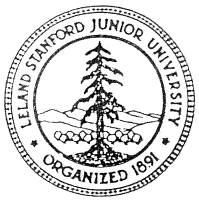
COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY · STANFORD, CA 94305-2192



Improving Garbage Collector Performance in Virtual Memory

Robert A. Shaw

Technical Report: CSL-TR-87-323

March 1987

This research was supported by, and is reproduced with the permission of, the Hewlett–Packard Company.

Improving Garbage Collector Performance in Virtual Memory

by
Robert A. Shaw
Technical Report CSL-TR-87-323
March 1987

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

Garbage collection and virtual memory have long been in an adversarial relationship. Implementers of virtual memory systems have cited garbage collectors as an excellent test case for undermining page replacement algorithms [BabJoy81] and users of Lisp systems have turned off garbage collection (and suffered the consequences) rather than live with the slowdown of garbage collector paging [Moon84].

This paper describes a way in which the garbage collector and virtual memory system can work together to improve overall system performance. By using a simple layout for storage and information already maintained by most virtual memory systems, a garbage collector can substantially reduce the amount of effort necessary to reclaim a large majority of the available space. The techniques presented require no special hardware and minimal disruption of the runtime environment.

Key Words and Phrases: Garbage Collection, Virtual Memory, Lisp Measurement, Data Lifetime, Generation, Dynamic Storage, Allocation, Reclamation.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Introduction

Garbage collection and virtual memory have long been mortal enemies. Virtual memory (VM) provides the user freedom from concerns about the size limitations of physical memories. Automatic storage reclamation, or garbage collection, provides the users of many systems with freedom from storage management. Unfortunately, these two worksaving conveniences interact terribly. Though they both attempt to free the user of unwanted work, they do so by adding some expense to the computation. Virtual memory minimizes its cost by exploiting a general observation about program execution: instruction and data references tend to be localized to reasonably small areas of the address space [Dennin70]. Garbage collection saves the user effort by searching through every data object in the system to locate those which are no longer in use. Garbage collection violates the foundation upon which virtual memory depends. Virtual memory presumes locality of reference, while garbage collection attempts to reference everything which can be referenced in as short a time period as possible. The user pays for this by enduring long interruptions that can often be spent doing little more than moving memory contents back and forth between physical memory and backing store, a process known as *thrashing*.

The usually stated goal of garbage collection in a system is to recover reusable address space that is no longer needed. As new architectures, operating systems, and virtual memory systems provide larger and larger virtual address spaces, this goal decreases in importance. Indeed, if there is such bad interaction and address space is not such a valuable resource, why bother with garbage collection at all in large virtual memory systems?

There are two obvious reasons. Space used in virtual memory is not free: if data is not in memory, it resides in some form of backing store. Although backing store is relatively cheap, new architectures are making address space extremely cheap. In other words, address space may be free, but the physical storage to back it up is not. If the majority of what is allocated could be reused, then any real or backing store holding this useless information is truly wasted. For a system with realistically limited resources, this may be a serious problem. The second reason involves performance. Both [FenYoc69] and [White80] argue that due to the enormous size possible in virtual memory systems, it is no longer proper to simply view a garbage collector as a means of reclaiming address space; the purpose of a garbage collector in a large virtual address space is to improve locality of reference so that virtual memory performance might be improved through increased data density. This view is consistent with recent experience. Several users of the Lisp system on which the author's experimentation is being done have begun using extremely large heaps in order to avoid lengthy garbage collection interruptions. What has been found is that, as time progresses, the system becomes "sluggish." The apparent cause of this sluggishness is the increased virtual memory traffic in the system. Because data is spread throughout the heap and separated by vast amounts of garbage, many more pages must be brought into the physical memory to satisfy the needs of the user computation. A compacting garbage collector takes a long time to run, but drastically improves the density of the data and thus reduces the sluggishness.

The work described in this paper was motivated by an attempt to achieve a dramatic improvement in garbage collector performance for an existing Common Lisp implementation running on multiple types of conventional architectures. Major overhauls of the Lisp system or the addition of custom hardware were not viewed as viable options, so simpler solutions were sought. The remainder of this paper will explore classic garbage collection techniques, recent improvements to them, and a new approach to collection that is designed to improve its performance when run on conventional hardware.

2. Terminology

Terminology has always been an important part of the garbage collection literature. Terms such as *transporter*, *broken heart* and *remembered set* are commonplace. There will be an attempt to limit terminology even though the temptation to create new names remains very strong.

Since the organization, allocation, reclamation, and compaction of data memory is the focus of this paper, a few definitions are necessary before proceeding. *Data memory* is the area in which essentially all *data* or *data objects* reside. Data objects may be stored in the execution stack(s) and in processor registers as well as in data memory. Data objects are the structures containing the actual information to be processed by the program. Examples of data objects are Lisp CONS cells, vectors, or bignums.

References identify data objects and provide a uniform and efficient way to handle them. Efficiency can be gained by passing around constant-sized references rather than arbitrary-sized data objects.

Data memory is broken into a dynamic area and a static area. The dynamic area will be called the *heap*. For the purposes of this paper, a heap will always be considered to be a reasonably large contiguous area of memory. This is not always the case in practical systems, but helps simplify the explanations. Storage in the heap is allocated for new data objects as the user program creates them.

The static area of data memory will be called the *base set*. In a Lisp system the base set consists of data objects referred to by code or objects such as symbols which must always be available. Data objects and references contained in the stack(s) and registers are usually also considered part of the base set. A data object is considered *live* as long as it can be reached through some chain of references originating in the base set. The storage associated with live data objects must not be reclaimed. Only data objects in the heap are subject to reclamation; data objects in the base set can never die.

An additional term, the *minimal base set* will be used to denote only the set of references from the base set into the heap. References which do not point into the heap, and self-contained data in the base set, are not considered part of the minimal base set. Duplicate references from the base set into the heap do constitute separate entries in the minimal base set.

3. Classic Garbage Collection Schemes

All basic garbage collection schemes consist of two steps:

- identify live (or dead) data objects, and
- recover the storage associated with the dead data objects.

Different collection schemes use varying approaches to accomplish these steps. Three of the most popular schemes are described below.

3.1 Reference Counting

Reference counting systems attempt to identify the point at which the data object is no longer live by noting when the last reference to the data object is lost. This is done by incrementing a counter, the *reference count*, associated with each data object every time a reference to it is created and decrementing the counter every time a reference is lost. References are created and lost through assignment or the creation and destruction of environments. When the reference count reaches zero, the data object is no longer referenced and its storage may be reclaimed.

Reference counting has the desirable feature that garbage collection is accomplished at the earliest possible moment. The reclamation is spread out through time rather than being lumped into a single long and potentially disruptive operation. Deutsch and Bobrow have implemented effective methods for improving performance of reference counting systems by avoiding frequent cases of incrementing and decrementing [DeuBob76]. Unfortunately there are still two fundamental problems with this garbage collection scheme. First, reference counting cannot reclaim either dead data objects linked into a circular structure or data objects whose reference counts have overflowed. Lieberman and Hewitt have commented that styles of programming that utilize circular structure are becoming more prevalent [LieHew83], and [Rovner85] stated that circular structure in the language Cedar (which used reference counting) caused more trouble in garbage collection than had been expected. The second fundamental problem with reference counting is, as Ungar has pointed out, the effort is proportional to the number of dead data objects [Ungar86]. This may not sound like a substantial problem, but most of the significant improvements recently made in garbage collection are premised on heuristics about the ratio of live to dead objects in areas of recently created data objects. The importance of these heuristics will be discussed in more detail later.

3.2 Mark and Sweep

One-space or mark and sweep collectors allocate storage from a free list (i.e. a list of free storage in the heap) until some minimum threshold of available storage is reached. Upon reaching the threshold the user computation is stopped and garbage collection begins (Figure 1a). The base set is traversed to find all data objects recursively reachable. When an object in the heap is encountered for the first time, it is marked. Then all objects in the heap which are recursively reachable from the object are traversed and marked, if not already marked. Marking usually involves setting a bit associated with the data object to indicate that it is live. Figure 1b shows the state of the mark bits after the bottom element of the base set and those data objects recursively reachable from it have been traversed and marked. Should a marked object be encountered later through some other chain of references, the fact that it is marked will prevent further effort from being expended in traversing the object and those reachable from it. When all reachable objects have been marked, the mark phase ends and the sweep phase begins. See Figure 1c. At this point it is known that all live objects in the heap have been marked; any unmarked object is known to be dead. The sweep phase recovers the storage used by dead data objects by linearly sweeping the heap looking for unmarked data objects. When a live object is encountered, the mark associated with it is cleared. When a dead object is encountered, the storage associated with it is placed on a free list. At the end of the sweep the garbage collection is complete and the user computation may be resumed (Figure 1d).

The effort spent is a function of the size of the base set (which must be fully traversed), the number of live references in the data memory (used during mark phase), the number of live objects in the heap (marked once during mark phase and mark cleared during the sweep phase) and the number of dead objects in the heap (added to free list during sweep phase). It is possible to change the term involving the number of dead heap objects to a term proportional to the size of the heap and number of contiguous chunks occupied by dead objects. This can prove to be worthwhile in systems where mark bits are not part of the data object in memory but are held somewhere else in a dense table. When an unmarked object is encountered in the table it is only necessary to search the table for the next marked object to determine the size of the region to be added to the free list; actually looking through dead objects in memory can be avoided.

An additional optimization of the mark and sweep scheme is compaction. Compaction reduces allocation problems associated with fragmentation of the memory. Remember, space from dead

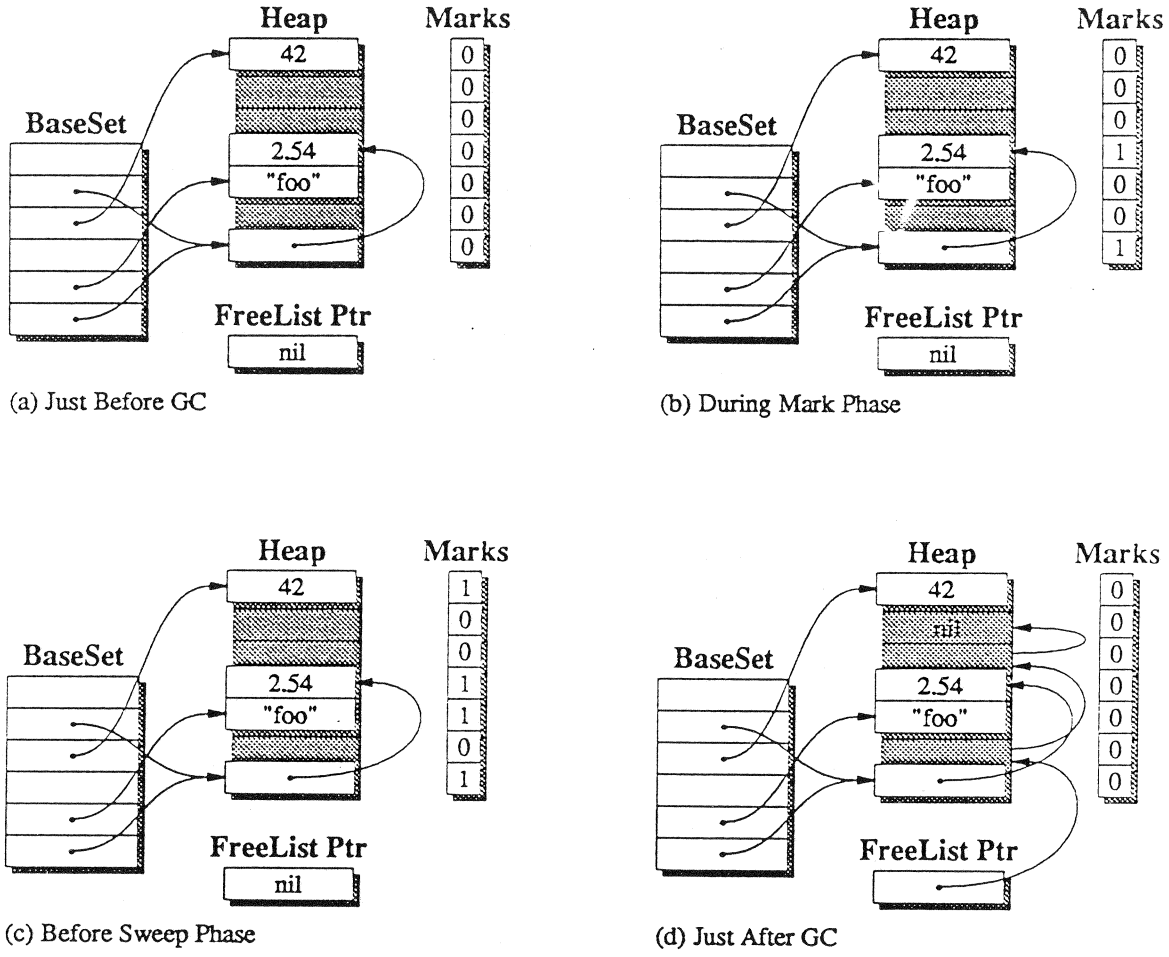


Figure 1. Mark and Sweep Garbage Collection

objects has merely been added to a free list, it has not been combined into a single large space. The heap is probably filled with holes. Compaction is an expensive operation, theoretically requiring the movement of at least enough data objects to fill all the gaps, and in most realistic systems, requiring the movement of every data object appearing after the first gap in the heap. Additionally, a relocation map must be built to record what was moved where, and a linear pass over all live objects must be made to adjust any references (presuming references do not use indirection through a table in which case only the table need be updated).

