavenger

The entry table scavenger can be divided into two major steps, the scanning of the entry table and the generation scavenger algorithm as in [5]. First, the scanning of the Entry table is performed during which any Entry Table cells which are unreferenced can be reclaimed. The second step is to execute the standard uniprocessor algorithm which has been extended to perform the additional work required when interprocessor references are reached. The ET Scavenger is described as two separate steps because the ET scan can be performed without scavenging. This capability and its uses are described later in this section. We first describe the actual scavenger.

The scavenger begins with the root set which consists of the **Remembered-Set** and the **FixedObjects (Registry)**¹³. Each object in newSpace reachable from this root set is copied to the flip space. The objects in the flip space are then scanned in the scavenge operation. The function `copy_and_update_object` will copy the object to the flip space if necessary and update the pointer to the new position. Objects which qualify for tenure are copied to oldSpace instead of flip space and thus must be scanned as well. This is interleaved with every object

¹³The Registry is a set of well known objects like the Smalltalk system dictionary

```

TO scavenge DO
  FOREACH et IN EntryTable DO
    isReferenced = ET_REFERENCES(et)
    hasScavenged = HAS_SCAVENGED_FLAGS
    no_referencers = NOT MARKED(et) AND
                      isReferenced AND
                      hasScavenged
    IF no_referencers THEN
      freeET(et);
    ELSE
      copy_and_update_object (et->real_pointer)
    ENDIF
  END FOREACH
  FOREACH object IN RootSet DO
    copy_and_update_object (object)
  END FOREACH
  FOREACH object IN flip_space DO
    FOREACH field IN object DO
      copy_and_update_object (field)
    END FOREACH
    FOREACH tenured IN old_space DO
      FOREACH field IN tenured DO
        copy_and_update_object (field)
      END FOREACH
    END FOREACH
  END FOREACH
  FOREACH et IN EntryTable DO
    create_shadow_entry (et)
  END FOREACH

```

Figure 16: The Entry Table Scavenger

processed by the scavenger. The tenuring policy is currently designed to minimize scavenge time by reducing the amount of memory copied per flip (this tenures garbage faster than we would like but makes flips really fast). It is our belief that a better tenuring policy would discriminate between locally-only referenced objects and remotely referenced objects.

The main difference from the uniprocessor scavenger and the new entry table scavenger is that the Entry Table must be scanned. The object referenced by an ET cell is not copied unless it is found that the ET cell is still referenced externally. Thus, the object referenced will become garbage unless referenced locally via some other path in the root set. The determination of reachability is done using the criteria described in Chapter 4.

An interesting feature of the Entry Table scanning step is that it can be performed independently of the scavenger (ie the scavenge need not be called after such a scan). The step executes as if the scavenge were in progress except that the copying of the referenced objects is not performed. This feature allows garbage collection of the Entry Table Memory **without having to scavenge**. This is a powerful feature of our collector because this it can be called every time a remote processor scavenges which will result in more efficient reclamation of ET cells.

The final step must recreate the shadow space because the previous shadow space was overwritten by the destination flip space. The pseudocode for the Entry Table Scavenger is show in Figure 16. See Appendix A for the C source code for the Entry Table scavenger.

5.5 Entry Table Manager: ET Creation and Deletion

The Entry Table management system provides ET creation and deletion primitives for use by the interprocess communication system.

The major function provided for ET creation is **ET_for (anObject)** (Figure

```

To ET_for (anObject) DO
    IF IS_IN_OLD_SPACE (anObject) THEN
        return anObject;
    ENDIF
    IF HAS_ET (anObject) THEN
        return ET_FROM_SHADOW (anObject);
    ENDIF
    et = allocateET ();
    ET_REAL_POINTER (et) = anObject;
    ET_MARKERS (et) = myProcessorId ();
    ET_REFERENCES (et) = myProcessorId ();
    return et;

```

Figure 17: Creating an Entry Table Cell

17). This will create an Entry Table cell if necessary or return an existing cell. The markers and references flags are set to be the local processor's Id. This will protect the ET cell from garbage collection until after the next local flip. Any storing of this ET cell into remote memory must set the references bit for that object.

Entry Table cells are managed as a linked free list. This is acceptable because all Entry Table cells are the same size. Currently, a number of Entry Tables are statically allocated. This does not preclude the use of dynamic allocation for ET cells, the sole requirement being that ET cells dynamically allocated must be located in a non-mobile space. The technique we use is to statically allocate a cache of ET cells, any overflow can be handled with dynamic allocation of any extra cells required at runtime. In this case the ET cells must be stored in the RememberedSet for the local processor. These extra entry table cells can be compacted (moved to the cache) during a global garbage collect.

5.5.1 Scanning the Entry Tables

The current implementation scans Entry Tables linearly. For systems which pre-allocate large numbers of entry tables cells, we have devised a faster scanning mechanism. The unallocated entry table cells are used to describe which Entry Tables are in use. This allows us to scan the ET without having to scan all the unused cells. Unused cells are marked as follows. The first unused cell in every unused group of cells contains a link to the next unused Entry Table cell. The scavenger scanning the table can skip over unused blocks. When recovering unused cells, the scavenger maintains these links. This is currently not implemented but is expected to be added to the system later in its development.

5.5.2 Choosing Unique or Duplicate Cells

A system can decide to use unique cells or multiple cells allowed per object. Unique Entry Table cells, (ie each remote reference will use the same entry table to access that object) allow optimization of some operations locally. For example, identity (`===`) can be determined without a remote message send. Unfortunately, unique cells can take longer to create because the system must first determine whether an entry table cell already exists for the object.

Conversely, allowing many handles per external object reference simplifies the creation of these ET entries, as previously assigned handles need not be found. The disadvantage with this approach is that a large number of entry tables can be created for a particular object. This type of entry table is closely related with reference counts [23] [40]. The number of ETs pointing to an object is it's external reference count and allows a simplified mechanism for reclamation of entry tables cells. When the external reference disappears it is clear that the handle can be reclaimed.

A third alternative is to use semi-unique entry table cells. This system would

use a unique entry table for each pair of processors. The advantage of this system is that it can be determined when an external reference is not required and thus reclaim entry table cells when a processor no longer references the ET.

Multiple Entry Table cells per object (including semi-unique ET cells) can be very useful in a distributed environment. A processor which gives out a non-unique Entry Table cell for each local object can use this information to determine the origin of the ET cell (eg. the processor for which the handle was originally created).

Our entry table system uses unique tables and bit flags to determine when an entry table cell is garbage. The unique tables allow us to optimize some operations locally and the bit encoding is used to determine reachability. The use of the unused scavenge memory allows an efficient implementation of the location of existing ET cells.

5.6 The Multiprocessor Interpreter

The multiprocessor interpreter consists of only some minor changes from the uniprocessor interpreter. The major areas of change include:

- object references (`at:`) will create an ET cell if required;
- ET cell methods are forwarded via the local master Actor; and
- some primitives will fail and are implemented in Smalltalk to allow parameters to cross machine boundaries.

