A Lifetime-based Garbage Collector for LISP Systems on General-Purpose Computers

Patrick G. Sobalvarro

Submitted in Partial Fulfillment of the Requirements of the Degree of

**Bachelor of Science
in
Electrical Engineering and Computer Science**

at the

**Massachusetts Institute of Technology
September 1988**

Author____
Department of Electrical Engineering and Computer Science
February 1, 1988

Certified by______________________________________
Robert H. Halstead, Jr.
Associate Professor
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by______________________________________
Leonard A. Gould
Chairman, Department Committee on Undergraduate Theses

A Lifetime-based Garbage Collector for LISP Systems on General-Purpose Computers

Patrick G. Sobalvarro

Submitted in Partial Fulfillment of the Requirements of the Degree of

Bachelor of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

Abstract

Garbage collector performance in LISP systems on custom hardware has been substantially improved by the adoption of lifetime-based garbage collection techniques. To date, however, successful lifetime-based garbage collectors have required special-purpose hardware, or at least privileged access to data structures maintained by the virtual memory system. I present here a lifetime-based garbage collector requiring no special-purpose hardware or virtual memory system support, and discuss its performance.

Thesis Supervisor: Robert H. Halstead, Jr.
Title: Associate Professor
Department of Electrical Engineering and Computer Science

Contents

List of Figures

List of Tables

Introduction

The pointer-oriented semantics of LISP, and the nature of heap allocation, often result in poor virtual memory performance on general-purpose computers. Garbage collection, in particular, has traditionally required examination of at least all live data created by user programs (in copying garbage collectors) and sometimes of storage recovered as well (in mark-sweep garbage collectors).

Lieberman and Hewitt introduced in 1981 a garbage collection algorithm based on the lifetimes of objects [9]. By grouping objects according to their ages, the proposed garbage collector avoided examination of relatively old objects when garbage collecting relatively young objects. LISP implementations on machines with special-purpose hardware or instruction sets microcoded to support this algorithm have realized significant performance improvements, as noted by Moon [10].

Moon's contention was that the overhead of bookkeeping required to keep track of references to newly-created objects in lifetime-based garbage collectors on general-purpose computers would result in prohibitive performance degradation. Shaw [12] has recently described a scheme in which the virtual memory hardware present on modern general-purpose computers can be used to keep track of newly-stored pointers, provided one has access to the data structures used by the virtual memory system.

I describe here the portable lifetime-based garbage collector used in Lucid Common LISP on the Apollo and Sun workstations. As Lucid Common LISP is portable, this garbage collector does not have the cooperation of the virtual memory systems on the computers it runs on. Nor does it make use of special-purpose hardware; still, the techniques of lifetime-based garbage collection are sufficiently powerful that overall system performance is often enhanced, and delays for garbage collection are unnoticeable, resulting in better interactive behavior.

A Note on Terminology

The popularity of lifetime-based garbage collectors since the publication of Lieberman and Hewitt's paper has led to a number of implementations, many of whose designers have invented their own terms to describe their work. In what follows, I usually use Moon's terminology where it is applicable, as this work was influenced most directly by his. In discussing the techniques of copying garbage collectors, I use the terminology of Fenichel and Yochelson [6]: memory in a copying garbage collector is divided into *semispaces*, only one of which is in use at any time (the *current* semispace), except during garbage collection, when both are used.

The names of LISP data types are used in discussions where they pertain; in particular, list cells are referred to as *cons* cells.

An understanding of traditional garbage collection techniques is assumed; in particular, the reader is assumed to understand the behavior of mark-sweep garbage collectors, and of Cheney's copying, compacting garbage collection algorithm [5]. This last will be referred to as copying garbage collection, or sometimes as *stop-and-copy* garbage collection. The term *root set* will be used to refer to the set of objects explicitly specified to the garbage collector for preservation; all objects preserved through a garbage collection are either in the root set or are encountered in some directed walk beginning at an object in the root set.