The benefits of a mark and sweep collector are that no dead structure can survive a garbage collection, data need not be moved (which may be important in some systems), and that the entire space allocated to the heap can be used for data. A negative aspect of mark and sweep collectors is that memory fragmentation can become a serious problem unless compaction or elaborate multi-space allocation is done. Also, compaction must be an atomic operation since the data memory is inconsistent during that time (barring complex schemes which will degrade performance) and, because of the multiple garbage collection phases, accesses of the same location are guaranteed to occur at substantially disjoint points in time. This can have substantial bearing on the virtual memory performance.

3.3 Stop and Copy

The third and final collection scheme to be described is the two-space stop and copy collector. Stop and copy works by copying live data objects from the space being collected into an unused space. In this scheme the heap is divided into two equal-sized spaces (Figure 2).

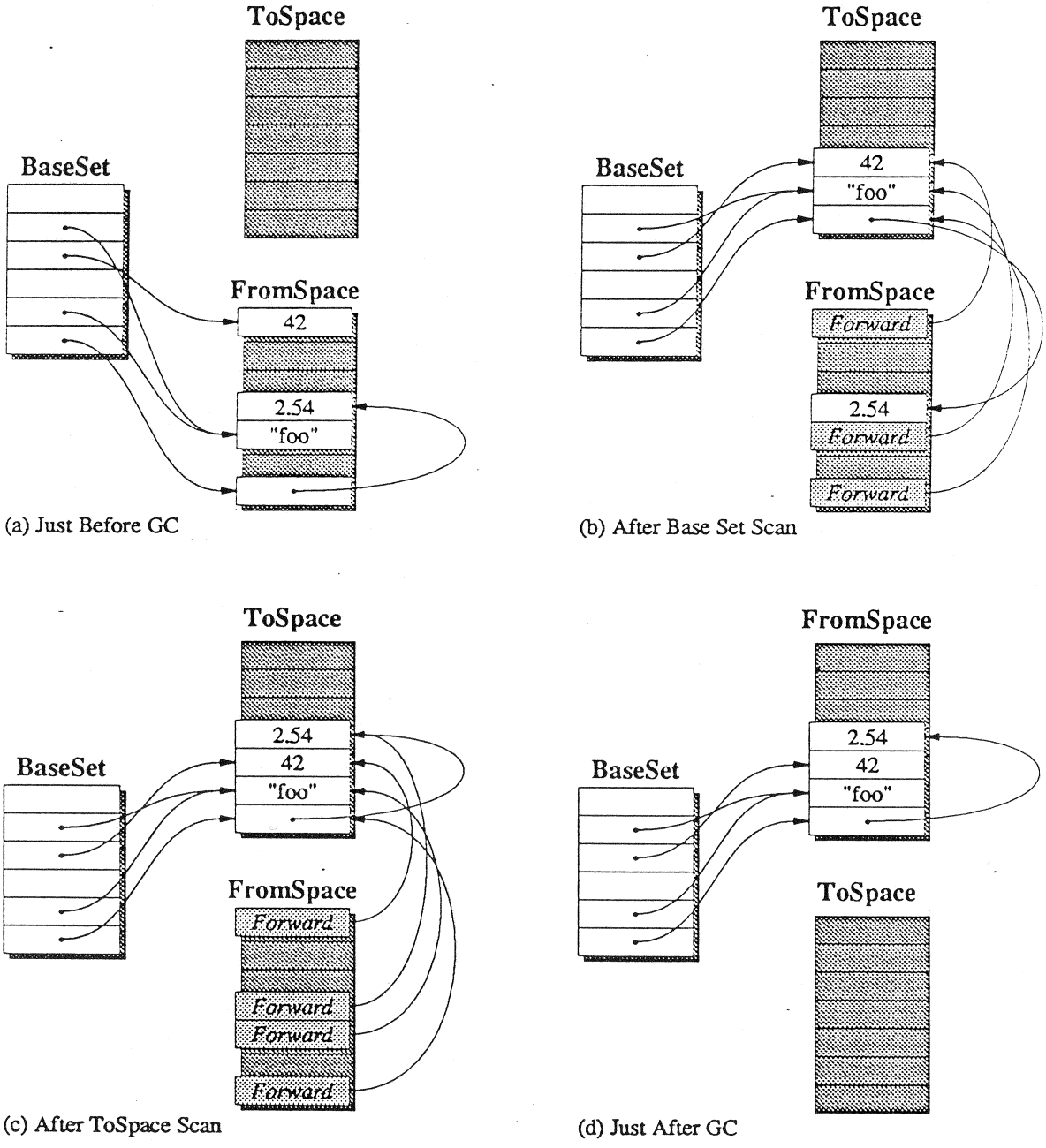


Figure 2. Stop and Copy Garbage Collection

All allocation takes place from one of the spaces (called **FromSpace**). The second space remains empty until garbage collection begins. We will call the empty space **ToSpace**. When there is no more room in **FromSpace** the user computation is stopped and garbage collection begins (Figure

2a). The base set is traversed looking for references into **FromSpace**. When one is found a check is made to see if there is a *forwarding pointer* where the data object should be. A forwarding pointer is a pointer used strictly by the collector to indicate the new location of a moved data object. If there is a forwarding pointer, the original reference is updated to point where specified by the forwarding pointer and the base set traversal is continued. If the data object is in **FromSpace**, it is copied to the next available location in **ToSpace**, a forwarding pointer is placed at the old location in **FromSpace** to indicate where the data object was copied to, and the original reference is updated to reference the moved data object (Figure 2b). When the base set has been fully traversed, **ToSpace** can be scanned linearly looking for references from copied objects into **FromSpace**. As references are found into **FromSpace**, the data objects are copied to **ToSpace** (if they have not already been copied) and the reference is updated to reflect the new location. Since **ToSpace** is scanned from the same end that allocation first occurred and the scan is in the direction of new allocation, data objects copied during the scan will always be placed at the end of **ToSpace** where scanning has yet to happen. When the scanning pointer reaches the new allocation pointer, the garbage collection is complete (Figure 2c). There should no longer be any references into **FromSpace**; all that remain are dead objects and forwarding pointers. The uses of **FromSpace** and **ToSpace** are interchanged and the user computation can continue. The interchange of the spaces reclaims all space in **FromSpace** for use as **ToSpace** during the next collection (Figure 2d).

The algorithm described does a breadth first traversal of the objects referenced by the base set [Cheney70]. A breadth first search can be quite detrimental to virtual memory performance since data objects which are logically close may be made physically quite distant. Optimizations such as list linearization [ClaGre77] or pseudo-depth first search [Moon84] can help alleviate problems introduced by using this scheme.

The stop and copy garbage collector has expense proportional only to the size of the base set (which must be fully traversed), the total number and size of live objects in the heap (which must be copied), and the number of references into **FromSpace** (which must be updated). This seems very attractive since there is no dependence on the number or size of the dead objects. Unlike the mark and sweep collector, however, every live data object in the heap must be moved; a data object is live only if it is copied from **FromSpace** to **ToSpace**. If there is a large amount of relatively stable structure, unless special steps are taken, the structure must be copied back and forth during successive garbage collections. Neither reference counting nor the mark and sweep collector has this problem. Even in the compacting variations of the mark and sweep scheme, relatively stable structure will have a tendency to settle and remain in an area.

For systems which cannot tolerate movement of the data (e.g. those that pass non-updateable pointers out of the system), stop and copy may be unacceptable. An additional aspect of the stop and copy scheme that causes concern among implementers is the issue of memory utilization. At best, the classical version of the algorithm described permits only half of the available dynamic data memory to be allocated before a garbage collection must take place; the second half of the memory remains idle and empty, waiting for copying to occur. In address space constrained implementations this may make the scheme inappropriate. A VM expense that might be excessive in the stop and copy scheme is that of having to actually access the heap when a reference to **FromSpace** is encountered. The least predictable accessing in garbage collectors is that involved with marking or checking to see if a data object needs to be copied. If there were only one reference in existence for each data object, the same number of heap references would be required for either scheme. But there is often more than one reference to a data object. In the mark and sweep scheme a small table of mark bits can be used to avoid actually making accesses to a large heap for any but the first reference encountered. The stop and copy scheme essentially

requires making these fairly random heap accesses to obtain the forwarding information.

The stop and copy scheme does have several advantages. Its implementation is quite simple. There are no tricky tables to build and no elaborate techniques necessary to achieve recursion without using large amounts of space. In the simplest case, in which the base set is a single contiguous block, a linear scan of it followed by a linear scan of **ToSpace** will accomplish the entire traversal of all live objects. A nice feature of this scheme in virtual memory is that each reference is touched only once. While a compacting mark and sweep collector needs to access each reference once to initiate marking and once to relocate the reference after compaction has occurred, the stop and copy scheme combines these phases into a single pass, potentially reducing virtual memory paging by a factor of two. Another aspect of the stop and copy scheme which has added to its attractiveness is its ability to be converted into an incremental algorithm [Baker78]. Since the heap can be guaranteed to be consistent by making the runtime system cognizant of forwarding pointers and by making the copying of data objects and storing of forwarding pointers atomic, garbage collection can be distributed over long periods.

4. Making Measurements

Much of the design and discussion which follows is based upon analysis of real programs running on a commercially available Common Lisp system. All programs have been run on a Hewlett-Packard Series 9000/350 computer under HP Common Lisp. The HP350 is a single user workstation based on a 25MHz Motorola 68020 processor and 68881 floating point coprocessor. The virtual memory system uses 4096 bytes per page. The workstation executes native instructions at an approximate rate of 4 MIPS. HP Common Lisp is a full implementation of Common Lisp as described in [Steele84]. The implementation is written primarily in Lisp. Lisp references are encoded in four bytes.

The programs analyzed were chosen to be representative of some typical uses of Lisp. These programs are not fabricated benchmarks, nor are they hand optimized for the purpose of benchmarking. All the normally present "missed opportunities" for optimization that appear in medium sized systems exist in this code. There is even evidence that a function or two in some of the tests is running interpreted rather than compiled. Execution of these tests represents well over one billion instructions. The tests were chosen from programs which are either in daily usage or wide distribution. The programs, along with the input used, are:

Compile	a Common Lisp compiler compiling itself,
NL	a natural language system computing unambiguous parses of 44 sentences of varying complexity,
Reduce	a symbolic math package executing a standard test file, and
RSIM	a transistor level circuit simulator simulating a ten bit counter counting to ten.

Tools used in the analysis took three forms. All data pertaining to dynamic writes were gathered using a meta-circular 68020 emulator running in the Common Lisp system. This emulator is heavily instrumented and quite knowledgable about the Lisp implementation. It allows measurement of both the application and Lisp system while not gathering information about either the garbage collector or underlying operating system. All garbage collector specific data were obtained through instrumentation of the existing mark and sweep collector and a new stop and copy collector. Allocation, timing and static data were gathered using specially designed tools.

5. Reducing Garbage Collection Effort

Most people (all, in the author's experience) would agree that garbage collection using the classical schemes described takes too long. Interruptions due to non-reference counted garbage collection are viewed as unwelcomed distractions that impede progress and destroy trains of thought. Reducing the interruptions is a laudable goal.

Reference counting was included above only for completeness. For the remainder of this paper reference counting collectors will not be considered. This is done for several reasons. Reference counting is not a full storage reclamation scheme; a second, different, scheme must be used to recover even high mortality rate data which is circular or has overflowed counters. Reference counting is an intimate part of the systems in which it is used and much work has been done finding ways to optimize its performance. Finally, reference counting is sufficiently different in spirit from the mark and sweep and the stop and copy collectors that ideas which can easily be used for either mark and sweep or stop and copy cannot be used at all for reference counting. In removing reference counting from the following discussion there is no intent to diminish its viability as a very appropriate scheme for systems with specific requirements.