Recall that when storing a new object into an old object, the old object must be placed in the Remembered Set. The additional requirement when performing a remote store is that the storing processor may have to convert the stored object into an ET before storing it. Similarly, remote fetches, (fetches from remote old space objects) may require the local processor to ask the remote processor to create an ET cell if the fetch retrieves a new space object.

Two such functions support these requirements **Cross_Boundary_Store** and **Cross_Boundary_Fetch**. **Cross_Boundary_Store** must be called when storing a new space object into a remote old space object. This is the same criteria as **remember** (when adding to the Remembered Set) and thus is performed as a side effect along with the Remembered Set code. **Cross_Boundary_Fetch** must be called on fetches from another processor's old space and may result in the remote processor creating an Entry Table. In this case, the newly created Entry Table cell is returned with the correct **Referenced_By** bits set in the ET. These bits are set according to the local state of the creation processor **Has_Scavenged** flags.

5.6.1 Interprocessor Message Sending

Interprocessor messaging is encapsulated in class **Actor** which is used to manage a processor's memory access and message forwarding. There is one **Actor** for each processor performing any remote ET access or remote message requests. When a remote message send is required, the local **Actor** sends a perform request to the remote **Actor**. The receiving **Actor** queues the message on its processor. When the work is done, the **Actor** replies to the requester.

5.6.2 Task Structure

Each processor runs the following tasks -- a global garbage collector, a **ReaderWriter**, and a Smalltalk interpreter. The global garbage collector is a parallel mark and sweep in which all processors must synchronize. The global garbage collector runs at the highest priority and thus cannot be disturbed during execution. This type of collector was described in [23]. Additionally, every processor runs **Reader** and **Writer (RW)** tasks for high speed remote processor access. These processes are 'scavenger aware' and allow remote access to objects even when a scavenge is in progress. The **RW** tasks run at a higher priority than the local scavenger while the scavenger and the interpreter each run at the same prior-

ity. Each processor may also run many Smalltalk Actors as created by the user. These tasks must run at the same or lower priority than the Smalltalk interpreter if they require use of Smalltalk objects. The reason for this requirement is that we use process priorities to provide MUTEX synchronization between the tasks.

5.7 Reader and Writer Tasks

Low level, interprocessor access to the new space is performed via a *reader* and a *writer* task. These tasks are mainly used by primitives which require synchronized access to two or more objects in separate new spaces. The reader task is responsible for accessing new space objects for remote clients. Correspondingly, the writer task is used to write into new space objects. Additionally, these tasks are used to perform housekeeping requests such as remote scavenge and GGC requests.

A problem arises when a processor is *not* creating garbage and thus *not* scavenging. This could cause the ET of other processors not to be cleared of old references as well as new objects to be tenured even when not referenced remotely. A solution to this is to provide a mechanism to request a scavenge ('request' means that the request is refusable). This request is serviced by the interprocessor Reader/Writer tasks. These requests for scavenges are refusable but if a scavenge request is refused too often then it is possible that a larger number of new objects will be tenured than necessary. A processor receiving a scavenge request can decide to mark reachable ET cells without performing the copy phase of the scavenge. In this case, the remote processor can be informed that the remotely referenced ET cells have been marked and the unmarked cells can be reclaimed.

5.8 Concurrency During Scavenges

Additional concurrency is possible during scavenges. Access to the new space may be given to processes on an incremental basis. The reader and write tasks may access any object in new space (in the destination space side of the flip) which has been scanned. These processors need only check the scavengers 'high water mark' to determine whether it may access the new space object. Accesses which must be denied are processed after the scavenge is completed. At this point, the scavenger must send a message to the reader/writer task stating that it may complete any queued read or write requests.

5.9 Tape and Glue: 'Putting It All Together'

The final section describes the Smalltalk class support for Actra. Actra is currently implemented as a customized version of Smalltalk VME/V, a VME based version of Smalltalk/V¹⁴ [28]. The implementation of this prototype system, required implementation of many supporting pieces. The full description of these pieces is beyond the scope of this thesis. We present a brief description of each and describe how they fit together in the prototype system.

5.9.1 Actors

Actors are currently implemented as a Smalltalk class with Harmony primitives used to talk between Actors. This implementation was chosen for simplicity as it allowed us to write the Actor code in Smalltalk. Future implementation plans include moving parts of the Actor implementation into the virtual machine to allow more rapid interprocessor message forwarding.

Actor messages are converted into Harmony messages and forwarded to the remote processor. The Smalltalk state of the local Actor is stored into itself for use when locally restarting the Actor. This is very similar to the implementation used

¹⁴Smalltalk/V is a trademark of Digitalk Inc.

in [11]. Any new space parameters are converted into Entry Table objects. The receiver Actor converts any local Entry Table parameters into real pointers and then sends the message. Actors execute a receive-perform-reply loop repeatedly when waiting for a request.

5.9.2 Implementing Entry Table Cells as Forwarders

Entry Table cells are implemented as Smalltalk objects similar to Proxy Objects [43].¹⁶ The major difference in our implementation is `doesNotUnderstand:` is not used to forward messages, instead we have implemented each of the object primitives specially (ie. Either as redirections of message sends or local optimizations). This is in effect a 'by hand' implementation of the forwarding mechanism described in [19]. The advantage of this mechanism is that debugging such a system is simplified. It is intended that this mechanism will be implemented in the virtual machine in future versions of Actra.

¹⁶An interesting note about this approach is that it allows visibility of garbage collection structures from the language.

Chapter Six

6 Conclusions

6.1 Overview

We have presented a multiprocessor garbage collection algorithm based on Entry Tables. The Entry Table algorithm has the following advantages;

- Processors can garbage collect independently allowing time critical tasks on other processors to proceed while other parts of the system garbage collect;
- Local mutator performance is not reduced by extra synchronization requirements;
- Processors can be tightly coupled allowing direct interprocessor access on a logical processor level;
- The tenuring policy allows faster direct shared memory object access; and
- The garbage collector can be interfaced to external languages.

We believe that the above features can be directly attributed to the use of a hybrid system using a combination of direct pointers and tables. The use of previously unused generation scavenging memory allowed an efficient implementation of address translation across memory boundaries. Reducing synchronization requirements allows more efficient garbage collection/mutator interaction because the two processes do not *step on each other's toes*. Multiple processors can be

treated as a single processor domain with a single Entry Table for all of the new spaces in the processors space. This allows us to configure the garbage collection strategy best suited for coupling of processors.

6.2 New Directions, Future Research

Many questions remain to be answered ¹⁶. Some of these questions have answers suggested by the work performed during the course of this thesis. However, the answers are incomplete and thus remain as future work.

- Object migration in multiprocessor OOPS is not a well understood feature.
- Is real time possible in multiprocessor garbage collection?
- Can fault tolerance be built into the garbage collector?
- Is memory usage in a multiprocessor the same as in uniprocessor? Some points to consider are:
 - Do interprocessor referenced objects have the same lifetimes as in uniprocessor systems?
 - Are uniprocessor tenuring policies sufficient for multiprocessor systems?
 - Can we use interprocessor reachability to determine suitability for tenuring?
 - Do we need better multiprocessor memory architectures?