During a copying garbage collection, the space copied from is called *fromspace;* the space copied to is called *tospace,* or *copyspace. Scavenging* is the operation that copies objects referenced by a set of roots from fromspace to tospace, and updates in the root set the references to the copied objects. It may be used as a transitive verb; thus 'scavenging the stack' would mean finding the objects in fromspace pointed to by pointers in the stack, copying them and their descendants to tospace, and updating the pointers in the stack to point to the newly relocated objects.

Fromspace and tospace together are called *dynamic space*, as the objects in them are moved dynamically. In systems with lifetime-based garbage collectors, if there is a space where long-lived objects are maintained and are garbage-collected with a copying garbage collector that is not the lifetime-based garbage collector, this space is also called dynamic space, and the garbage collector is referred to as the *dynamic garbage collector.*

This paper assumes some knowledge of the behavior of virtual memory systems; in particular, an understanding of terms like *page*, *main memory*, and *backing store*. The term *dirty bit* is sometimes used; this refers to the information maintained on a per-page basis by virtual memory systems, stating whether the page in question has been modified (made "dirty") while it has been in main memory. The sections of main memory that hold individual virtual memory pages are called *page frames.*

In discussions where it is advantageous to consider particular architectures, I use as an example the Motorola MC68020, using the assembler syntax in [11]. My terminology differs from theirs only in that, when I refer to a *word,* I mean a 32-bit quantity; Motorola refer to these as *longwords.*

Where specific LISP tagging schemes are considered, I use that employed by Lucid Common LISP on the MC68020.

Acknowledgements

Much of the work described in this paper was performed at Lucid, Inc., of Menlo Park, California, in 1986 and 1987. It was strongly influenced by Moon's Ephemeral Garbage Collector for the Symbolics 3600 [10], but was independent of Shaw's work [12].

The author owes a debt of gratitude to the following people, all of Lucid, Inc.: Eric Benson, Dick Gabriel, Harlan Sexton, and Jon L. White, for many illuminating discussions during the design of the program; James Boyce, for his work in integrating the garbage collector into the LISP system and in ferreting out obscure bugs; and Leonard Zubkoff, for performance measurements on the Apollo. Prof. Rodney Brooks of M.I.T., and also of Lucid Inc., gave me the opportunity to work at Lucid and encouragement in my long struggle to buy a very expensive bachelor's degree.

Prof. Robert H. Halstead, Jr., was a most supportive thesis advisor, provided many useful comments during the writing of this document, and was a helpful and friendly presence through its writing.

The performance analyses of the garbage collector and the writing of this document were performed at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by Department of Defense Advanced Research Projects Agency under Office of Naval Research contract N00014-85-K-0124.

Part I

Motivation and Prior Work

1 Garbage Collection in Modern LISP Systems

1.1 Copying Garbage Collection

Fenichel and Yochelson [6] have described how performance will degrade over time in LISP systems utilizing virtual memory. Their solution, copying garbage collection, as further modified by Cheney [5], was widely adopted in modern LISP systems; but its performance was limited by the need to scan a potentially large root set, and to move from one area to the other, on each garbage collection, all the structures maintained through a computation. In a large LISP system running on a machine with virtual memory, garbage collections could result in quite lengthy pauses; enough so that White prescribes a scheme in which garbage collection is avoided altogether in virtual memory systems [14].

Certainly refinements are possible. One popular refinement used in many LISP systems [3] is to create in a "static" space those objects one knows will be relatively permanent, and to scan these along with the root set; then, while pointers in static objects to objects in dynamic space are still updated during a garbage collection, static objects (as their name implies) are not relocated, and the work of copying them is saved.

Another refinement uses an "unscanned" space,[1] in which permanent immutable objects that contain only pointers to the static or unscanned spaces are stored; because static objects are not copied by the garbage collector, the pointers to them need not change, and so unscanned space need not be scanned by the garbage collector; and, of course, the objects contained in it will not be relocated, so that, again, the work of copying them is saved.