The remainder of this paper will focus on compacting mark and sweep schemes and especially on stop and copy schemes. These schemes are in some ways outsiders to the systems in which they operate. They are effectively a second process which is scheduled to run when the memory resource is depleted. Only non-incremental versions of garbage collectors will be considered. It is felt that incremental collectors attempt to hide the effort rather than reduce it. Some of the techniques which will be described can be applied to incremental collectors, but discussing the application will probably do little more than add complexity to schemes that might be adequate without becoming incremental. Only mark and sweep and stop and copy collectors operating in systems where the computation has been stopped for the duration of the collection will be considered.

5.1 The Heuristics (Lieberman and Hewitt)

What can be done to reduce the time spent in garbage collection? In virtual memory systems attempts to compress key information into small tables and make memory accesses more orderly appears to be a good place to start. Allocating objects in the base set such that they are physically contiguous (i.e. they may be linearly rather than randomly scanned) and using compact tables of mark bits rather than mark bits spread through memory are examples of ways to improve VM performance. Modifications such as these can result in substantial improvements, but do not address a key problem. Garbage collectors historically must deal with (at least) every live object in the system during a collection. This includes all the data structures used by those programs which were created but rarely if ever used. Requiring the collector to access and possibly move all this relatively stable structure takes time that would be better spent on the user computation. In one of the worst cases it is possible that the entire user computation must be moved out of physical memory to make room for the relatively stable structure which must be traversed during garbage collection.

It is easy to believe that garbage collection in a system with a small base set and heap is faster than garbage collection in a system with a large base set and heap. There is simply less to be done. If the total space occupied by the base set and the heap is less than the physical memory available to the process, it might be possible to do the entire garbage collection without any VM paging. This has potential for enormous gains in performance. Unfortunately, reducing the size of data memory available in the system is not an issue most users wish to discuss (remember that reduced data space is considered to be a disadvantage of the stop and copy approach). The trick,

then, is to make the base set and data memory appear smaller to the garbage collector, but not to the user.

Recent work, beginning with [LieHew83] and continuing with [BalShi83], [Moon84], and [Ungar86], has suggested that effort spent in garbage collection can be reduced by making use of two simple heuristics which depend on knowledge of the age of an object:

- The mortality rate among newly created objects is quite high and is substantially higher than that of older objects (reduce collected area of heap).
- References contained in an object tend to identify previously created objects (reduce base set).

The first heuristic leads to a reduction in effort by reducing the size of the heap to be collected. The reduction results from noting that newly allocated areas tend to yield the highest ratio of dead to live objects. By focusing the effort on regions of newly allocated objects where the mortality rate is expected to be high, the size of the heap is effectively lessened. The older portion of the heap can be temporarily absorbed into the base set. Only dead data objects in the newer part of the heap are reclaimed, dead objects that are contained in the older part of the heap are not reclaimed and may artificially prolong the life of some objects in the newer part.

But why should pushing objects out of the heap into the base set help performance? All that has been done is to reduce the collectable heap size by enlarging the size of the base set. Indeed, why isn't performance worse since some dead data objects have been resurrected and added to the base set? Informal tests were run to determine how much, if any, improvement could be expected. Both a compacting mark and sweep and a stop and copy collector were modified to do partial collection by temporarily moving data objects from the heap to the base set as described above. Although many of the specific characteristics of the heap, base set and physical memory were not measured, a general improvement of about 30% was seen for each collector over several tests. This is certainly a substantial gain, but hardly enough to cause real excitement.

Where did the improvement come from? There are two bases for the speedup. First, by reducing the size of the collectable heap it is possible to avoid much of the marking or copying that would normally happen. The entire base set is to be traversed, so pointers into the base set are not recursively followed. Thus, no effort is spent following pointers into the stable part of the heap; the stable heap will be traversed as part of the base set. Second, the randomness of access has been reduced substantially. Random accesses into the stable heap dictated by the base set are replaced by a linear sweep of the stable heap. Basically, work has been reduced by replacing decision (involving which objects are live in the stable heap) with definition and by imposing order on an otherwise chaotic accessing pattern.

It is still necessary to look through the entire base set to find references into the reduced heap. The price is still being paid to look through every single data object that could ever be used by any program that was ever loaded into the system. This is not good. The second heuristic says that the base set can be reduced. This is done by noting that when data objects are created they are initialized to reference other data objects which already exist. It is quite hard to reference something which has yet to be created. Therefore, there is a tendency for references to point backward in time. Of course it is possible for an old object to be modified to reference a newer object, but empirically that is the exception to the rule [BalShi83]. Thus, to garbage collect a particular portion of the heap, it is only necessary to look for references in newer parts of the heap and in places where older parts of the heap have been modified to reference the newer parts. Very few of the variations of garbage collectors discussed so far preserve information regarding order of creation (i.e. relative age) of objects across a collection. Using the heuristics requires having

some knowledge of this age information. As will be shown later, it is useful to differentiate between more than old and new objects; there needs to be multiple age groups. In a scheme developed by Lieberman and Hewitt [LieHew83], it is suggested that the heap be broken into many smaller subheaps. Associated with each subheap is an "age." By maintaining a table (or *entry vector*) of older locations which have been modified to point into each younger subheap, the base set for a given subheap can be reduced to the entry vector plus younger subheaps.

Unfortunately, by looking through newer areas of the heap to find references into older areas of the heap, there will be scanning of a substantial percentage of dead objects that will be treated as live. This is the price to be paid for doing a partial collect of some middle-aged portion of the heap. Based on the heuristics, under normal circumstances it would be unwise to do a collection of some middle-aged subheap since it is the youngest subheaps that contains the bulk of the reclaimable space. If a subheap other than the youngest is to be collected, the best approach would probably be to collect it at the same time younger areas are being collected.

5.2 Generation Scavenging

Ungar improved on the Lieberman and Hewitt algorithm in a scheme known as *Generation Scavenging* [Ungar86]. Generation Scavenging breaks the heap into several logical (not physical) subheaps or *generations*. Each reference contains a *generation tag* which identifies the "age" of the data object. When a reference to a newer generation is stored in an older generation data object, the location of the older data object is added to an entry vector known as the *remembered set*. At garbage collection time the remembered set acts as the base set for the younger generations. By reducing the size of the base set which needs to be searched, the amount of paging that needs to take place will in all likelihood be reduced. If garbage collections occur frequently enough the base set to be searched can be contained totally within the memory resident working set [Dennin68].

Generation Scavenging employs an interesting memory layout to help minimize paging. The dynamic memory is broken into an *old area* and a *new area*. Objects are always created in the new area. When an object has survived for a sufficiently long time, it is moved from the new area to the old area. Old area objects are considered fairly stable. Only occasional garbage collection is performed on objects in the old area; during normal execution only the new area is collected. By moving objects to the old area, the heap size can be reduced.

The new area of the heap is broken into three areas: *NewSpace*, *PastSurvivorSpace*, and *FutureSurvivorSpace*. PastSurvivorSpace and FutureSurvivorSpace perform much the same functions as **FromSpace** and **ToSpace** in the stop and copy collector. During a collection, live objects in PastSurvivorSpace are moved to FutureSurvivorSpace. When the collection is complete, the two spaces are switched in preparation for the next collection. NewSpace is used to avoid excessive VM paging. Rather than allocate new objects in PastSurvivorSpace as would be done in a classic stop and copy scheme, all new objects are allocated in NewSpace. By keeping the new allocation local to NewSpace, the pages always used for new allocation stand a good chance of remaining memory resident. It is not necessary to move the allocation area back and forth as PastSurvivorSpace and FutureSurvivorSpace are switched following collections. Obviously, during a collection, live objects in NewSpace are moved to FutureSurvivorSpace. Indeed, by careful layout of the area for new allocation, Generation Scavenging avoids gratuitous paging and permits the partial collections to be run unnoticed during interactive sessions on the SOAR Smalltalk system [Ungar86].

References contain the age information in a generation tag, so data objects of different generations may be freely mixed in PastSurvivorSpace without fear of losing track of the age of

any individual object. When an object survives a given number of collections, its generation tag is modified to indicate that it has gotten older. When the tag shows that the object is old enough, the object is moved to the old area in a process known as *tenuring*.

Generation Scavenging was first implemented using custom hardware to trap situations in which references to new objects were stored into older objects. Ungar has stated that in retrospect the hardware does not justify itself; it is possible to use inline code surrounding the appropriate stores to identify the exceptions with only minimal expense. High frequency stores to stack frames and registers do not require the inline tests; they are considered part of the youngest generation during execution. This makes Generation Scavenging appear quite attractive for systems implemented on conventional hardware. In fact, Generation Scavenging has been used successfully in a commercial implementation of Smalltalk on a conventional processor [CauWir86].

When evaluating the possible use of Generation Scavenging on the existing Common Lisp implementation used in measurement, a fundamental problem arose. As defined, Generation Scavenging requires the use of a field in the reference to provide the age of the associated data object. There is an additional bit necessary in each data object to aid in maintenance of the remembered set. Experiments done by the author as well as studies by others have all concluded that efficient implementation of tagging in a Lisp system is important to performance. It is not unusual for a Lisp implementation that runs reasonably "safely" to spend approximately 20% of its time dealing with extraction and testing of tags [TaHLPZ86] [SteHen86]. Special purpose Lisp architectures are almost always tagged in recognition of this fact. Lisp implementations that require efficiency on conventional hardware usually exploit some peculiarity of the architecture (e.g. word alignment on byte addressed machines or an address range smaller than the data word size) and use some well thought out encoding scheme to get the best tagging performance possible. Remember, on an untagged architecture, every dereference can require tag removal. Getting the tag field to minimum size with maximum useful information while making access and removal cheap are some of the primary goals of the implementer. Adding a two to four bit field to each reference for the purpose of encoding a generation number may have great impact on how tag handling is done. Even if the tag handling is not too expensive, the loss of bits in the address field may cause an unacceptable loss of address space. Neither the additional tag handling nor the loss of address bits is practical on the system being studied.

5.3 The Ephemeral Collector

About the same time Generation Scavenging was being added to Smalltalk, the Ephemeral Collector was being added to ZetaLisp. Ephemeral Collection is an incremental scheme which utilizes hardware to maintain the entry vectors and to decrease effort necessary to deal with nonuniformity resulting from the interleaving of the user computation (*mutator*) with the garbage collector (*transporter* and *scavenger*). The Ephemeral Collector is much more complex than the following description. For full details see [Moon84].

Much like Generation Scavenging, this scheme is based on a two space copying scheme modified to deal with reduced heap and base set size. The Ephemeral Collector supports two categories of dynamic data (*dynamic* and *ephemeral*) and a single category of static data. Dynamic objects can be viewed as either the *tenured* objects in Generation Scavenging or as a third, intermediate lifetime generation. Within the category of ephemeral, or short lifetime objects, are several *levels* which indicate the age of an object. The level of an object is derivable from its address through the use of a table; no bits in the reference are needed to encode the age. As an object survives collections it moves to higher levels until it is finally moved to the category of a dynamic object. The Ephemeral Collector provides for fine grained control over which levels are being collected at any time.

Hardware is used to maintain the entry vector for the ephemeral regions. When a store occurs, hardware (the *barrier*) which monitors the bus looks at the value being stored. If the tag of the data indicates that the data is a reference, and from looking at the address contained in the reference it is known that this reference is into an ephemeral area, then a mark is made in a page table (*GCPT*) to indicate that the page being written contains a reference into ephemeral space. Should it be necessary to remove a page from physical memory to make room for another page, the system software searches the page looking for references into ephemeral space. If any are found, the page and level of ephemeral object referenced are recorded in a B* tree (*ESRT*). The GCPT and ESRT thus define the base set to be used during collection of any level of ephemeral object. The difference between the remembered set of Generation Scavenging and the GCPT and ESRT of the Ephemeral Collector is in the way they are maintained (inline code vs. hardware) and in the detail of information being kept. The remembered set contains specific objects in the base set. The GCPT and ESRT identify an object according to the page containing it. Because entire pages can be scanned quickly, the detail of exactly which object contains the possible reference of interest is not believed to be necessary.

The improvements suggested or implemented by Lieberman and Hewitt, Ungar, Moon, Ballard, and Shirron have all pointed out ways to increase performance of garbage collection. They are all based on reducing the heap to be collected and the base set to be traversed. Each of these schemes provides a solution to the problem of improving garbage collector performance, but there is a price to be paid. The Ephemeral Collector requires new hardware, the barrier and GCPT, to be present in a tagged architecture to record references from region to region. Generation Scavenging has been shown effective in systems running on conventional hardware, but requires a new tag in each reference. The cost in additional tag handling and reduced address range, along with the effort needed to maintain the remembered set, may make Generation Scavenging impractical for many new or existing implementations.

6. The New Scheme

The scheme we present attempts to supply most of the benefits of improved schemes such as Lieberman and Hewitt, Generation Scavenging and Ephemeral Collection while providing the additional advantages of being reasonably easily graftable onto many existing stock systems, being only minimally disruptive of the system at runtime, and actually benefiting from the existence of virtual memory.