Much is known about memory usage patterns in uniprocessor Smalltalk systems ¹⁷. It remains an open question whether or not a multiprocessor system will show these same types of usage patterns. Different memory usage could require new, more adaptive tenuring policies, less aggressive scavenging of shared spaces (due to the possibility that shared objects may live longer) and better memory architectures for sharing objects more efficiently. Additional complexity is introduced when processor coupling is dynamically changed.

¹⁶This is a truism.

¹⁷In general there are many typical memory usage patterns depending on the application.

6.2.1 Hardware Solutions

It is the dream of many programmers and system designers that their problems will be solved by newer, faster, bigger and better hardware. We are glad to be a member of that group and as such present a small list that represents some of the hardware features we think will assist in not only better multiprocessor garbage collection algorithms but better multiprocessor object-oriented systems.

- MMU support for multiprocessors.
- Programmer controlled tag bits.
- Object level processor such as the REKURSIV [48].

The use of an MMU for garbage collection has been shown to be feasible in the Firefly garbage collector. The possible applications of an MMU to our algorithm are in the areas of address translation and memory region ownership enforcement. An MMU could be used to perform the pointer indirection required to reference an Entry Table cell. The access permission bits associated with object pointers can be used to restrict memory access during synchronized activity as well as to memory regions not owned by the processor.

Microprocessor tag bits are often cited as a desirable feature for efficient implementations of object-oriented languages. The use of such tag bits would allow type information to be stored with every pointer in the system. The garbage collector could use these tag bits to provide assistance in detecting cross processor stores, skipping well known objects (nil)¹⁸ during garbage collections, implementing synchronization, and determining processor ownership of pointers.

Processor architecture features, such as TAG bits, were investigated by the SOAR project at Berkeley [21]. While successful, the SOAR project did not result in a processor running Smalltalk substantially faster than high performance implementations on stock hardware.

¹⁸30 percent of all Array contents are nil in the Actra image.

The above, suggest the need for object level processors. An object level processor works with objects, *not* with bits and bytes. An example of such a processor is the REKURSIV. In this processor memory pointers do not exist, objects are referenced by typed ids, and the concept of address is unknown (eg. referencing (id+1) is impossible). It is unknown if object level processors will provide performance significantly greater than commercial processors.

6.3 The Last Word

The design and implementation of a multiprocessor algorithm is both difficult, frustrating, and time consuming while at the same time being very rewarding. Thankfully, there is always an end, even when there is more to be done, so we stop here.

References

- [1] A. Goldberg and D. Robson, **Smalltalk-80 The language and it's Implementation**. Addison-Wesley, Reading, Mass. 1983
- [2] J. McCarthy et al., *LISP 1.5 Programmers Manual*. MIT Press. Cambridge Mass. 1965
- [3] J.G. Mitchell, W. Maybury, R. Sweet, *MESA Language Manual XEROX Research Centre*. Palo Alto Mar. 1978.
- [4] H.G. Baker, *List Processing in Real-Time on a Serial Computer*. COMM. ACM. Vol. 21, No. 4, pp 280-294, April 1978
- [5] D. Ungar, *Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm*. ACM SIGSOFT-SIGPLAN Practical Programming Environments Conference, Pittsburgh, PA, April 84.
- [6] D.W. Hillis, **The Connection Machine**. MIT Press, Cambridge, Mass. 1985
- [7] E.F. Gehringer, J. Abullarde, M.H. Gulyn, *A Survey of Commercial Parallel Processors* Computer Architecture News. ACM Press. Vol. 16 No 4. 1986.
- [8] R.G. Babb, **Programming Parallel Processors**. Addison-Wesley, Reading, MA, 1987
- [9] J. Halstead, *Multi-Lisp: A Language for Concurrent Sybmbolic Computation*. ACM TOPLAS. October 85, pp 501-538.
- [10] J.S. Miller, *Multi-Scheme: A Parallel Processing System Based on MIT Scheme*. Ph.D Thesis September 87. MIT/LCS/TR-402.
- [11] D.A. Thomas, W.R. Lalonde, J.R. Pugh. *Actra - A Multitasking-Multiprocessing Smalltalk*. Technical Report SCS-TR-92, Carleton University, May 1986.
- [12] E.W Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens, *On-the-fly garbage collection: an exercise on cooperation cacm*, Vol 21, Number 11, Nov. 1978 pp 966-975.
- [13] J. Pallas, and D. Ungar, *Multiprocessor Smalltalk: A Case Study of Multiprocessor-based Programming Environment*. In proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation. SIGPLAN Vol 23 Number 7 July 1988, pp 268-277

- [14] D.G. Bobrow, *Managing Reentrant Structures Using Reference Counts*. *toplas*, Vol 2, Number 3, July, 1980, pp 269-273
- [15] L.P. Deutsch and D.G. Bobrow, *An Efficient, Incremental, Automatic Garbage Collector*. *cacm*, Vol 19, Number 9, September 1976, pp 522-526
- [16] D. P. Friedman and D.S. Wise, *Reference Counting Can Manage the Circular Environments(sic) of Mutual Recursion*, *Information Processing Letters*, 1979, Vol 8, Number 1, pp 41-44
- [17] D.S. Wise, *Morris's Garbage Compaction Algorithm Restores Reference Counts*, *toplas*, Vol 1, Number 1, July 1979, pp115-120
- [18] J.M. Barth, *Shifting Garbage Collection Overhead to Compile Time*, *cacm*, Vol 20, Number 7, July 1977, pp 513-518
- [19] D.A. Thomas, W.R. Lalonde, J.S. Duimovich, *Efficient Support for Object Mutation and Transparent Forwarding*. Technical Report SCS-TR-128, Carleton University, November 1987.
- [20] J. Cohen, *Garbage Collection of Linked Data Structures*. ACM Computing Surveys. Vol. 13, No. 3, pp. 341-367. September 1981.
- [21] D. Ungar. **Evaluation of a High Performance Smalltalk System**. MIT Press. Cambridge, Mass. 1986.
- [22] S. Ballard and S. Shirron, in G. Krasner, Ed. **Smalltalk-80 Bits of History, Words of Advice**. Addison-Wesley, Reading, Mass. September 83. pp 127-150
- [23] Y.K. Jew, *Distributed Garbage Collection*, Masters Thesis August 86, School of Computer Science, Carleton University, Ottawa, Canada
- [24] D.G. Bobrow and W.C. Douglas, *Compact Encodings of List Structure*, *toplas*, Vol 1, Number 2, Oct 1979, pp 266-286
- [25] D.E. Knuth, **The Art of Computer Programming Vol I-III**. Second Edition. Addison Wesley, Reading Mass, 1968.
- [26] A.J. Courtemanche, *Multi-Trash, a Parallel Garbage Collector for Multi-Scheme*. Bachelor's Thesis, Department of EE and CS MIT, January 1986
- [27] D.G. Brobow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel, *CommonLoops, Merging Lisp and Object-Oriented Programming*. OOPSLA '86 Proceedings Portland, Oregon.