[1] Brooks et al. [3] refer to this as a "read-only" space; however, to avoid confusion with, for example, pure shared pages, and concentrate exclusively on the garbage collection issue, I refer to it as unscanned.

There is something unsatisfactory about this sort of refinement, however. The garbage collector does not needlessly transport objects placed in static or unscanned spaces, or even examine those in unscanned space. But the storage they occupy can never be recovered, either, as these spaces are not, by their nature, garbage-collected. Thus, in order to decide whether to place an object in a static space, that is, a space whose contents are never relocated, the programmer must think about its lifetime; and this begins to smack of the sort of storage management details from which LISP purports to free programmers. Similarly, when deciding whether to place an object in a unscanned space, whose contents will not be scanned by the garbage collector, the programmer must decide whether it contains pointers to dynamic objects, and this, again, is the sort of task from which we would like to free the programmer.

Although we are unsatisfied with the additional tasks these refinements force on the programmer, we can see that they are motivated by a desire to perform two valuable optimizations: reduction of the size of the root set, and reduction of the number of times that relatively permanent objects are copied. We may understand lifetime-based garbage collection as an automation of these optimizations.

1.2 Lifetime-Based Garbage Collection

1.2.1 Lieberman and Hewitt's Garbage Collector

Baker [1] introduced in 1978 a copying garbage collection algorithm that operated in an incremental fashion: the work of transporting objects from one semispace to the other was interleaved with the normal object creation and manipulation functions of the LISP system.[2]

In practice, as implemented on the Symbolics 3600, Baker's garbage collector

[2]Incremental garbage collection has the advantage that there are no long pauses for garbage collection; however, I choose not to discuss it in any detail here, as its practical implementation requires the use of an architectural feature called the *invisible pointer* [8], which is not usually present on general-purpose machines. As will be obvious, however, Lieberman and Hewitt's methods have immediate application to stop-and-copy garbage collection, despite the fact that, as originally presented, they make yet another use of invisible pointers. This use, however, may be circumvented on general-purpose computers.

suffered from poor virtual memory performance (see note A.1). Lieberman and Hewitt [9] described a modification to Baker's algorithm, in which, rather than being divided into semispaces, memory was divided into many small sections called *regions*.

At any time there is a *current creation region*, in which new objects are allocated. When the current creation region is filled, a new, empty region is allocated to be the current creation region, and objects are created there, rather than in the old region.

Each region has a number, called a *generation number*. Generation numbers increase monotonically with time. Occasionally the garbage collector will be run on the contents of a region. When this happens, a new region is allocated; the live contents of the old region are copied into it, and the old region's storage is recycled. The old region's generation number is retained for the new region. Thus this scheme distinguishes between the number of garbage collections that an object may survive, and the object's chronological age; this is motivated because the garbage collector is run on regions individually, and thus there is not necessarily a relation between the two numbers.

In a traditional copying garbage collection scheme, the garbage collection of any of these regions would require scanning all others, both to discover which objects in the region were being referenced by objects outside the region, and thus needed to be preserved, and also to update the pointers from other regions to objects within the garbage-collected region, as these objects will be relocated during the garbage collection.

To avoid the necessity of scanning all other regions, Lieberman and Hewitt's garbage collector maintains for each region an *entry table*. The entry table is a table of pointers to objects in the region. Pointers from outside the region to objects inside it are made to point to entries in the entry table. Along with the stack and registers, the table is used as the root set when the region is garbage-collected; thus other regions need not be scanned.

As proposed for implementation on the MIT LISP Machine [8], the entry table was implemented using *invisible pointers*, an architectural feature which causes references to a location in memory to be transparently forwarded to another location. Thus no software overhead was incurred to check whether a pointer to an object actually pointed to an entry in an entry table. In practice, however, the lifetime-based garbage collectors discussed in the present text do not maintain entry tables; rather, they use structures that instead somehow record the locations of pointers outside a region to objects inside the region.