As with the other schemes, the focus of this scheme is performance improvement through the reduction of collectable heap and base set size. The reduction of heap size does improve performance of the collector and allows the effort to be focused in high return areas, but is most important in that it allows substantial reduction of base set size. Time will not be wasted searching through old, relatively stable data. The primary differences between the new scheme and the other improved schemes are in the way the entry vectors or remembered set is maintained and the identification of generations. The new scheme is premised on the idea that in virtual memory systems a less specific form of the entry vector information used in Generation Scavenging and Ephemeral Collection is already being maintained for other purposes.

Previous attempts to make garbage collection cooperate with virtual memory systems have involved the collectors giving hints *to* the virtual memory system, either to change the page replacement algorithm [FodFat81] [BabJoy81] or to "lock down" areas of memory [Ungar86]. This scheme is based on getting hints *from* the virtual memory system about the location of references and the current working set.

In paged virtual memory systems physical memory acts as a cache for pages on disc representing the entire address space of the process. There is usually hardware associated with the physical memory page frames to maintain status information about each page, and to help the translation from virtual to physical addresses. Among the status information normally kept is a bit set by hardware to indicate that a given page has been written. This bit, the *dirty bit*, is used to let the system software know whether or not the memory resident page is in sync with the corresponding disc resident page. If it is necessary to remove the page from memory to make room for another (a process called *paging*), the dirty bit is checked. When it is not set there has been no change to the page; the disc copy is identical and there is no need to write the contents of the page to disc. The page frame may simply be overwritten by the new page from disc. If the dirty bit is set, then the page frame contains changes which must be written to disc before the page frame can be overwritten. As the new page is brought into the memory, the associated dirty bit is reset to indicate that this page is in sync with its counterpart on disc. The new scheme makes use of these dirty bits to maintain the entry vector. Application of the scheme will be developed through a series of examples after first describing the unusual requirements this scheme places on the language and virtual memory systems.

6.1 Requirements on the Language and Virtual Memory System

In order for the scheme being proposed to work, the underlying virtual memory system must support two requests. One is a request to clear all dirty bits for the pages in the process initiating the request. For the moment one could imagine accomplishing this by forcing all dirty pages to be written to disc (probably a very expensive operation!). A more practical solution for clearing dirty bits will be described later. The second request is to return a map indicating which pages in the process have been written (or written and paged out) since the last clear request. The map could be encoded in many ways, but for this paper the map is assumed to be encoded as a vector of bits with each bit representing a page in the process (or some portion of the process). For many systems, this map is not as large as it might first seem. For a 32 Mbyte process on the machine on which experimention was done, which has 4096 bytes per page, the entire map fits in fewer than 256 bytes.

To accomodate the scheme, the language system also requires (not very extensive) modification. Obviously the garbage collector must change.[1] There are two additional modifications aimed at providing enough context to make it possible to scan a single arbitrary page in the process for references into the region being collected.

First, it must be possible to identify the parts of the page map that can contain only valid base set elements. This implies that code and data must be separated by page or that untagged code and data must be encapsulated to make their bounds easy to determine.

Second, we need to be able to linearly scan arbitrary pages in the base set, so data (and code) must be aligned on page boundaries. This makes it possible to determine whether or not something that appears to be a reference actually is a reference and not just some random bit pattern that looks like a reference. Patterns used in representing things such as the internal format

---

1. In a fully tagged system no more changes would be required, but few stock systems are fully tagged. Being fully tagged means that every word in the memory contains information about how to interpret its content. Most systems implemented on conventional hardware tag most data objects but fail to tag areas such as blocks of code which would not normally be accessed as data by the program.

for a floating point number are often, but not intentionally, ambiguous. An easy way to get the necessary context yet provide the flexibility to represent large structures is to use the following allocation approach. If a structure can be allocated totally on the current page, do it. If it will overflow the page, set the remainder of the current page to a known value and start the structure on the next page. If the structure overflows the page even though it starts at the beginning of the page, note that the second and possibly following pages should be searched starting at the first page on which it is allocated. Depending on the algorithm used to search pages during collection, it may be possible to loosen the page alignment constraint if the structure which crosses pages is known to contain only references. The exception table used to identify multi-page objects might be nothing more than a bit table similar to that returned by the virtual memory system containing ones for pages that are continued from the previous page.

6.2 A Stop and Copy Approach

The garbage collectors in which the new scheme is useful all involve traversing the base set to identify live data objects in a heap. Unfortunately, it is not always the case that every reference in the base set identifies some data object in the heap; a reference might point to some other piece of data in the base set or, as is common in the representation of small integers, might actually contain the data. Any effort spent looking through self-contained data in the base set or references from the base set to other parts of the base set is nonproductive and may cause pages to be retrieved from disc that are never referenced except during garbage collections. The first example shows how the use of virtual memory information can help the garbage collector reduce the base set to an approximation of the minimal base set and become less disruptive by reducing unnecessary paging in the process.

Recall a system employing a conventional stop and copy collector. When a collection begins, **FromSpace** is full of data and garbage. **ToSpace** is empty. As the base set is traversed and references into **FromSpace** are found, the data is moved from **FromSpace** to **ToSpace** and a forwarding pointer is left in **FromSpace** to indicate the new location of the data. The reference in the base set is updated to reflect this new location. If a forwarding pointer is found where the data should have been in **FromSpace**, the new location of the data is known and the reference can be updated. When the base set has been entirely traversed, all references from the base set into **FromSpace** have been updated to references into **ToSpace**.

ToSpace is then linearly searched from the beginning looking for additional references to **FromSpace**. Any references to **FromSpace** are treated the same way that references found in the base set would be treated. Because the data moved from **FromSpace** is always moved to the next free part of **ToSpace**, the area to be linearly searched grows as the search progresses. When the end of copied data in **ToSpace** is reached, the garbage collection is complete. **FromSpace** contains only trash and **ToSpace** contains only live data and unallocated space.

Imagine what happens if the dirty bits are cleared just before the collection starts. As the collection begins there are no references into **ToSpace** because there are no data objects there. When the collection ends the pages written will include all pages in **ToSpace** which contain live data, all pages in **FromSpace** which contain forwarding pointers, and all pages in the base set containing references which have been updated. It is the pages in the base set that are important; the other pages in **ToSpace** and **FromSpace** may be ignored. The pages marked in the base set are only those pages that contain references updated to point into **ToSpace**. This is the smallest set of pages containing the minimal base set for the newly collected heap. If another garbage collection were requested at this instant, only the pages in this set would need to be scanned to find all references into the heap. Another collection at this time would be unnecessary, but as the user computation is continued, any additional pages in the base set which are written represent

the only other pages which could possibly contain references into the collectable heap.

When the next garbage collection occurs, it is not necessary to traverse the entire base set to find references into the collectable heap. It is only necessary to scan the pages written during and since the last collection. If a page in the base set contains no heap references and is not written during the computation, there is no need to scan it during the collection.

Although the benefits of this example are not necessarily dramatic, the example is intended primarily as a way of demonstrating the basic function of the scheme. By maintaining information about where references into the collectable heap could have been stored, it is possible to approximate the minimal base set. All of the following examples employ the same approach: just before the first reference from the base set to data in the collectable heap is updated, clear the dirty bits. In this way, if the only subsequent modifications to the base set are to update heap references, then the pages in the base set written at the finish of the garbage collection will be those containing the minimal base set for the collected heap.

As is fairly evident, adding the VM information to the operation of the collector did not require any substantial garbage collector changes other than that necessary to scan the reduced base set. In fact, the first map of dirty pages was generated via a normal traversal of the full base set. The following examples will show ways in which the base set and collectable heap can be reduced to focus on the areas with the highest mortality rate.

6.2.1 Analysis. Superficially this reduction in base set may not sound very useful. To get some indication of how useful this might actually be, the four test programs were analyzed to learn about their base sets and minimal base sets. The results of the measurements follow in Table 1.

TABLE 1. Static Base Set Measurements

Program	Total Base Set *(Pages)*	Without Packages			With Packages		
		Minimal Base Set *(Pages)*	%	Refs to Heap	Minimal Base Set *(Pages)*	%	Refs to Heap
Compile	220	86	39	5403	200	91	16483
NL	276	120	43	7447	254	92	20549
Reduce	278	95	34	6287	247	89	20469
RSIM	239	93	39	5625	210	88	17165

As can be seen in the table, the results are broken into two categories. The partition labeled "With Packages" is a naive measure. These numbers were generated by simply searching all base set pages for references into the heap. Each page containing a reference to the heap is considered part of the minimal base set. A total of all references found is reported in the "Refs to Heap" column. Unfortunately references were spread around enough that about 90% of the pages in the full base set contained at least one reference into the heap and thus would have to be scanned to find all references into the heap. A collection would require a search through approximately 1 megabyte of base set. This does not bode well for the scheme just presented.

A closer inspection of the nature of the heap references showed that, because of the way in which the system encodes symbols, well over half of the references were to package structures. These are references which identify the home package for a symbol [Steele84]. Since there are approximately 20 different package structures contained in the system and packages are by nature fairly static, most can easily have their (small) top level structure absorbed into the base set. This

quirk due to encoding was factored out and the modified results are presented in the "Without Packages" portion of the table. The number of heap references from the base set can thus be reduced dramatically, and the number of pages containing heap references can be reduced to about 40% of the total. This reduction implies that around 60% of the pages in the base set must be searched *only* if they have been written during the user computation. Potentially, this reduction could have a very significant impact on collector performance.

6.3 A Compacting Mark and Sweep Collector Focusing on Newly Allocated Space

Assume a system that uses a compacting mark and sweep collector which has just finished a collection. All live heap data have been compressed into the bottom of the heap. The heap remaining above the live data will be used for future allocation. The currently unallocated area will be referred to as the *new heap*. This example will use the most simplistic method of reducing the collectable heap size. The collectable region of the heap for the next collection will consist only of the new heap. All objects which have survived the just concluded collection will be declared stable: they will be considered part of the base set.

The heap size for the next collection has been reduced to include only the highest mortality area. Now consider what can be done to reduce the base set for the next collection. Optimally, the minimal base set will consist only of those locations in which references into the new heap will exist when the next collection begins.

At this point it is known that there are no references into the new heap because there are no objects allocated there yet. If the dirty bits are now cleared, when the user computation continues, each write that takes place will cause a dirty bit to be set. When the user computation is next suspended for garbage collection, the set of pages written will include all the pages in which references into the new heap were stored. Also included in the pages written will be all pages in the new heap (allocated storage must be initialized) and any other page which may have been side-effected as a result of the computation. What is not included is the set of pages in the base set that have not been written during the user computation. The garbage collection can now occur in a normal manner with a minor exception. Instead of traversing the entire base set, the map from the virtual memory system is requested and only the pages which are contained in the base set and have been written are used. They are linearly scanned looking for references into the reduced heap and later scanned again during the pointer update phase of the collection. When the collection is complete, the dirty bits are cleared and the user computation is continued.

As before, the actual technique for garbage collection has not really been altered; all the same phases normally present in a compacting mark and sweep collector take place. The only difference is in the manner of traversing the base set.

This approach capitalizes on the idea that the minimal base set for reduced heap is substantially smaller than the minimal base set for the previous, larger heap. Identifying only the minimal base set is quite difficult. Neither Generation Scavenging nor the Ephemeral Collector can guarantee that only the minimal base set will be identified since multiple side effects of the same location may result in unnecessary entries in the tables. It certainly is the case that the remembered set used in Generation Scavenging is a much closer approximation of the minimal base set than the set of pages obtained using the proposed scheme.

The above example demonstrated an easy application of the scheme to reduce effort in a compacting mark and sweep scheme. The major problem with this example is that it requires an object to survive only a single collection before it becomes stable (part of the base set). Surviving a single collection is probably not sufficient criteria for making a piece of data permanent in most systems. The stable heap might quickly fill with rapidly dying data.

Resolving this problem will be discussed in the following sections.

It is interesting to note that even though data that has survived a collection is considered to be part of the base set, it is still possible to do collections on the data. By completely traversing that part of the base set which excludes the stable heap and, once again considering the stable part of the heap dynamic, a full compacting mark and sweep collection can be performed. Separating the stable heap from the base set should be quite easy. Fully traversing the base set was the normal operation of the collector prior to the initial clearing of the dirty bits. In fact, at the end of this full collection the system is in exactly the state described at the beginning of the example.

6.3.1 Analysis. The performance of this scheme depends on the realized reduction in the base set and the comparison with methods that record individual references rather than the page containing the reference. The issue is whether there is sufficient overhead involved in scanning the excess data on a page to make the proposed scheme ineffective. The programs mentioned earlier were re-examined to determine how many pages would need to be scanned if only pages modified by the user computation were searched. Table 2 shows the results for the number of pages modified during the complete computation with no intervening collections.