- [28] **Smalltalk/V 286 Users Guide**. Digitalk Inc. Los Angeles. 1988
- [29] G. Krasner, Ed. **Smalltalk-80 Bits of History, Words of Advice**. Addison-Wesley, Reading, Mass. September 83.
- [30] J. Potter, **The Massively Parallel Processor**. MIT Press, Cambridge, Mass. 1985
- [31] M. Raynal, **Algorithms for Mutual Exclusion**. MIT Press. 1986 North Oxford Academic Publishers Ltd. 1986. London, GR.
- [32] E. Strauss, **80386 Technical Reference**. Simon and Schuster, NY, NY, 1987.
- [33] Motorola, **MC68020 32-Bit Microprocessor User's Manual** 1985 Prentice-Hall, Englewood Cliffs, N.J. 07632.
- [34] Motorola, **MC68551 User Reference** 1986 Prentice-Hall, Englewood Cliffs, N.J. 07632.
- [35] Akmori Yonezawa, Mario Tokoro, **Object Oriented Concurrent Programming**. MIT Press, Cambridge, Mass. 1987
- [36] P. Rovner, *On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically Checked, Concurrent Language*, XEROX Parc, 1985, CSL-84-7
- [37] A. P. Batson and R. E. Brundage, *Segment Sizes and Lifetimes in ALGOL 60 Programs*, *cacm*, Vol 20, Number 1, Jan 1977, pp 36-44
- [38] D. W. Clark and C. C. Green, *An Empirical Study of List Structure in LISP*, *cacm*, Vol 20, Number 2, Feb 1977, pp 78-87,
- [39] D.W. Clark, *Measurements of Dynamic List Structure Use in Lisp*, *ieeese*, Vol 5, Number 1, Jan 1979, pp 51-59
- [40] H. Lieberman, C. Hewitt. *A Real-Time Garbage Collector Based on the Lifetimes of Objects*, *Comm. ACM* Vol 26, Number 6, June 1983 pp 419-429.
- [41] C. Hewitt. *The Apiary Network Architecture for Knowledgeable Systems* Conference Record of the 1980 Lisp Conference. Stanford University, Aug. 1980.
- [42] A.W. Appell, J.R. Ellis, and K. Li. *Real-Time Concurrent on Stock Multiprocessors*. SIGPLAN 88 Conference on Programming Language Design and Implementation. ' SIGPLAN Vol 23 Number 7 July 88.

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-51159-1

```

Local_Request MSG_SIZE = sizeof (Local_Request),
sender = _Receive (&Local_Request, 0),

DMESSAGE ("\\nRequest from %x", sender),

IF Local_Request MSG_TYPE != REMOTE_EXECUTION_REQUEST THEN
    FAIL,
ENDIF

V_objectAtPut(SELF, ACTOR_SELECTOR,
    Remote_To_Local (Local_Request PARAMETERS [0])),
V_objectAtPut(SELF, ACTOR_ARGUMENTS,
    Remote_To_Local (Local_Request PARAMETERS [1])),
V_objectAtPut(SELF, ACTOR_SENDER, TO_SmallInteger (sender)),

SUCCEED (SELF),
ENDBODY

/* Actor send
 * Send a message from self to the Actor on the remote processor
 * Succeed the primitive with the result returned in the reply
 */
DEFINE_USER_PRIMITIVE(actorPrimitiveSend)
BODY
    SmalltalkRequest Local_Request,
    SmalltalkReply Local_Reply,
    int i,
    uint_32    result, id,

    DMESSAGE ("In actorPrimitiveSend", 0),

    Local_Request MSG_SIZE = sizeof (Local_Request),
    Local_Request MSG_TYPE = REMOTE_EXECUTION_REQUEST,

    Local_Request PARAMETERS [0] = LOCAL_TO_REMOTE (PARAM (1)),

    IF !OBJECT_IS_POINTERS(PARAM (2)) THEN FAIL (1), ENDIF

    Local_Request PARAMETERS [1] = LOCAL_TO_REMOTE_ARRAY(PARAM (2)),

    DMESSAGE ("\\nSending selector %x", Local_Request PARAMETERS [0]),
    DMESSAGE ("\\nand arguments %x", Local_Request PARAMETERS [1]),

    id = V_objectAt (SELF, ACTOR_PROCESSOR_ID),
    AS_long (id),

    DMESSAGE ("\\nSending to process %x", id),

    /* Convert processor id to the process id for the execution
     * Actor serving on that process
     */

    id = processor_ids [id],
    DMESSAGE ("Task id %x", id),

    result = _Send (&Local_Request, &Local_Reply, id),

    DMESSAGE ("Reply received result %x", result),
    IF result THEN
        SUCCEED (Remote_To_Local (Local_Reply PARAMETERS[0])),
    ELSE
        FAIL;
    ENDIF
ENDBODY

```

```

RECORD
    uint_16      MSG_SIZE,
    int_16       MSG_TYPE,
    uint_32      PARAMETERS [MAXIMUM_ACTRA_PARAMETERS],
ENDRECORD SmalltalkRequest,

typedef struct SmalltalkReplyStruct
RECORD
    uint_16      MSG_SIZE,
    uint_32      RESULT,
    uint_32      PARAMETERS [MAXIMUM_ACTRA_PARAMETERS],
ENDRECORD SmalltalkReply,

void VtaskRequest (request, id, count, p1, p2, p3)
uint_32 request,
uint_32 id,
uint_32 count,
uint_32 p1, p2, p3,
BODY
    /* Send a message for a specific function from the Vtask
     * This allows you to start a task on a processor
     * initialize it, do some of the initialization and customize
     * the V task
     */
    SmalltalkRequest Local_Request,
    SmalltalkReply Local_Reply,
    Local_Request MSG_SIZE = sizeof (Local_Request),
    Local_Request MSG_TYPE = request,
    IF count-- THEN Local_Request PARAMETERS [0] = p1, ENDIF
    IF count-- THEN Local_Request PARAMETERS [1] = p2, ENDIF
    IF count-- THEN Local_Request PARAMETERS [2] = p3, ENDIF
    _Send (&Local_Request &Local_Reply, id),
ENDBODY

#define UK          0
#define BAD_REQUEST 1

/* Process V task Requests
 * This is the Reader/Writer task
 * 1 Copy of this task runs on each Smalltalk processor
 * This allows a process to initialize the newly created Vtask
 * The routines called by this task are processor specific
 * For convenience, some extra functionality was deposited on this task
 */

void Vtask ()
BODY
    SmalltalkRequest Local_Request,
    SmalltalkReply Local_Reply,
    int i,
    uint_32 sender, result, command,

    WHILE 1 DO
        Local_Request MSG_SIZE = sizeof (Local_Request),
        sender = _Receive (&Local_Request, 0),
        command = Local_Request MSG_TYPE,
        result = OK, /* Default is no error */
    CASE command OF
        CHOICE initialiseMainV
            initialise_processor_idc(),
            initialise_020_interface (),
            initialise_graphics(),
            initialise_fileio(),
            initialise_memory (),
            break,