Lieberman and Hewitt's scheme called for making entry table entries only for objects pointed to from regions older than the region being garbage-collected; these were referred to as "pointers forwards in time." Objects pointed to only from regions younger than the region being garbage-collected, called "pointers backwards in time," were not entered in the entry table; and such regions were to be scanned as part of the root set during garbage collection. This optimization was motivated by the consideration that the majority of pointers would be pointers backwards in time, and that the regions most often garbage-collected would be the youngest; and thus there would be few regions younger than them to be scanned.

But what is it that makes this garbage collector lifetime-based? Lieberman and Hewitt observed that the mortality rate of older objects is much lower than that of younger objects. Thus garbage collections spaced at set intervals will likely reclaim less space from regions containing older objects than from regions containing younger objects, so that more storage will be reclaimed per unit of garbage collector work by garbage-collecting younger regions more often than older regions.

It should be noted, then, that lifetime-based garbage collection affords two separate optimizations. By dividing memory into regions with recording structures specifying the objects that point to them, they limit the size of the root set. By allowing objects to be segregated according to age, and garbage-collecting areas those containing relatively permanent objects less often, they limit the amount of copying that must be done in a garbage collection.

1.2.2 Address Space Utilization and Physical Memory Utilization in Lifetime-based Garbage Collection

Lieberman and Hewitt's scheme allows for utilization during user computations of a higher percentage of the available address space than does simple stop-and-copy garbage collection. This is because not all regions are garbage-collected simultaneously, and only as much storage as will be needed to hold the data in the region or regions actually currently being garbage-collected must be maintained free. In simple stop-and-copy garbage collection (or in Baker's incremental garbage collection), however, a semispace's worth of free storage must be maintained at all times; otherwise a flip may fail.

The traditional wisdom about copying garbage collectors, as first advanced by Fenichel and Yochelson in [6], is that address space utilization is of no great importance; garbage collections are performed to improve locality of reference, and address space recycling is only a secondary concern. In lifetime-based garbage collectors, however, the frequency of garbage collections of younger levels is such that both the space being copied from and the space being copied to must be considered part of the working set. We expect, then, that the best results will be attained by memory layouts that make the most efficient possible use of the address space occupied by the most frequently garbage-collected levels.

Lieberman and Hewitt's garbage collector does allow for utilization of a greater portion of the address space during user computations than would a scheme that statically maintained semispaces for each generation, because the space occupied by a region just copied from may be re-utilized in the copying of another region. However, because the space occupied by the youngest objects changes with each garbage collection of the youngest generation, the virtual memory performance of this system will not be all we might hope for.

2 Previous Implementations

Lieberman and Hewitt's work has provided a basis for several lifetime-based garbage collectors. I discuss here three garbage collectors examined during the design of Lucid's lifetime-based garbage collector; these are: Ungar's generation-scavenging garbage collector [13], the Tektronix Large Object Space Smalltalk garbage collector [4], and Moon's ephemeral garbage collector for the Symbolics 3600 [10]. The points considered in each case are: division of memory, determination of the root set, and advancement policy.

2.1 Ungar's Generation-Scavenging Garbage Collector

2.1.1 Description

Ungar's Berkeley Smalltalk system divides objects into two classes: new and old. The space that old objects live in is called OldSpace. These old objects are garbage-collected offline; which is to say, not at all during a Smalltalk session. There are three spaces for new objects: NewSpace, where new objects are created, PastSurvivorSpace, where new objects are also stored, but are never created, and FutureSurvivorSpace, which is the space into which PastSurvivorSpace and NewSpace are copied.

New objects are always created in NewSpace. Every time a pointer is set, if it is a pointer from Oldspace to NewSpace or PastSurvivorSpace, the location in which the pointer was set, that is, the referring object, is recorded in a table, called the *remembered set.* When NewSpace is full, a copying garbage collection is performed from NewSpace and PastSurvivorSpace to FutureSurvivorSpace, using as the root set the references to NewSpace and PastSurvivorSpace from the objects in the remembered set. FutureSurvivorSpace and PastSurvivorSpace are then exchanged.