TABLE 2. Dynamic Base Set Measurements

Program	Total Base Set *(Pages)*	Written During Execution *(Pages)*	%
Compile	220	41	19
NL	276	16	6
Reduce	278	37	13
RSIM	239	17	7

The percentage of the total original base set pages that were modified is quite low (6-19%). There is reasonable probability that the reduced base set could be held in physical memory. In fact, there is some chance that all the pages in the reduced base set would be present in memory when a collection was required. Even in the unlikely event that all base set pages need to be paged in, it should be possible to do the entire traversal in no more than one fifth the full base set traversal time.

An additional test was run to learn about the speed at which pages could be linearly scanned. If page scanning is too costly, it would seem that the only reasonable approach would be adoption of an entry vector system using inline code similar to that suggested by Generation Scavenging. There are no clear criteria for deciding the point at which one scheme becomes more attractive than the other. Many factors must be considered. Certainly as pages get smaller, the scanning algorithm begins to look better, since, in the limit of page size matching reference size, they are approximately equivalent. This does not take into account the overhead of maintaining or searching the entry vector, the density of refererences into the heap contained on the pages written, or paging involved in making the page to be scanned available. In the case of the machine used for experimentation, 1024 references can be contained on a single page, so clearly a single reference can be scanned in about 0.1% of the time for an entire page scan. As more references into the heap are contained on a single page, the overhead in scanning the page goes down. The question becomes whether or not on an absolute scale the time differential between entry vector access and scanning is important. Table 3 shows the times necessary to linearly scan three classes of memory resident pages: a page filled with references into the collectable heap, a

page filled with references to data objects not in the collectable heap, and a page filled with non-references. Times given include calculating an address, fetching an item from the page, extracting the tag, dispatching on the tag and performing a range check on references.

TABLE 3. Page Scanning Time (HP9000/350)

Type of Items	Time to Scan 1K Items *(milliseconds)*
References into heap	2.54
References outside heap	2.30
Non-references	1.89

There is no clear conclusion. The absolute times are not excessive (30 to 104 milliseconds to scan all pages touched), but probably much worse than looking through an entry vector containing only actual references into the heap. Nonetheless, there is no consideration of other time expenses such as maintenance of the entry vector. It is not even obvious that this time will not be dwarfed by the time necessary to copy data from one space to the other. It is obvious that scanning a page takes substantially less time than that required to retrieve a page from disc. On the system used a page fault takes between 20 and 30 milliseconds to handle, scanning takes about 2.5 milliseconds. Regardless of comparison to other systems, the ability to avoid page faults will significantly decrease the time spent in a garbage collection. This is one of the key goals in reducing both heap and base set.

Maintenance of the entry vector is an overhead cost which must also be considered in deciding what scheme to use. In the new scheme, there is essentially no time spent maintaining the entry vector. The entry vector is primarily maintained in VM hardware with a small amount of time being spent in maintenance during paging operations. There is some time required to retrieve the entry vectors from the VM system when a garbage collection is to occur. Using the inline coded version of Generation Scavenging, unnecessary page scanning is avoided by keeping much more detailed information in the entry vector. When a store operation to a location other than the stack or registers occurs, a test must be made to see if the generation tag of the destination object indicates an older object than the generation tag of the value being stored. If a young object is being stored in an older object it is necessary to add the destination object to the entry vector if the value being stored is a reference into the collectable heap and the destination object is not already in the entry vector. On the experimental machine, the inline code to extract the two generation tags, make the comparison, and conditionally jump to a location where a more detailed routine could decide if a new object should be added to the entry vector, requires a minimum of five instructions. This assumes registers are available to hold the generation tag values. Adding the necessary five instructions to each appropriate write will cause from 7.6 to 15.2% more instructions to be executed in the test programs (not including any instructions executed if the entry vector may need modification). These results were generated from information contained in Table 4. Clearly, this is not an insignificant cost.

A possibility which has not yet been mentioned, but which may be useful in systems which do not support virtual memory or which do not provide the dirty page information, is the use of inline code to keep track of pages written. This also provides a basis for making a cost comparison of a detailed entry vector and a dirty page map. If the inline code merely records the pages written in a vector of bits similar to that returned from the VM system, then the benefits of the new scheme can be realized without any virtual memory modification. The cost is that of the inline maintenance of the dirty page map. This cost may be traded off against any time spent in

the virtual memory system maintaining or returning the dirty page map. The inline code is only required to extract the page number that is the destination of the store and set the appropriate bit in the map to reflect the write. On the experimental machine, two instructions would be required at each write to record the information. This represents a 3.0 to 6.1% increase in the number of instructions executed and assumes that the map is at a fixed location and a register is available for the page number.

If we now make the (very simplistic) assumption that all instructions execute in the same amount of time, a reasonable comparison of the cost of entry vector maintenance versus page scan time can be made. The test program Reduce represents the worst case situation for the new scheme (i.e. minimum write ratio with one of the largest percentages of pages written), so it will be used in the comparison. On a 4 MIP machine with no paging or garbage collection, Reduce should execute in approximately 18.8 seconds. If individual locations are stored in the entry vector, the overhead would be approximately 1.429 seconds (just for generation tests). If only dirty pages are recorded, the overhead would be about 0.564 seconds. The difference is 0.865 seconds which would be the equivalent of scanning 346 pages. Since Reduce only writes 37 pages during its entire run, the time difference corresponds to scanning all these pages more than nine times. Thus, using this very simplistic analysis, the inline dirty page map approach is faster than the inline version of Generation Scavenging until garbage collections are necessary at intervals of less than two seconds. There is substantial reason to believe that the use of the VM system to maintain the dirty page map will result in even less overhead, and therefore better performance.

6.4 Stop and Copy Collection with Stable Data

The stop and copy example given earlier only reduces the base set by eliminating some self-contained structure. This example will show how the stop and copy scheme can be modified in much the same way as the compacting mark and sweep to reduce the space being collected and, therefore, further reduce the base set.

Consider a heap divided into two equal spaces. The dirty bits are cleared. New data objects are allocated starting from the lower end of the upper space (i.e. starting from the middle of the heap), and continuing toward the upper end of the space (Figure 3a). The user computation continues until the upper space is filled with newly allocated data objects. The user computation is suspended for garbage collection.

The dirty bit map is requested but not cleared. A normal stop and copy collection proceeds using bits in the dirty bit map to determine which pages to scan. The upper space is treated as **FromSpace** and the lower space is treated as **ToSpace** (Figure 3b). Surviving data is copied to the lower end of **ToSpace** and scanned for other references into **FromSpace**.

When the collection is completed, all live data from the upper space has been copied to the bottom of the lower space. This is the data which has survived a collection and will be considered stable. It no longer needs to be considered part of the heap. Removing data from the heap obviates the need to have storage in both spaces to allow the data to be copied back and forth. In essence this means that stable data has half the storage requirement of dynamic data.

Because this new stable data is sitting at the bottom of a contiguous heap, it is possible to remove it from the heap by defining a new bottom for the heap which sits just above the data being removed (Figure 3c). The newly defined heap is divided into two equal spaces and, after the dirty bits have been cleared, the user computation may be continued once again using the upper space for allocation.

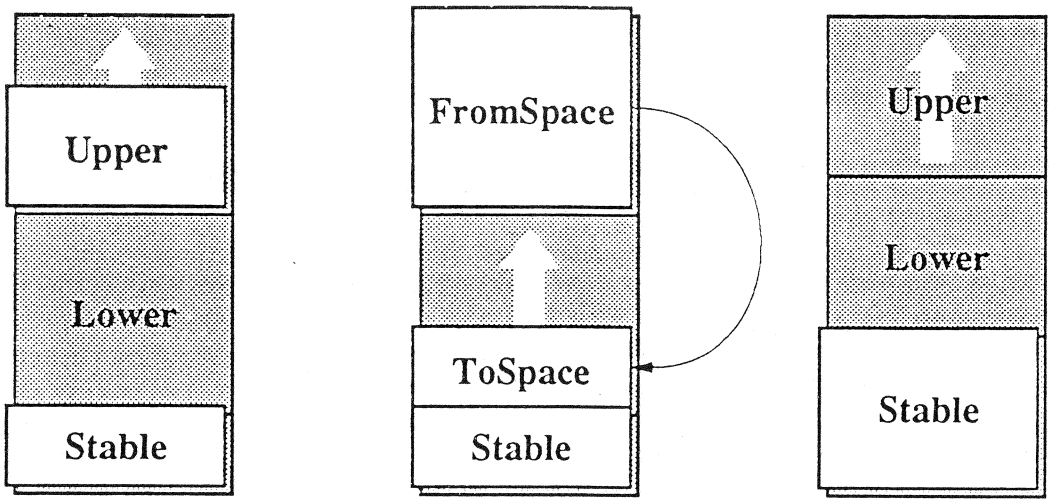


Figure 3. Heap Layout for Quick Stabilization

Even though this layout differs dramatically from the Baker semispaces [Baker78] used in most stop and copy garbage collectors, the fundamental operations involved in setting up the spaces, allocating and copying data are identical. There are several benefits of this approach over a conventional stop and copy collector. First, the collectable heap has been reduced to only the upper half of the dynamic heap and the newly allocated, high mortality objects contained in it. The base set can be reduced to reflect this reduction in collectable heap. This should result in faster collections focused in high return areas.

Second, paging performance should be improved since the new allocation is always being done in the upper half of the heap. The top of the lower space in the heap should never be touched during normal operation since data is only copied to the bottom of the lower space. The pages corresponding to the top of the lower space should be able to remain on disc only to be brought in during full collections or in the unlikely event that almost the entire collectable heap survived a collection.

Third, conventional stop and copy collectors can only utilize half of the available data memory since any collection may require copying all of the data in **FromSpace** to **ToSpace**. This scheme allows up to two thirds of the space to be utilized while still permitting an almost complete conventional stop and copy collection of the entire stable and dynamic data memory.

The full collection is slow but possible. It should be triggered when, just after a reduced collection, the stable area exceeds one third of the available data space. Note that just before the reduced collection approximately two thirds of the data space was in use. If the stable heap does not exceed half of the available data space, then dividing the available data space into two halves and performing a full conventional stop and copy collect after clearing the dirty bits will reduce the stable area but leave it in the upper half of the data space. A second (reduced base set) collection either before or after allocating the remainder of the upper half of data space will minimize the stable data size and return it to the proper place. After clearing the dirty bits and re-establishing the upper and lower spaces, partial collections can continue if stable space was reduced to less than a third of available space or if it is acceptable to lose the ability to collect stable data space using a stop and copy collector.

If, when the full collection was indicated, the stable heap was over one half of the available space, a more complex sequence must occur. A full stop and copy collection must be done to move the newly stable data from lower space back to the upper space. The old stable space must be treated as part of the base set. Then one full and one reduced base set collection must be done to move the stable area into lower space and back while considering upper space part of the base set. A final full collection of upper space to lower space with stable space in the base set completes the process. The only structures which cannot be collected using this regrettably complex scheme are circular structures which cross the boundary between stable space and dynamic space. Of course it is always possible to use a compacting mark and sweep collector to regain all possible space, but this means that two different types of collectors must be supported.

6.5 Minimizing the Dirty Pages

This paper is written with the intent of describing techniques to improve garbage collection performance in existing or planned systems implemented on conventional hardware. It is not considered reasonable to expect a major overhaul of an existing language system as a prerequisite of adopting the techniques presented. Nonetheless, in analyzing some of the data generated, it became quite evident that a single aspect of data representation could have major impact on both the virtual memory and garbage collector performance.

Locality of reference is important for efficiency in virtual memory, and minimizing the number of dirty pages in the base set is important for scanning efficiency in the garbage collector. Writing to a particular page causes it to be added to the set of pages to be scanned during garbage collection. It is important to understand more about the nature of writing in the system. The four test programs were again analyzed. All data presented in this section are based on dynamic measurements.

Table 4 shows the frequency of writes to memory (not the stack) as a function of instructions executed. The test programs tend to have about 1.7% of their instructions writing to memory. RSIM was the only exception with 3% of its instructions performing writes. The most probable cause for the increased write rate in RSIM is data representation: RSIM makes heavy use of floating point arithmetic. Floating point numbers are fairly large and are created for each floating point operation that returns a floating point result.