```

A Entry Table Source Code

A.1 Entry Table Scavenger

```

/*
 * File ET REF
 * Entry Table External references
 * Contents Multi-Processor Entry Table External references
 * Version 1
 * Revision 0
 * Created 880301
 * Comments
 * This file contains the the scavenger externals of the entry
 * table GC The corresponding globals are in ET DEF The include sequence
 * is GC REF ET REF, ET DEF, GC DEF.
 * Note This file and the ET * files associated with it assume that
 * some IPC primitives These need not be Harmony primitives
 */

#define ENTRY_TABLE_SIZE 1024

/*
 * MAGIC Entry Table Class Hash This number is from the ACTRA image
 */
#define H_ET 82

/*
 * An Entry Table Cell
 */
typedef struct Entry_Table_Cell_Structure
{
    OBJECT_HEADER          /* Generate the object fields */

    Object *Real_Pointer, /* The actual object pointer */
    ubit32 References,    /* Which processors references me */
    ubit32 Markers;      /* Have I Been Marked on Last LGC */
} Entry_Table_Cell, *ET_Pointer,

/*
 * Entry Table Referencing Macros
 */

#define ET_REAL_POINTER(anET) \
    (((Entry_Table_Cell*)(ET_PTR(anET)))->Real_Pointer)
#define ET_REFERENCES(anET) (((Entry_Table_Cell*)(ET_PTR(anET)))->References)
#define ET_MARKERS(anET) (((Entry_Table_Cell*)(ET_PTR(anET)))->Markers)

/*
 * Determine whether the handle is referenced by the processorID
 */

#define ET_REFERENCED_BY(anET,processorID) \
    (ET_REFERENCES(anET) & (1 << processorID))

#define ET_MARKED_BY(anET,processorID) \
    (ET_MARKERS(anET) & (1 << processorID))

#ifdef RANGE_IMPLEMENTATION

/* This implementation uses a specific range of addresses for ET cells and
 * masks off the extra bits to reach the ET Other alternatives are to use
 * a bit in the ET itself This requires a memory reference when testing
 */

```

```

/* ADDRESS MASK */
#define ET_MASK      (0x0FFFFFFF)
/* MAX BITS IN ADDRESS */
#define PID_SHIFT    (28)
#define IS_ET(anObject) \
    (((long)(anObject)) > 0) && (((ubint32)(anObject)) > ET_MASK)

#define ET_PTR(anET) (((long)anET) & ET_MASK)
#define ET_PID_MAP(anET) (((long)anET) & ~ET_MASK) >> PID_SHIFT)

#else

/* This version uses a field in the ET that is guaranteed to be
 * set only for ET cells.
 */
#define IS_ET(anObject) CLASS_HASH_FOR(anObject) == H_ET

#define ET_PTR(anET) anET
#define ET_PID_MAP(anET) OBJECT_OBJECT_HASH(anObject)

#endif

#define HAS_SCAVENGED(pID) (HAS_SCAVENGED_FLAGS & (1 << pID))

extern Entry_Table_Cell Entry_Table [ENTRY_TABLE_SIZE],

#define SHADOW_POINTER(st) SHADOW_ET(st)
#define ET_FROM_SHADOW(o) (*(Object**)SHADOW_POINTER(o))
#define HAS_ET(o) (IS_IN_NEWSPACE(o) && OBJECT_IS_REMEMBERED(o))

extern Object *EI_for(), /* name change to LOCAL_TO_REMOTE */
extern Object *LOCAL_TO_REMOTE(),
extern Object *REMOTE_TO_LOCAL(),

/*
 * File ET DEF
 * Entry Table Global Variables
 * Contents Multi-Processor Entry Table Globals
 * Version 1
 * Revision 0
 * Created: 880301
 * Modified:
 * Comments
 * Note that the global variables are global to each processors
 * scavenger This does not include variables which are shared by all
 * the processors Each processor maintains his own copies of these
 * variables
 */

/* Has_Scavenged flags
 * The address of this flag is passed to all the other processors when
 * the new processor is initialized and stored in REMOTE_HAS_SCAVENGED_FLAGS
 */

ubint32 HAS_SCAVENGED_FLAGS,
ubint32 *REMOTE_HAS_SCAVENGED_FLAGS [32],

/* Each processor has an entry table */
Entry_Table_Cell Entry_Table [ENTRY_TABLE_SIZE],

/*
 * File ET C
 * Contents Multi-Processor Entry Table Garbage Collector
 * Version 1

```

```

* Revision 0
* Created 880301
* Modified:
* Comments
* This file contains the additional scavenging functions needed for the
* ET garbage collector. These additional functions are called from the
* uniprocessor scavenger as well as the standard functions
*/

/*
* Scavenging the ET is the same as all scavenges
* Scan the ET and COPY_AND_FORWARD each of the referenced objects
* IF any of the Handles are not referenced then give them back
*/

scavenge_entry_table (destinationNewSpace)
MemoryArray *destinationNewSpace,
BODY
    register int i,
    Entry_Table_Cell *et,
    ubit32 ref_and_scavenge, no_referencers, markers,

    FOR i = 0, i < ENTRY_TABLE_SIZE, i++ DO
        et = &Entry_Table [i],
        IF ET_REAL_POINTER (et) != R_nil THEN
            COPY_AND_FORWARD (ET_REAL_POINTER(et),
                ET_REAL_POINTER(et),
                destinationNewSpace),
            /* get the processors who have marked this cell */
            markers = ET_MARKERS (et),

            /* IF REFERENCED_BY(et,PID) && HAS_SCAVENGED (PID) */
            ref_and_scavenge = ET_REFERENCES(et) & HAS_SCAVENGED_FLAGS_COPY,

            /* IF not marked and his referencers have scavenged, collect it */
            no_referencers = !markers & ref_and_scavenge,
            IF no_referencers THEN freeET(et), ENDIF
        ENDIF
    ENDFOR
ENDBODY

/* Return the SHADOW ADDRESS where the et for the new object
* can be found
*/
ubit32 *shadow_address_for_object (anObject)
char *anObject,
BODY
    ubit32 *shadowAddress,

    shadowAddress = (ubit32*) (anObject - (char*) (&InactiveNewSpace)),
    return shadowAddress,
ENDBODY

/* Set the shadow address of the object to be an ET */
set_shadow_for_object(et, object)
Entry_Table_Cell *et,
Object *object,
BODY
    ubit32 *p,

    p = shadow_address_for_object (object),
    *p = (ubit32) et,
ENDBODY

/* Walk the entry table and fix the SHADOW POINTERS stored in the ET */