Each object has associated with it a generation count; when an object has survived a certain number of garbage collections, it is copied into OldSpace, rather than PastSurvivorSpace.

2.1.2 Address Space Utilization

Ungar's generation-scavenging garbage collector, like that of Ballard and Shirron [2], on which it is based, is capable of utilizing more than half of its address space at any time, as only FutureSurvivorSpace remains unused during a computation. Because the two survivor spaces are much smaller than either NewSpace or OldSpace, a large percentage of the address space is in use during user computations.

2.1.3 Suitability

There is an obvious problem with the use of an approach that requires the storage of generation counts. LISP systems on general-purpose computers usually have highly-optimized storage formats; thus a cons, for example, will consist of exactly two pointers, with no space for a generation count. This is not nearly such a problem in Smalltalk systems, where dynamic objects are usually vector-structured, with object references to the object's class, instance variables, etc.; but in high-performance LISP systems we would probably have to store object generation counts externally.

Suppose we were to store four bits of age information for each object. In order to memory-map these, we would need to allocate four bits of storage for every sixty-four bits stored in the ephemeral spaces, as this is the greatest common denominator of LISP object sizes on thirty-two bit machines. This gives some 6.25% additional storage required, in addition to that required for structures used to record the remembered set. We should like to avoid this overhead if at all possible.

2.2 The Tektronix Large Object Space Smalltalk Garbage Collector

2.2.1 Description

The Tektronix Large Object Space Smalltalk implementation [4] includes a generation-scavenging garbage collector that differs from Ungar's mainly in the organization of memory. Memory is divided into seven regions, each of which consists of two semispaces. Objects still include generation counts; during a garbage collection, they are copied from one semispace to the other, and their generation counts are incremented. When an object's generation count reaches some preset value, it is advanced to the next region.

Garbage collection is stop-and-copy, motivated by the filling of a region. Stores to the stack are not recorded; rather, the stack is scanned on each garbage collection. Remembered set tables, as used in Ungar's Berkeley Smalltalk garbage collector, are maintained for each region, in order to limit the size of the root set. These tables contain only pointers forwards in time; that is, pointers from older objects to newer objects. As in Lieberman and Hewitt's garbage collector, the entirety of each younger region is used as part of the root set whenever an older region is garbage-collected.

2.2.2 Address Space Utilization

As in Cheney or Baker's garbage collectors, during user computations the Tektronix garbage collector uses half of its address space for empty semispaces.

2.2.3 Suitability

We anticipate problems with using in LISP systems a garbage collection scheme like that used in Tektronix's Large Object Space Smalltalk. One problem is that of recording generation counts; we discussed this in considering Ungar's generation-scavenging garbage collector, in Section 2.1.3, above. Furthermore, as discussed in Section 1.2.2 above, the organization of memory into semispaces should result in poorer virtual memory performance than we might hope for.

2.3 The Symbolics Ephemeral Garbage Collector

2.3.1 Description

The Symbolics ephemeral garbage collector uses an allocation scheme different from that of either the Berkeley Smalltalk or the Tektronix Smalltalk garbage collectors. Memory is divided into areas. The user may specify for each area the number of ephemeral levels, the capacity of the youngest level, and the ratio of the capacity of each succeeding level to that of the youngest level.

Ephemeral levels are not divided into semispaces; rather, when a level is garbage-collected, live objects within it are copied into the next older level. As in Lieberman and Hewitt's garbage collection algorithm [9], ephemeral garbage collection is incremental, and proceeds in parallel with user computation. Ephemeral levels are garbage-collected independently of each other; pointers 'backwards in time' (see Section 1.2.1) are recorded by the same means as are other pointers.

The capacity of an ephemeral level is the number of words that may be allocated in it before a garbage collection is initiated. A garbage collection is motivated when the youngest level's capacity is exceeded; this causes its contents to be garbage-collected into the next oldest level. If that level's capacity is exceeded, its contents are copied into the following level; objects that survive through all the levels are advanced to dynamic space. Storage in dynamic space is recovered with a separate Baker-style incremental garbage collector.