TABLE 4. Dynamic Frequency of Writes

Program	Instructions Executed *(thousands)*	Writes Executed *(thousands)*	Write Frequency %
Compile	471,087	8,543	1.8
NL	703,344	11,591	1.7
Reduce	75,187	1,146	1.5
RSIM	23,849	725	3.0

The write data in the table does not differentiate between writes to pages in the base set and writes to the newly allocated heap. In order to understand the writes which define the pages in the base set to be scanned it is necessary to eliminate writes to the new portion of the heap. For the purpose of this analysis, writes done to initialize newly allocated data are factored out. The amount of storage allocated to each of the Lisp data types was measured over the duration of each of the programs. Almost all of the storage allocated is dynamic. There are only a few instances in which the allocation is done in the static area of the heap. The memory allocation information

is shown in Table 5. Also measured was the number of writes made to data objects of the various types. Based on the total number of writes made to each data type and the number of writes necessary to handle initialization for the newly allocated data, a measure of the number of writes that could result in pages being added to the scanning set was generated. This set of results is shown in Table 6. CONS cells are clearly the dominant data type allocated in the heap. This is not too surprising. However, once a CONS cell is allocated and initialized, it becomes relatively unimportant as the destination of a write operation.

TABLE 5. Memory Allocation by Data Type

Type	Compile		NL		Reduce		RSIM	
	4-byte Items *(thousands)*	%	4-byte Items *(thousands)*	%	4-byte Items *(thousands)*	%	4-byte Items *(thousands)*	%
Cons	2,472	73.8	1,383	45.7	370	75.2	290	48.3
Vect/Struct	721	21.5	1,301	43.0	50	10.1	14	2.3
String	127	3.8	158	5.2	51	10.4		
Float							296	49.3
Object			97	3.2				
Ratio			87	2.9				
Symbol	32	1.0			1	0.2		
Fundef					21	4.3		

TABLE 6. Writes to Data Type (ignoring initialization)

Type	Compile		NL		Reduce		RSIM	
	Writes *(thousands)*	%	Writes *(thousands)*	%	Writes *(thousands)*	%	Writes *(thousands)*	%
Symbol	3,845	74.1	2,529	29.5	434	66.4	239	87.6
Vect/Struct	611	11.8	4802	56.1	107	16.4	33	12.1
Cons	351	6.8	466	5.4	20	3.1		
String	385	7.4	287	3.4	80	12.2	1	0.4
Object			481	5.6				
Fundef					13	2.0		

The rows labeled "Symbol" contain the most interesting information. No symbols at all are allocated during execution in two of the four programs. The two programs which do allocate symbols, Compile and Reduce, use less than 1% of their total dynamic allocation for new symbols. Symbols appear to be relatively static. Yet when it comes to non-initializing writes, at least 29% and as many as 87% of all writes are to symbols. As shown in the table the NL program is skewed toward writes to vectors. Upon examination as to the reason for the skew, it was found that approximately 46% of the total non-initializing writes (to all types) in NL are to a single vector. This vector is used by the interpreter to cache arguments passed from compiled code. If all code were compiled, the percentage of symbol writes would rise to well over 50%. Even factoring out the effects of code interpretation, NL has a lower symbol write rate than the other programs. This is believed to result from NL's use of an object-oriented extension to Common Lisp. The use of objects has a tendency to shift accesses of global values to accesses of instance variables which are held in data objects of type "object." Accesses of the object data type account for an additional 10% of the writes in NL.

With the exception of the NL program, writes to symbols exceed writes to any other data type by at least a factor of four. The symbols being discussed are not the local variables used in functions. Local variables are stored in activation records on the stack. The symbols being written are in the global name space.

It certainly appears that some effort should be applied to maximizing the density of symbols in the system. Some measurements were made of the base set to determine what type of data objects were modified on the pages written in the dynamic base set. Table 7 shows the breakdown of pages by contents.

TABLE 7. Type Written on Base Set Pages

Program	Total Pages Written	Symbols Written Only	%	Non-Symbols Written Only	%
Compile	41	36	88	5	12
NL	16	13	81	3	19
Reduce	37	34	92	3	8
RSIM	17	15	88	2	12

The large majority (81-92%) of pages written were accessed only to alter a symbol. Additional tests showed that there were many more symbols written than pages, so there is some locality to the symbol accesses. How can the locality be increased so that fewer pages would need to be scanned or included in the working set? In other words, what is the best way to get the maximum number of symbols onto a single page?

Consider what constitutes a Lisp symbol. According to Common Lisp [Steele84], a symbol consists of five parts: a function cell, a value cell, a property list, a home package, and a name. The five fields are independent and each may be viewed as being represented by a uniform sized reference. It is not important to understand the use of each field, only their relationship. There are two obvious ways of encoding the five fields: they may be encoded into a five field record which represents a single symbol, or they may be encoded as the contents of an identical offset into five parallel vectors. Each vector contains the same field for all symbols. The record approach is used by the experimental system and has the benefit of keeping all fields for a single symbol quite local. There is an additional benefit in that, because each symbol is independent, symbols may be created and reclaimed quite easily. The parallel vector approach is not so flexible, but keeps fields local by the type of field, not by the symbol. That is, the function cell and value cell of a given symbol are probably spread far apart using this encoding, but the function cells of all the different symbols are as close as is possible. The problems with the parallel vector approach are that the vectors are not easily extensible and reclamation of unused symbols can become quite messy.

Since the two schemes offer different types of locality, it is necessary to determine which will be most beneficial. Through examination of all defined symbols in the system, it was found that a symbol tends to be used for one of three purposes: a function name, a global variable, or just a name (i.e. a flag). The use of a symbol as a name has no bearing on locality, so that use will be ignored. The use of symbols as function names and global variables tend to be disjoint: fewer than 10% of the symbols in the system had both a bound function cell and a bound value cell. The access rates of the different fields are also important. The bulk of read accesses to symbols are to the function cells. Almost every time a new function is to be called, it is necessary to

access a function cell. This represents about 4% of the instructions in the experimental system. Value cells are a distant second most likely to be read. The remaining fields are rarely read. As for write access of symbol fields, things change as Table 8 shows. Essentially every write access of a symbol is a write of the value cell. For the four programs, the *lowest* non-initializing write rate of the value cell was 99.59%.

TABLE 8. Writes Within Symbols by Part of Symbol Written (ignoring initialization)

Symbol Part	Compile %	NL %	Reduce %	RSIM %
Function	0.13		0.03	
Value	99.59	100.00	99.87	100.00
Prop-List	0.13		0.07	
Package	0.05		0.02	
Name	0.10		0.01	

The data is all very supportive of the parallel vector encoding. Use of the record style of encoding surrounds one of the two most useful fields of a symbol with four fields that have little chance of being accessed. Useful locality is artificially reduced. By using the parallel vector encoding, there is a greater probability of keeping the heavily used parts of the symbols memory resident while allowing the relatively unused portion of each symbol to migrate to secondary storage. Based only on locality and accessing issues, there seems to be a clear winner. A program using lots of functions will make better use of virtual memory with the parallel vector encoding. The same encoding means that the value cells are all closer together resulting in more writes falling on fewer pages. Scanning fewer pages should improve garbage collector performance.

6.6 Adding Aging

It has been stated that new data has a much higher mortality rate than old data. The obvious problem is defining the age of an object. The previous two examples provided a very simple way of establishing age: young data is that which has not survived a collection, old data has. This may be acceptable, but certainly requires some form of evidence that it is sufficient for a practical system.

What is a reasonable technique for determining age? The obvious direct use of wall clock time does not seem appropriate since the lifetime of an object on a slow processor would be longer than the lifetime of the same object on a faster processor. The age needs to be tied in some way to the computation. There are many possible metrics which could be used. The one which will be used here is one of the simplest to implement that still maintains some intuitive meaning. This metric bases the age of an object on the amount of storage allocated. Each time a fixed amount of storage is allocated, a new generation is created.[2] If it is believed that a computation has a roughly constant allocation rate, the age can be converted into wall clock time scaled for the speed of the processor.

---

2. It can be argued that the aging of data (i.e. frequency of reclamation) should be based on wall clock time since virtual memory algorithms often include time in page freeing policies. The justification for excluding time directly is that it seems inappropriate to expend any effort reclaiming garbage if there is no demand on the memory resource. As long as new data objects are not being allocated, the primary benefit of periodic reclamations is to increase the density of the working set. It is felt that if the area of new allocation is sufficiently small, the impact of these periodic reclamations on the size of the working set should not be very significant.

Experiments were run to both test mortality rates and learn more about how to deal with aging data objects. The system used for the experiments contains a hand-coded compacting mark and sweep collector. This particular collector has the very useful property that the compaction phase maintains the creation order of the data objects. That is, a given data object in the heap is younger than those lower in the heap and older than those higher in the heap. Consequently, markers placed in the heap can be used to gather statistics about the mortality rate in various generations.

The collector was modified to keep track of the size of each generation in the heap both before and after a collection. Statistics were maintained for the nine youngest generations and for a single group including all older data objects. As described above, to give some intuitive meaning to the concept of a generation, after each collection the region for new allocation was always initialized to a constant size. The size of the area to be allocated was increased by factors of two from 16 kilobytes to 1 megabyte for different runs. A constant allocation area implied that collections would occur at roughly equal time intervals if the rate of new allocation could be approximated as a constant. Were this not done the collections would occur at times which depended on the number of data objects in previous generations; when lots of data objects existed, the collections would become more frequent and objects would age more quickly.

The test programs were analyzed with respect to their allocation characteristics. The amount of storage allocated and the number of garbage collections required to run each test is given in Table 9. A plot of the fraction of the originally allocated area which survives to a given generation is given in Figure 4. For the purpose of this table and plot, a generation is said to pass when 32 kilobytes of the heap have been allocated. This corresponds to 8192 references being allocated during a single generation.

TABLE 9. Storage Allocated During Execution

Program	Total Storage Allocated *(megabytes)*	Generations During Run *(32KB/Gen)*
Compile	13.6	435
NL	12.4	398
Reduce	2.0	64
RSIM	2.6	84

We are unaware of any other quantitative studies of data object lifetimes in Lisp. Ungar has done a study of the lifetimes of objects in Smalltalk which involved a single test program that only required three collections before termination of a single iteration [Ungar86]. Nonetheless, the Smalltalk data is similar in appearance to that reported here. There are significant results contained in Figure 4. First, even though these programs are different (yet reasonably representative of some normal types of Lisp loads) they all substantiate the premise that the mortality rate of young objects tends to be significantly higher than that of older objects. Even the lowest first generation mortality rate is over 69% with only 32 kilobytes being allocated per generation. With larger allocation areas the rate rises quickly into the 80-98% range. The mortality rate does not always monotonically decrease as the generation number increases, but a weak tendency in that direction is apparent. In every case the mortality rate of the youngest generation far exceeds that of any other generation.

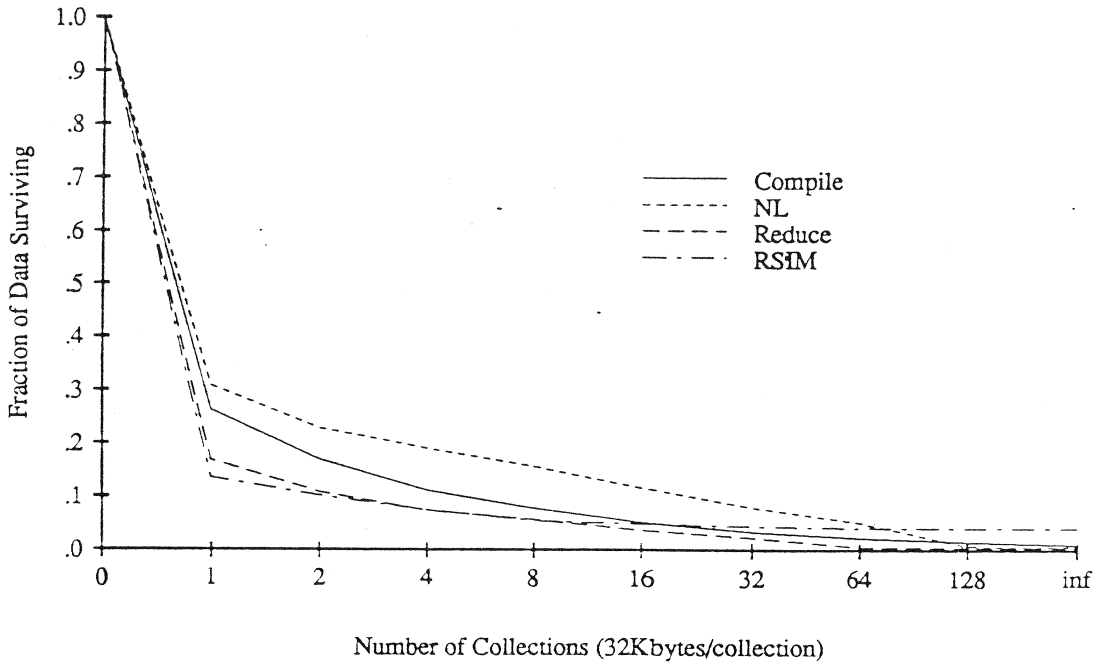


Figure 4. Data Object Survival Rate in Four Programs

Second, since the members of older generations are just the survivors of younger generations it is also true that, in absolute terms, the storage reclaimed from the youngest generation is greater than the storage reclaimed from any other single generation. In fact, the raw data from which the plot was created indicate that on the average the storage recovered from the youngest generation exceeds that recovered from all (eight) other generations.