```



```

fix_shadow_pointers ()
BODY
    register int i;
    Entry_Table_Cell *et;

    FOR i = 0,i < ENTRY_TABLE_SIZE, i++ DO
        et = &Entry_Table [i],
        IF EY_REAL_POINTER (et) != E_nil THEN
            set_shadow_for_object (et, EY_REAL_POINTER (et)),
        ENDIF
    ENDFOR
ENDBODY

/*
 * Special Cross Processor store of a NON-EI into an Old Object
 * If the store is out of your memory space and you are storing a new object
 * then you must store an EI cell instead
 */

Cross_Boundary_Store (destinationObject,sourceObject,index)
Object *destinationObject,
Object *sourceObject,
SmallInteger index,
BODY
    IF !IS_IN_MEMORY(destinationObject) && IS_IN_NEW_SPACE(sourceObject) THEN
        V_objectAtPut( destinationObject,EI_for (sourceObject),index),
    ELSE
        V_objectAtPut(destinationObject,sourceObject,index),
    ENDIF
ENDBODY

/*
 * Cross_Processor_Fetch
 * Attempt a fetch from a remote old object and if it is a new space
 * object from another processor then invoke a remote message send
 */
Object *Cross_Boundary_Fetch (sourceObject,index)
Object *sourceObject,
SmallInteger index,
BODY
    Object *result,
    result = (Object*) V_objectAt(sourceObject, index),
    IF IS_SmallInteger (result) THEN return result, ENDIF

    IF !IS_IN_MEMORY(result) && !IS_EI(result) THEN
        result = V_RemoteAt(sourceObject,index, myProcessorId ()),
    ENDIF
    return result,
ENDBODY

static Entry_Table_Cell    *EI_Free,

/* Initialize the free list of EI cells in the processor
 * this is c'led at EI creation time
 */
initializeEI()
BODY
    int i,
    Entry_Table_Cell *et,

    printf ("\nInitializing the EI for "),
    info (),

    /* set all the et cells to link with the next free cell */

```

```

FOR i = 0,1 < ENTRY_TABLE_SIZE, 1++ DO
  et = &Entry_Table [i],
  ET_REAL_POINTER (et) = E_nil,
  et->Markers = i + 1,
ENDFOR
/* The last ET has -1 in the marked field */
et = &Entry_Table [ENTRY_TABLE_SIZE-1],
et->Markers = -1,
/* The first free is set to 0
ET_Free = &Entry_Table [0],
ENDBODY

Entry_Table_Cell *allocateET()
BODY
  int nextFree,
  Entry_Table_Cell *et,

  nextFree = ET_Free->Markers,
  IF nextFree > ENTRY_TABLE_SIZE THEN
    fatal ("Out of entry cells"),
  ENDIF
  et = ET_Free,
  ET_Free = &Entry_Table [nextFree],

  /* Make it look like an object */
  et->sizeInBytes = sizeof(Entry_Table_Cell),
  et->flags = OBJECT_IS_POINTERS_FLAG,
  et->classHash = H_ET,
  et->numberOfNamedInstanceVariables = (ubint16) 3,
  et->gcField = 0xABCD,
  et->objectHash = myProcessorId(),

  return et,
ENDBODY

freeET(et)
Entry_Table_Cell *et,
BODY
  int currentFreeET,

  currentFreeET = ET_Free - &Entry_Table [0],
  et->Markers = currentFreeET,
  ET_Free = et,
ENDBODY

/* Convert any object in new space to ET cells
*/

Object *LOCAL_TO_REMOTE_ARRAY (anArray)
Object *anArray,
BODY
  Object **instance_pointer,
  long i,

  WALK_OBJECT (instance_pointer, anArray, i)
  IF IS_IN_NEW_SPACE (*instance_pointer) THEN
    *instance_pointer = LOCAL_TO_REMOTE (*instance_pointer),
  ENDIF
  EFD_WALK_OBJECT
  return LOCAL_TO_REMOTE (anArray),
ENDBODY

Object *REMOTE_TO_LOCAL_ARRAY (anArray)
Object *anArray,
BODY

```

```

Object **instance_pointer,
long i;

WALK_OBJECT (instance_pointer, anArray, i)
  IF IS_IN_NEW_SPACE (*instance_pointer) THEN
    *instance_pointer = REMOTE_TO_LOCAL (*instance_pointer),
  ENDIF
END_WALK_OBJECT
return REMOTE_TO_LOCAL (anArray),
ENDBODY

/* Return a new ET representing the Object
 * See if it has one already If so, then return the
 * existing ET stored in the shadow location
 */
Object *LOCAL_TO_REMOTE (anObject)
Object *anObject,
BODY
  Entry_Table_Cell *et,

  IF IS_IN_OLD_SPACE (anObject) THEN
    return anObject;
  ENDIF
  IF HAS_ET (anObject) THEN
    return ET_FROM_SHADOW (anObject),
  ELSE
    et = allocateET ();
    ET_REAL_POINTER (et) = anObject,
    ET_MARKERS (et) = myProcessorId (),
    ET_REFERENCES (et) = myProcessorId (),
  ENDIF
  return (Object*) et,
ENDBODY

Object *REMOTE_TO_LOCAL (anObject)
Object *anObject,
BODY
  Entry_Table_Cell *et,
  IF IS_MY_ET (anObject) THEN
    return ET_REAL_POINTER (anObject),
  ELSE
    return anObject,
  ENDIF
ENDBODY

```

A.2 Reader/Writer Task

```

/*
 * Actra Messaging Interface
 *
 * Initialization sequence (Non-Image Loading Processors)
 *
 * 1) Initialize Memory
 * 2) Initialize Registry
 * 3) Patch E_startUp -> initializeHarmony symbol
 * 4) Start Processors helper tasks -> Reader/Writer
 * 5) Start Interpreter
 * - Interpreter goes into a waitForFirstMessage state
 * and then you fly from there
 */

#define MAXIMUM_ACTRA_PARAMETERS 4
typedef struct SmalltalkRequestStruct

```

```

RECORD
    uint_16      MSG_SIZE,
    int_16       MSG_TYPE,
    uint_32      PARAMETERS[MAXIMUM_ACTRA_PARAMETERS],
ENDRECORD SmalltalkRequest,

typedef struct SmalltalkReplyStruct
RECORD
    uint_16      MSG_SIZE,
    uint_32      RESULT,
    uint_32      PARAMETERS [MAXIMUM_ACTRA_PARAMETERS],
ENDRECORD SmalltalkReply,

void VtaskRequest (request, id, count, p1, p2, p3)
uint_32 request,
uint_32 id,
uint_32 count,
uint_32 p1,p2,p3,
BODY
    /* Send a message for a specific function from the Vtask
    * This allows you to start a task on a processor
    * initialize it, do some of the initialization and customize
    * the V task
    */
    SmalltalkRequest Local_Request,
    SmalltalkReply Local_Reply,
    Local_Request MSG_SIZE = sizeof (Local_Request),
    Local_Request MSG_TYPE = request,
    IF count-- THEN Local_Request PARAMETERS [0] = p1, ENDF
    IF count-- THEN Local_Request PARAMETERS [1] = p2, ENDF
    IF count-- THEN Local_Request PARAMETERS [2] = p3, ENDF
    _Send (&Local_Request &Local_Reply, id),
ENDBODY

#define OK      0
#define BAD_REQUEST  1

/* Process V task Requests
* This is the Reader/Writer task
* 1 Copy of this task runs on each Smalltalk processor
* This allows a process to initialize the newly created Vtask
* The routines called by this task are processor specific
* For convenience, some extra functionality was deposited on this task
*/

void Vtask ()
BODY
    SmalltalkRequest Local_Request,
    SmalltalkReply Local_Reply,
    int i,
    uint_32 sender, result, command,

    WHILE 1 DO
        Local_Request MSG_SIZE = sizeof (Local_Request),
        sender = _Receive (&Local_Request,0),
        command = Local_Request MSG_TYPE,
        result = OK, /* Default is no error */
    CASE command OF
        CHOICE initializeMainV
            initialize_processor_ids(),
            initialize_020_interface (),
            initialize_graphics(),
            initialize_fileio(),
            initialize_memory (),
            break,