The scheme used for recording the root set is different from those of either of the Smalltalk implementations; instead of maintaining remembered sets (or entry tables in the sense of Liebermann and Hewitt's original paper, which the 3600 could implement because of its invisible pointer hardware), the Symbolics scheme instead maintains one mark bit per page for each ephemeral level. It detects when a pointer into an ephemeral level is being stored, and marks the page in which the reference occurred. When an ephemeral level is scavenged, marked pages are scanned for references to that level, and such references, if found, are used as roots.

Note that the mark bits resemble the dirty bits maintained by virtual memory systems; but, unlike dirty bits, they are only set when the word stored is actually a pointer to an ephemeral level. No great harm is done if the word stored was not a pointer to an ephemeral level, though, and sometimes at scavenging time the mark bit will be set for a page that contains no references to ephemeral objects, as a word that is not a pointer to an ephemeral object may overwrite all pointers to ephemeral objects on a page without causing the mark bit to be reset. Shaw [12] has exploited this similarity to a table of dirty bits in his lifetime-based garbage collector.

Scanning the entirety of each marked page may sound wasteful, but the 3600 has hardware to assist in the detection of references to ephemeral levels, and so a 256-word page with no such references is scanned in 85 microseconds [10, page 242].

The scheme used in the Symbolics ephemeral garbage collector for recording pointers into ephemeral spaces is in fact more complex than is implied by the short description above. Two tables are actually maintained; one, called the Garbage Collector Page Tags (GCPT), holds a bit for each of the pages present in physical memory. Only one bit is stored, and thus no information about which ephemeral level the pointers in the page may point to is recorded; the bit says only that a pointer to an ephemeral level may still be present on the page.

Another table, called the Ephemeral Space Reference Table (ESRT), is stored sparsely, and contains for each swapped-out page a bit for each ephemeral level; the bit is set to indicate that the page contains a reference to the level in question. The table is maintained entirely in physical memory, and allows the garbage collector to determine as it garbage-collects a level whether to scan a page; because scanning a page requires fetching it from backing store, the maintenance of per-level information saves page faults. Clearly, the ESRT must be updated whenever a page is ejected from physical memory.

Given that the intention is to avoid any needless scanning of pages residing on backing store, how must the ESRT be updated when pages are ejected from physical memory? If the page has not been written at all, its ESRT entry (if extant) need not be updated. If the page has been written, and already has an ESRT entry, then the ESRT entry must be updated regardless of the setting of the page's GCPT bit, because it is possible that data written into it overwrote all the page's pointers to ephemeral objects. If the page has no ESRT entry, then it need only be scanned if its GCPT entry is set; only then might one have to create an ESRT entry for it.

2.3.2 Implementing the Symbolics Ephemeral Garbage Collector Without Special-Purpose Hardware

Suppose one wished to implement a garbage collector similar to the Symbolics Ephmeral Garbage Collector on a general-purpose computer in which one had the cooperation of the virtual memory system. It will be immediately apparent that the technique of scanning pages makes use of the 3600's fully tagged architecture. Because each word has a tag that tells whether it is a pointer, one may begin scanning memory in the midst of an object, without risk of interpreting non-pointer data (such as, say, collections of characters in a string) as pointers.

On a general-purpose machine, one would need to use some other method of determining whether words examined were pointers. One possibility is to divide all data into two types: those containing pointers and those not containing pointers. This is certainly possible using the data formats in (for example) Lucid Common LISP, but there is a price to be paid. Storage management is complicated by the addition of a parallel set of storage spaces. Locality of reference is degraded.