Third, the lifetimes of data objects in the different programs are not the same. Certainly they exhibit strong similarities, but, for instance, RSIM data objects seem to die quite rapidly with about 4% surviving the entire run while NL data objects average much longer lifetimes and dwindle to less than 1% when the run is complete. Much of the difference can be attributed to the nature of the data objects generated. RSIM is doing a large amount of floating point computation while NL does none. On the system being used, floating point computations always produce new allocations of a floating point object to hold the result. Since floating point data objects used for intermediate results in calculations have relatively short lifetimes and are created in substantial numbers by RSIM, the mortality rate is increased for small allocation areas. The increased use of global data structure in RSIM accounts for discrepancy in the percentage of data objects that survived the entire run.

Does the idea of making data stable after it has survived a single collection make sense? For sufficiently large allocation areas, this may be quite reasonable since the mortality rate will likely be well over 90% and the frequency of collections will be low. Unfortunately, for smaller allocation areas the mortality rate among data that has survived a single collection may not be low enough to prevent the heap from filling with uncollectable data that has died after surviving a single collection. Ungar has referred to the problem of stabilizing data too quickly as *premature tenuring*.

There is a tradeoff involved in choosing the size of allocation area. If the area is too small, collections will be frequent and not very effective. If the area is too large, collections will not be as frequent and will be much more effective, but may result in long delays due to sheer size and unwanted paging due to the inability to fit the entire area into physical memory. For a system which makes data stable after only surviving a single collection it would be best to err on the side of making the allocation area too large, but an optimum size would still allow the area to reside in memory.

There are benefits to using small allocation areas in determining generations. Garbage collections take less time and the entire heap stands a better chance of remaining in physical memory. There are also benefits to using larger allocation areas. Fewer short lived data objects will become stable so fewer full scale collections will be needed. By requiring multiple generations to be survived before stabilization, the benefits of both sizes of allocation areas can be realized. The next section will explore ways in which data objects can be made to survive an arbitrary number of generations before moving out of the collectable heap.

6.7 Handling Arbitrary Numbers of Generations

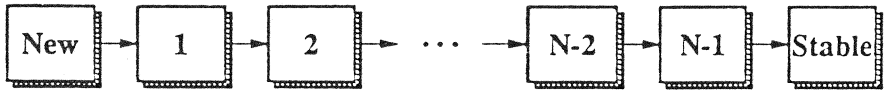
This final example will show how the use of virtual memory information can be combined with storage layout to create an efficient collection scheme that

- focuses on high mortality areas to reduce effort,
- is not excessively disruptive of the working set for a computation,
- allows for stabilization of data after an approximate age has been reached, and
- permits better utilization of memory than more conventional schemes on conventional hardware.

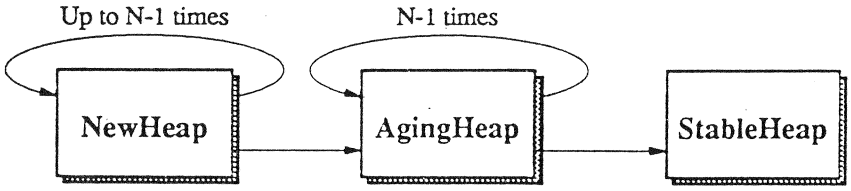
The method for using virtual memory information has been demonstrated in each of the preceding examples and is now restated. At a time when the area to be collected during the next collection is vacant, clear the dirty bits. References to the collectable area cannot exist outside the set of pages which will be written as the area is allocated.

The new problem which must be addressed in this example is how to handle an arbitrary number of generations. Generation Scavenging includes a field in each reference to indicate the generation of the referenced data object. The use of this field allows generations to be mingled without fear of losing track of age. This approach has some great characteristics, but, as mentioned earlier, this approach was not considered viable for use in the existing implementation. The additional tag field would require extensive redesign and would have severe effects on performance.

It is viewed as critical that the scheme not be overly complex nor put excessive constraints on memory utilization. For instance, the most straightforward way to handle an arbitrary number of generations would be to have one heap for each generation. As a data object ages it would pass from heap to heap. Figure 5a illustrates the organization of heaps for a system requiring survival of N garbage collections before a data object could become stable. This approach converts the explicit field used in Generation Scavenging into positional information; the generation of an object could be determined by its location. The problem with this approach is that it requires the division of a single contiguous heap into several smaller heaps. Problems concerning how the heap should be subdivided and how to adjust the various sizes dynamically will probably result in poor memory utilization and prevent this from being a viable solution.



(a) Simple view



(b) Modification for easy implementation

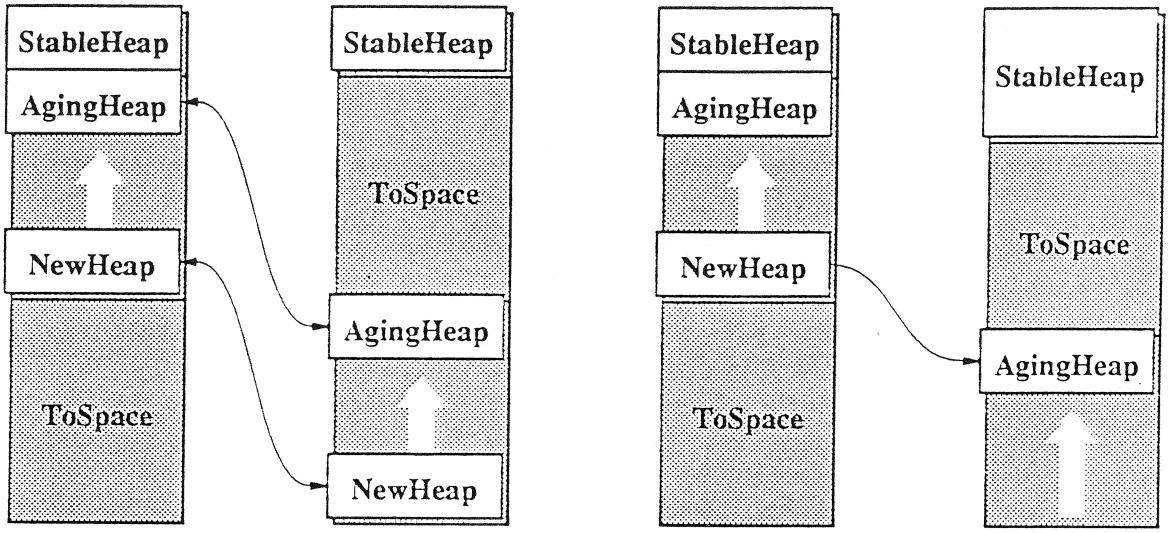
Figure 5. Views of a Multi-Generation Heap

Lieberman and Hewitt associate an age with each page of data. This scheme avoids the problem of having independent heaps since heaps for the various generations are segregated only by pages. Nonetheless, this scheme was not extensively investigated because it superficially appears to require a fair bit of bookkeeping and elaborate memory allocation to make everything work properly.

The storage layout chosen is an engineering compromise which trades control over the exact age at which an object becomes stable for high memory utilization with a very simple layout and garbage collector. The layout is based on the use of a stop and copy collector. The easiest way to understand the scheme is to view data space as consisting of three independent regions: **StableHeap, AgingHeap,** and **NewHeap.** These three regions form a bucket brigade as shown in Figure 5b. Operation of the scheme is quite similar to the operation of the N stage scheme just described. **StableHeap** is where data that has survived sufficient collections resides. Data in **StableHeap** is only subject to garbage collection at the rare occasions when a full collection is signalled. **AgingHeap** is where the guarantee of age is achieved. When data is moved from **NewHeap** to **AgingHeap** all that is known about its age is that the data has survived at least one collection. The data may have survived as many as N-1 collections before advancing to the **AgingHeap.** The data must then sit in **AgingHeap** for N-1 collections before being transferred to **StableHeap.** Because there is no way to differentiate among the various generations that sit in **AgingHeap,** no new data may be added to **AgingHeap** until the full N-1 collection aging period is up and/or **AgingHeap** is emptied. Consequently, new data objects, which may only be generated in **NewHeap,** must remain there through up to N-1 garbage collections waiting for **AgingHeap** to be vacated. When **AgingHeap** is empty the contents of **NewHeap** are then transferred. Any data objects which reach **StableHeap** are thus guaranteed to have survived at least N collections and possibly as many as 2N-1.

Due to the subdivision of a single contiguous heap into three sections it would seem that this proposal has all the same problems as the earlier subdivided heap proposal. Luckily, this is not the case. In fact, it is possible to lay out the three areas in such a way that all necessary movement between heaps and all dynamic sizing of the heaps is accomplished by trivial pointer manipulation.

The memory layout is shown in Figure 6. **StableHeap** sits at the top of memory and grows downward. Below, but touching **StableHeap**, is a pair of equally sized spaces used as the two spaces of a stop and copy collector. At any point in time only one of the two spaces is actually in use, so **NewHeap** and **AgingHeap** must share one of the spaces (**FromSpace**) yet maintain their independence. This is done by allocating **NewHeap** from the bottom of the space and allocating **AgingHeap** (downwards) from the top. A single pointer identifies the boundary between them.



(a) Normal Aging GC

(b) Heap Transfer GC

Figure 6. Layout and Collection of a Multi-Generation Heap

During a normal garbage collection in which there is no transfer between heaps, the VM page map is requested, the dirty bits are cleared, then a slightly modified stop and copy garbage collection is performed. Data objects copied from **NewHeap** are copied to the bottom of **ToSpace**. Data objects to be copied from above the boundary between **NewHeap** and **AgingHeap** are copied to the top of **ToSpace**. See Figure 6a. The linear search of the top of **ToSpace** must also be slightly modified to account for the reverse allocation into the next version of **AgingHeap** being done.

Transferring **AgingHeap** contents to **StableHeap** looks quite similar to a normal nontransfer collection. One constraint is that the transfer must always happen just prior to a collection which will move the data from the space adjacent to **StableHeap**. When in this configuration **AgingHeap** is contiguous with **StableHeap** (Figure 6b). The stabilization of the aging data requires nothing more than moving the boundary of **StableHeap** to include **AgingHeap**. The boundary between the two lower spaces can (but may not need to) be adjusted so that the spaces are again of equal size.

All that remains is to move the data from **NewHeap** to **AgingHeap**. The transfer of **NewHeap** to **AgingHeap** can be accomplished by just moving **NewHeap/AgingHeap** boundary pointer to indicate that what used to be contained in **NewHeap** is now contained in **AgingHeap**. Executing a normal collection will result in everything appearing in the proper place for the next N-1 normal collections to proceed as required.

The same garbage collector can be used for all collections and transfers of data from heap to heap, only the manipulation of three pointers differentiates among the types of collections being done. Because the division of the overall data space into three parts is similar to that used in a previous example, it is possible to follow the same algorithm described for doing a full collection.

If the allocation from the top of **NewHeap** is felt to cause excessive paging, the same technique used in Generation Scavenging can be applied to decrease VM activity. At the bottom of the heap a fixed size area can be defined which may only be used for new allocation. When garbage collection is required, this area would be considered part of **NewHeap**. The heap above the new allocation area should be broken into thirds, and the boundary between the two semispaces used in the collection should be fixed at the first third mark above the fixed space; the upper two thirds will be the original size of the upper semispace. If new allocation is done in the fixed area and the **NewHeap** and **AgingHeap** are copied back and forth as before, with no adjustment of the bottom semispace size when stabilization occurs, then the paging characteristics will be quite similar to those of Generation Scavenging. This refinement would be most useful in systems intending to do very frequent collections (e.g. each few seconds).

6.8 Making the VM Hooks Realistic

The reduction in base set hinges on the ability of the garbage collector to obtain information about the pages in the process which have been written since a particular time in the computation. Establishing the time in the computation from which the record is to be kept is the purpose of the "clear dirty bits" request. Since dirty bits are only set when writes occur, it is not possible to determine if a page has been written since a particular time unless the dirty bit is reset at that time. Unfortunately, the resetting of dirty bits interferes with their normal use by the virtual memory system. They are normally used to indicate that a page is different from the corresponding page held on disc. If the bits are artificially cleared, there is no record that the page must be written to disc to record the modifications when its time for being paged out arrives. The changes will be lost unless it becomes the policy that all pages are written out when they must vacate physical memory. Neither the loss of changes nor the unnecessary disc traffic resulting from writing out every page is acceptable. Also unacceptable is the technique described earlier for clearing dirty bits: writing all dirty pages in the process to disc. The purpose of the proposed scheme is to improve, not degrade garbage collector performance.