```

```

CHOICE initialiseSlaveV
    /* Copy the Registry and change the E_startup */
    initialize_processor_ids(),
    initialize_O2O_interface (),
    initialize_graphics(),
    initialize_fileio(),
    initialize_memory (),
    initialize_registry (Local_Request PARAMETERS [0]),
    initializeET (),
    break,
CHOICE startV:
    /* After this point you cannot send this task more messages*/
    /* Will execute the startup of E_startup */
    Local_Reply MSG_SIZE = sizeof (Local_Reply),
    Local_Reply RESULT = result,
    _Reply (&Local_Reply.sender),
    interpreter(),
    break;
CHOICE loadImageV
    load_image (VImageFileName ()),
    initializeET (),
    break,
CHOICE verifyImageV
    verify_space(VOldSpace(), "Image Loaded"),
    break,
CHOICE memoryInfoV
    memory_info (),
    break,
CHOICE taskInfoV
    info (),
    print_processor_ids(),
    break,
CHOICE remoteAtV
    result = LOCAL_TO_REMOTE (
        V_objectAt (
            Local_Request PARAMETERS[0]],
            Local_Request PARAMETERS[1])),

    break,
CHOICE remoteAtPutV
    result = LOCAL_TO_REMOTE (
        V_objectAtPut (
            Local_Request PARAMETERS[0]],
            Local_Request PARAMETERS[1],
            Local_Request PARAMETERS[2])),

    break,
CHOICE setIDV
    info (),
    processor_ids [Local_Request PARAMETERS[0]] =
        Local_Request PARAMETERS[1],
    break,
OTHERWISE
    result = BAD_REQUEST,
    break,
ENDCASE
Local_Reply MSG_SIZE = sizeof (Local_Reply),
Local_Reply RESULT = result,
_Reply (&Local_Reply.sender),
ENDWHILE
ENDBODY

/* Processor translation IDs */
uint_32 processor_ids[32],

void initialize_processor_ids()

```

```

BODY
    int i,
    FOR i=0,1<32,1++ DO
        processor_ids[i] = 0,
    ENDFOR
ENDBODY

/* Initialization task requests
 * The initialized V tasks first go into a infant state
 * in which every task must be told which of the initialization
 * routines they should call The reason for this is so that
 * every processor runs the same code
 * NOTE
 * Some of these requests are uniprocessor versions of the
 * the actual routines You may not send the Vtask interface
 * some of these messages 'after' you have started the main
 * task to verify 'maga is non parallel version that blindly
 * scans all memory
 */

/* The V main processes start at 70 + processorID */

#define V_BASE_TASK_INDEX      70
#define V1_TASK_INDEX         V_TASK_INDEX(1)
#define V2_TASK_INDEX         V_TASK_INDEX(2)
#define V_TASK_INDEX(processorID) (V_BASE_TASK_INDEX+processorID)

/*
 * An ACTOR Object contains
 *   processorId      - contains process ID of self
 *   selector         - selector to execute
 *   arguments        - array of parameters
 *   process          - Smalltalk contexts
 *   sender           - process to reply to
 */

/* Offets into Actors for instance variables

#define ACTOR_PROCESSOR_ID 0
#define ACTOR_SELECTOR     1
#define ACTOR_ARGUMENTS    2
#define ACTOR_PROCESS      3
#define ACTOR_SENDER       4

#define REMOTE_EXECUTION_REQUEST 0

/* File actors.c
 * This file defines the Actor Send,Receive,Reply primitives
 * used by the class Actor
 */

/* Actor Receive
 * Place the Actor in Receive state
 * When a message is accepted, store the parameters into self
 * and succeed the primitive
 */
DEFINE_USER_PRIMITIVE(actorPrimitiveReceive)
BODY
    SmalltalkRequest Local_Request,
    SmalltalkReply Local_Reply,
    wint_32 sender, result, command, id,
    int i,

    DMESSAGE("In actorPrimitiveReceive",0),

```

```

Local_Request MSG_SIZE = sizeof (Local_Request),
sender = _Receive (&Local_Request, 0),

DMESSAGE ("request from %x", sender),

IF Local_Request MSG_TYPE != REMOTE_EXECUTION_REQUEST THEN
    FAIL,
ENDIF

V_objectAtPut(SELF, ACTOR_SELECTOR,
    Remote_To_Local (Local_Request PARAMETERS [0])),
V_objectAtPut(SELF, ACTOR_ARGUMENTS,
    Remote_To_Local (Local_Request PARAMETERS [1])),
V_objectAtPut(SELF, ACTOR_SENDR, TO_SmallInteger (sender)),

SUCCEED (SELF),
ENDBODY

/* Actor send
 * Send a message from self to the Actor on the remote processor
 * Succeed the primitive with the result returned in the reply
 */
DEFINE_USER_PRIMITIVE(actorPrimitiveSend)
BODY
    SmalltalkRequest Local_Request,
    SmalltalkReply Local_Reply,
    int i,
    uint_32 result, id,

    DMESSAGE ("In actorPrimitiveSend", 0),

    Local_Request MSG_SIZE = sizeof (Local_Request),
    Local_Request MSG_TYPE = REMOTE_EXECUTION_REQUEST,

    Local_Request PARAMETERS [0] = LOCAL_TO_REMOTE (PARAM (1)),

    IF !OBJECT_IS_POINTERS(PARAM (2)) THEN FAIL (1), ENDIF

    Local_Request PARAMETERS [1] = LOCAL_TO_REMOTE_ARRAY(PARAM (2)),

    DMESSAGE ("Sending selector %x", Local_Request PARAMETERS [0]),
    DMESSAGE ("Send arguments %x", Local_Request PARAMETERS [1]),

    id = V_objectAt (SELF, ACTOR_PROCESSOR_ID),
    AS_long (id),

    DMESSAGE ("Sending to process %x", id),

    /* Convert processor id to the process id for the execution
     * Actor serving on that process
     */

    id = processor_ids [id],
    DMESSAGE ("Task id %x", id),

    result = _Send (&Local_Request, &Local_Reply, id),

    DMESSAGE ("Reply received result %x", result),
    IF result THEN
        SUCCEED (Remote_To_Local (Local_Reply PARAMETERS[0])),
    ELSE
        FAIL,
    ENDIF
ENDBODY