Changing the state of the dirty bits indirectly (e.g. by paging out dirty pages) is too expensive because of the time consuming work which must be done, yet only the hardware dirty bits can provide the information needed by the collector. The solution then centers on how to save the state of the bits at a given point in time so that the hardware can be used to record more recent events. The state recorded in the dirty bits is the information necessary for the garbage collection (presuming nothing has been paged out). The combination of the state recorded in the dirty bits and the state saved when the dirty bits were cleared represents the information previously available to the paging software.

How should the dirty bit state prior to a clear request be held? This information is not of any use to the garbage collector, so it should be held in the manner that leads to highest efficiency in the virtual memory system. It can certainly be viewed as a vector of bits in the same orientation as the physical dirty bits are found, ordered in the same way as page table entries. It is not necessary that this state information be duplicated for each process. There is only one set of dirty bits so there only needs to be one set of bits to represent the saved state. See Figure 7.

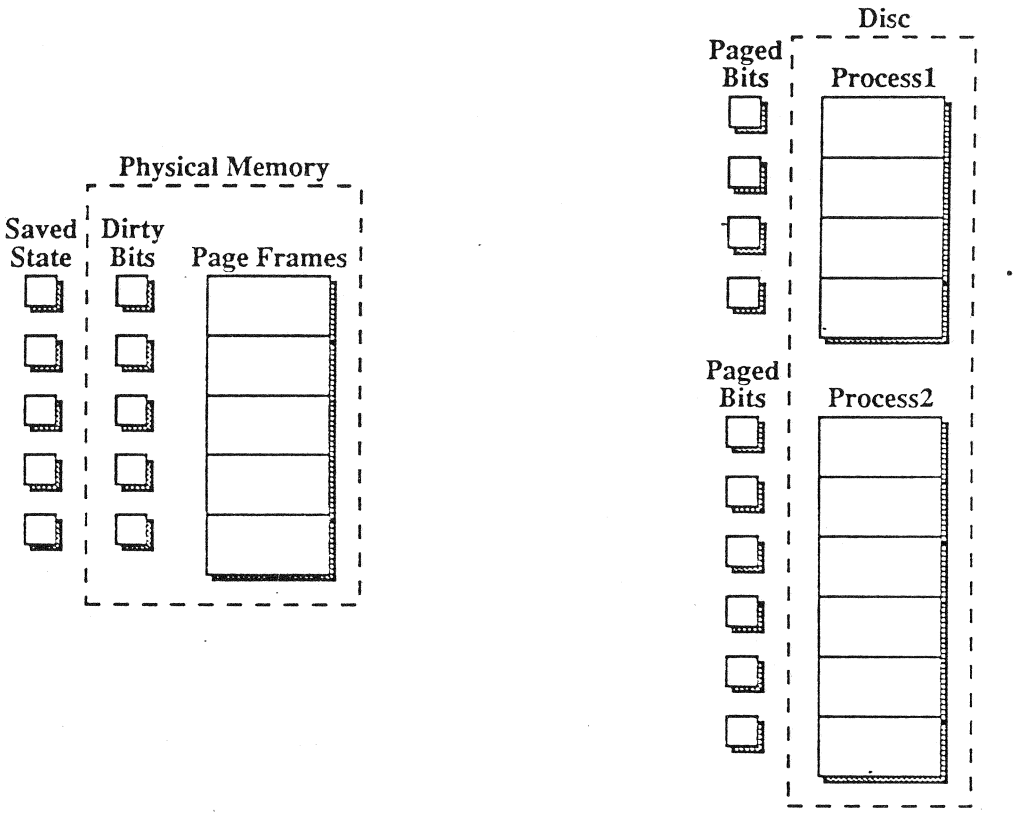


Figure 7. Virtual Memory Structures

Assuming the existence of clearable dirty bits and a bit vector to represent the saved dirty bit state, the clear request would cause the following sequence to occur. Page frames used by the process making the request must be found. For each page frame found, if the dirty bit is clear, it may be skipped. If the dirty bit is set, the corresponding bit in the saved state table must be set then the dirty bit must be cleared.

The paging software must also be modified to be cognizant of the separation of dirty bit information into two distinct areas. When a page is paged in, both the dirty bit and the saved state of the dirty bit must be cleared to indicate that the new page matches the page on disc. When a page is to be paged out, both copies of the bit must be tested to decide if modifications have been made to the page. If either bit is set, the page must be written to disc to record the modifications.

There is still one problem to be solved. Consider the situation in which a page is written then paged out. Because it is not memory resident, there is no record of it having been modified, yet all pages on disc cannot be assumed dirty or else each and every one would need to be paged in for scanning. A modification to the paging out software that recorded dirty pages written out would provide the necessary information. This record would best be held in exactly the same format as the dirty page map requested by the user process. By holding it in the same format, the map which needs to be returned to the user process is nothing more that the inclusive-or of the map generated by looking through the hardware dirty bits and the paged out map. The paged out map should be cleared whenever a clear request is received. It should not be cleared when a paged out page is paged in.

An interesting way to avoid combining maps and improve paging performance as well is to return the dirty page map and the dirty paged out page maps individually rather than as a single combined map. The benefit comes from the fact that the collector is now given information about which of the dirty pages are probably in physical memory and which are probably on disc. By allowing the collector to order the page scanning in such a way that memory resident pages are scanned first, paging can be minimized. Memory resident pages will not accidently be paged out to make room for disc resident pages. If pages in the dirty paged out map can be requested to be brought in from disc without causing the collector to wait, even greater gains are possible.

When virtual memory requirements were discussed earlier in this paper, the two requests mentioned were to clear the dirty bits and to return the dirty page map. For pragmatic reasons it is probably wise to add an additional call that combines both requests into a "return the map and clear the dirty bits" call. The reason for this addition is that operating system calls usually carry a substantial cost in time. Since these calls often occur in close sequence, the ability to combine them may save half of the overhead involved.

6.9 Bears in the Woods

As with any general approach to a problem, there are specific cases in which the approach is inappropriate or unnecessary. One must have some understanding of several aspects of a language implementation and the underlying virtual memory system before blindly adopting the scheme described. It is important to know that virtual memory paging is a substantial expense in the system. If the entire system is memory resident then the only speedups will come from reducing the base set or heap and base set. The big benefit of avoiding disruption of the working set will not be seen. If the overhead involved in clearing or requesting the dirty page information is too large (e.g. on systems which either have inefficient system calls or which "page the page table") or if these actions expose the process to overly expensive rescheduling, much or all of the potential gains may be lost. If the ratio of process size to page size is large the structures needed to maintain information about page state may become too cumbersome to readily handle. There are, however, some techniques of grouping and encoding that may alleviate this problem.

The key point is to understand the various expenses so that a reasonable assessment of the applicability of the scheme to a given system can be made. It is expected that this scheme will probably display the largest benefit on single user systems with a simple virtual memory implementation and a large, but not enormous, virtual to physical memory ratio.

7. Status

The scheme proposed has not yet been fully implemented; it is currently a paper tiger. As is well known, a paper tiger can have arbitrarily sharp teeth. It is hoped that claims made in this paper do not exceed the results that will be realized. Many of the key aspects of the scheme have been based on analysis of real data from real programs running on a real implementation. Special simulations have been done and some parts of the collection schemes have been implemented and tested. Some speculation has been made based on other collection schemes which share some common ideas. Nonetheless, the only way to verify that this scheme will not fall short of expectations (or exceed expectations) is through an actual implementation.

8. Summary

Classic garbage collection algorithms and virtual memory systems interract poorly because garbage collection violates some of the basic premises that make virtual memory effective. Past attempts to alleviate the problem have ranged from giving the virtual memory system hints about

the impending change in memory accessing characteristics during garbage collection to focusing effort in the collection process on smaller heaps and base sets so that garbage collection characteristics will more closely resemble the expectation of the virtual memory system. An approach has been presented which directly uses the information normally maintained by a conventional virtual memory system to reduce collector effort and improve virtual memory performance.

Experiments were done on an existing Lisp system which confirm for Lisp the validity of the data object "lifetime" heuristics observed in Smalltalk systems. These heuristics about mortality rates were used to design a simple heap layout that permits good memory utilization while allowing the garbage collector to control the period of aging necessary before a data object becomes stable.

Although the new scheme is not expected to outperform either hardware-assisted Generation Scavenging or the Ephemeral Collector, it should exhibit similar performance while providing several benefits important to existing or planned implementations on conventional hardware:

- no special hardware is required,
- no new tag field must be handled,
- no address range decrease is required,
- no runtime degradation of the language system, only page alignment guidelines must be followed, and
- good potential for retrofitting to existing systems.

There is a price to be paid in implementing this scheme. The language system must be designed or modified to obey some simple alignment rules, and the virtual memory software must be modified to maintain and provide information about some tables which only need to be touched during paging operations and at the time of a user request. The modifications to the virtual memory are straightforward and general; the data maintained could prove useful to users of other programs wishing information about the modified portions of the working sets of their processes.

Garbage collectors have typically had to do more work than necessary because they have had essentially no information about the process for which they collect. When memory runs out, the collector has to look through everything to find the garbage. Virtual memory systems know what has been going on in a process during the recent past and expect similar things to continue. If the virtual memory keeps just a little more information than usual and makes it available to the garbage collector so that it too can have knowledge of what has transpired in the process, the job of collection can be simplified and the expectations of the virtual memory system can be better met.

References

[BabJoy81] O. Babaoglu and W. Joy, "Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Referenced Bits," *Proceedings of the Eighth Symposium on Operating Systems Principles*, Pacific Grove, California, 1981, 78-86.

[Baker78] H. Baker, "List Processing in Real Time on a Serial Computer," *Communications of the ACM*, Vol. 21, 4 (April 1978), 280-294.

[BalShi83] S. Ballard and S. Shirron, "The Design and Implementation of VAX/Smalltalk-80," in *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner (editor), Addison Wesley, 1983, 127-150.

[CauWir86] P. Caudill and A. Wirfs-Brock, "A Third Generation Smalltalk-80 Implementation," *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, Portland, Oregon, 1986, 119-130.

[Cheney70] C. Cheney, "A Nonrecursive List Compacting Algorithm," *Communications of the ACM*, Vol. 13, 11 (November 1970), 677-678.

[ClaGre77] D. Clark and C. Green, "An Empirical Study of List Structure in Lisp," *Communications of the ACM*, Vol. 20, 2 (February 1977), 78-87.

[Dennin68] P.J. Denning, "The Working Set Model for Program Behaviour," *Communications of the ACM*, Vol. 11, 5 (May 1968), 323-333.

[Dennin70] P.J. Denning, "Virtual Memory," *Computing Surveys*, Vol. 2, 3 (September 1970), 153-189.

[DeuBob76] L.P. Deutsch and D. Bobrow, "An Efficient Incremental Automatic Garbage Collector," *Communications of the ACM*, Vol. 19, 9 (September 1976), 522-526.

[FenYoc69] R. Fenichel and J. Yochelson, "A LISP Garbage-Collector for Virtual-Memory Computer Systems," *Communications of the ACM*, Vol. 12, 11 (November 1969), 611-612.

[FodFat81] J. Foderaro and R. Fateman, "Characterization of VAX Macsyma," *Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation*, Berkeley, California, 1981, 14-19.

[LieHew83] H. Lieberman and C. Hewitt, "A Real-Time Garbage Collector Based on the Lifetimes of Objects," *Communications of the ACM*, Vol. 26, 6 (June 1983), 419-429.

[Moon84] D. Moon, "Garbage Collection in a Large Lisp System," *ACM Symposium on Lisp and Functional Programming*, Austin, Texas, 1984, 235-246.

[Rovner85] P. Rovner, *On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language*, CSL-84-7, Xerox PARC, Palo Alto, California, 1985.

[Steele84] G. Steele, *Common Lisp: The Language*, Digital Press, 465pp., 1984.

[SteHen86] P. Steenkiste and J. Hennessy, "LISP on a Reduced-Instruction-Set-Processor," *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, Cambridge, Massachusetts, 1986, 192-201.

[TaHLPZ86] G. Taylor, P. Hilfinger, J. Larus, D. Patterson, and B. Zorn, "Evaluation of the SPUR Lisp Architecture," *Proceedings of the Thirteenth Symposium on Computer Architecture*, Tokyo, Japan, 1986, 444-452.

[Ungar86] D. Ungar, *The Design and Evaluation of a High Performance Smalltalk System*, Ph.D. Thesis, UC Berkeley, UCB/CSD 86/287, March 1986.

[White80] J. White, "Address/Memory Management For A Gigantic LISP Environment or, GC Considered Harmful," *Conference Record of the 1980 LISP Conference*, Redwood Estates, California, 1980, 119-127.