```

```

/* Actor reply
 * Reply to your sender the result from the local execution Actor
 * Succeed the primitive with the result returned in the reply
 */

DEFINE_USER_PRIMITIVE(actorPrimitiveReply)
BODY
    SmalltalkRequest Local_Request,
    SmalltalkReply Local_Reply,
    int i,
    uint_32 sender, result, id,

    DMESSAGE ("%\\n in actorPrimitiveReply").

    sender = TO_long (V_objectAt (SELF,ACTOR_SENDEr)),

    Local_Reply MSG_SIZE = sizeof (Local_Reply),
    Local_Reply RESULT = LOCAL_TO_REMOTE (PARAM (1)),

    result = _Reply (#Local_Reply,sender),

    /* Succeed if reply OK */
    IF result THEN
        SUCCEED (SELF),
    ELSE
        FAIL,
    ENDDIF
ENDBODY

userPrimitiveTableEntry UserPrimitiveTable[] = {
    "actorPrimitiveReceive",    actorPrimitiveReceive,
    "actorPrimitiveSend",      actorPrimitiveSend,
    "actorPrimitiveReply",     actorPrimitiveReply,
    0,0
},

```

A.3 Examples From Class Actor

```

Object subclass #Actor
    instanceVariableNames
        'processorId selector parameters process sender '
    classVariableNames ''
    poolDictionaries '' !

!Actor class methods !
run
    " start the actor on its processor "
    Actor new initialize waitForRequest! !

!Actor methods !

id anInteger
    " set the processor id of the actor to be anInteger "
    processorId = anInteger!

initialize
    " MAGIC CODE HERE "
    " Initialize an actor for processor 8 (Harmony Processor Number)
    and set him waiting "
    processorId = 8
    selector = #waitForRequest
    parameters = #()

```



```

~ self!
primitiveReceive
  " Receive a request from any processor, the selector and parameters
  will be passed to you Any interprocessor conversion will be performed
  by the primitive "
  <primitive actorPrimitiveReceive>
  ~ self primitiveFailed!
primitiveReply result
  " Perform a reply to the sender processor. Result conversion will be
  performed by the primitive "
  <primitive actorPrimitiveReply>
  ~ self primitiveFailed!

primitiveSend message withArguments anArray
  " send the message and arguments to the processor represented by self "
  <primitive actorPrimitiveSend>
  ~ self primitiveFailed!

waitForRequest
  " loop forever receiving requests, performing the request and reply with
  the answer "
  | result |

  [ true ] whileTrue [
    self primitiveReceive
    result = self perform selector withArguments parameters
    self primitiveReply result]! !

```

A.4 Example From Class Entry Table

```

Object subclass #EntryTable
  instanceVariableNames
    'realObject references markers '
  classVariableNames 'LocalActor'
  poolDictionaries '' !

'EntryTable class methods !
initialize
LocalActor =>Actor new initialize ! !

'EntryTable methods !

= anObject
  "Local optimization example"
  ~ anObject = realObject!

at anInteger
  "Forward message example Ask the local actor to forward message"
  | m |
  m = Message new
  m selector #at
  m arguments (Array with self with anInteger)
  ~ LocalActor send m to self processorID for self!

```

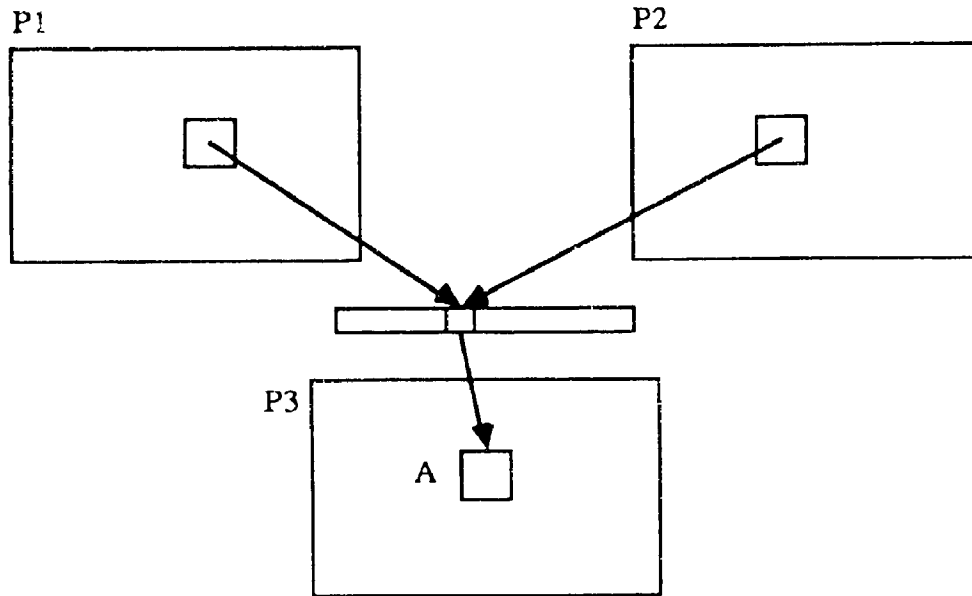


Figure 18: Entry Table Recovery Example: Initial State

B Entry Table Recovery Example

B.1 Example: Reclaiming an Entry Table cell

The following is an example of the ET cell reclamation algorithm.

We have three processors P1, P2, P3. An entry table cell $ET_{p_3}(A)$ exists on P3 pointing for object A which is in $MNew_{p_3}$. P1 and P2 contain references to A. The initial configuration is shown in Figure 18.

Note: RB = Referenced_By, MB = Marked_By, and HS = Has_Scavenged.

The initial state is:

Flags	Processor		
	1	2	3
RB	1	1	0
MB	0	0	0
HS	0	0	0

The following steps occur.

1. P2 scavenges.

RB	1	1	0
MB	0	1	0
HS	0	1	0

A is reachable from P2 and so P2 sets MB and HS bits on A.

2. P3 scavenges.

RB	1 1 0
MB	0 0 0
HS	0 0 0

P3 notices that P2 has scavenged but that MB bit is set on A so nothing is done to the RB flags. The ET cell is not collectable since is still referenced. After P3 finishes it's scavenge the bits are set as follows.

3. P1, P2 lose reference to A. This has no effect on any of the flags of A.

4. P3 scavenges.

RB	1 1 0
MB	0 0 0
HS	0 0 0

Neither P1 nor P2 has scavenged so P3 cannot make reachability determinations.

5. P2 scavenges.

RB	1 1 0
MB	0 0 0
HS	0 1 0

Note that the HS and MB bits show that A is not reachable from P2.

6. P3 scavenges.

RB	1 0 0
MB	0 0 0
HS	0 0 0

P3 determines that A is not reachable from P2 and clears the RB for P2.

7. P1 scavenges.

RB	1 0 0
MB	0 0 0
HS	1 0 0

The HS and MB show that A is not reachable from P1.

8. P3 scavenges

RB	0 0 0
MB	0 0 0
HS	1 0 0

After this scavenge, P3 may reclaim the ET cell for object A since it is not reachable from any other processor. Object A is now free to be reclaimed if it is not locally referenced.

END

03-03-90

FIN