Another possibility is to maintain in a table an entry for each page giving the offset from the base of the page to its first tagged word. A slight amount of overhead at object creation time suffices to maintain the table entries. In the data formats used by Lucid Common LISP, and a number of other modern LISP implementations for general-purpose computers, untagged words never follow tagged words without object headers appearing in between, because otherwise the linear scan of a space used as the root set in copying garbage collection would be impossible. The overhead for maintaining a memory-mapped table of such offsets on a 32-bit computer with 256-word pages is 8 bits per 256 words, or about 0.1%; this is certainly not a deleterious amount of storage.

We have surmounted, then, the problems for garbage collection associated with maintaining untagged words in the system; what about maintaining the GCPT? As the discussion above suggests, rather than maintaining a GCPT, we may simply use the table of dirty bits in the virtual memory system. When it is time to perform a garbage collection, we scan all the pages whose dirty bits are set; because these pages are necessarily all in physical memory, this operation should not take long compared to the time required to fetch pages from backing store. Exactly how long might it take to scan a page? Figure 1, on page 13, gives the MC68020 code and timing to scan a 256-word page.

The common path through the loop, in the cache case, is some 51 cycles; on an MC68020 clocked at 16 megahertz, this corresponds to some 816 microseconds for a 256-word page. The immediate tagged case is not included; it would involve checking the high five bits of the low byte to determine the object type. The immediate tagged case will often (in the case of vectors of untagged data, for example) lead to the skipping of some number of words of untagged data. This, and the fact that it is reasonable to expect that the loop will reside entirely in the MC68020's 256-byte on-chip instruction cache, give us some confidence that 816 microseconds is a conservative estimate.

Here is where special-purpose hardware comes into its own, then; the MC68020 takes nearly ten times as long to perform a page scan as does the 3600. How significant is this page-scanning time? The difference between the two is some 731 microseconds, but this does not tell nearly the whole story, as the 3600's GCPT bits are unlike dirty bits in that they are only set when the word stored is actually a pointer to an ephemeral level. A general-purpose computer does not perform tag or boundary checks, so many pages in the LISP process's address space will have their dirty bits set without pointers to ephemeral levels actually having been stored in them. These pages will be scanned needlessly at every garbage collection (see note A.2 for a comparison with the scheme described by Shaw [12]).

Metering would allow us to determine how many pages would actually be found needlessly dirty in typical LISP applications; however, even without such measurements, we can envision a worst case. Let us consider the performance of the Symbolics ephemeral garbage collector on a 3600 with 1 megaword of physical memory [10, table 2, page 244]. When running the BOYER benchmark, an average of 11.5 seconds were taken for each flip; when running the compiler benchmark, the figure was 1.6 seconds.

```
;;; First tagged word is in a0; segment table in a1, word
;;; after last on page in a2.  Parenthesized numbers after
;;; instructions give best case, cache case, and worst case
;;; timings, as per MC68020 User's Manual.
scanlp: cmp.l a0, a2            ;done yet? (1 4 4)
        ;; branch when done scanning.  ge because untagged
        ;; structure may cross page boundary.
        bge scandn              ;(1 4 5) (branch not taken)
        move.l (a0)+, d0        ;d0 holds pointer (4 6 7)
        ;; fetch tag bits; bashes low byte
        andi.b #$07, d0         ;(0 2 3) + (0 2 3)
        ;; see if this might be a header
        cmpi.b #dtp-other-immediate, d0
                                ;(0 2 3) + (0 2 3)
        ;; handle if so; this is also the case for tagged
        ;; characters and small floats (i.e., not in strings
        ;; or vectors)
        beq check-header        ;(1 4 5) (branch not taken)
        ;; now want only low two bits of tag
        andi.b #$03, d0         ;(0 2 3) + (0 2 3)
        ;; low bits are 0 if fixnum
        beq scanlp              ;(1 4 5) (branch not taken)
        ;; if we got this far, this is a pointer.  Fetch
        ;; segment index.  Note high part still valid.
        swap d0                 ;(1 4 4)
        ;; does the pointer point into an ephemeral segment?
        cmpi.b #ephemeral-segment, (d0, a1)
                                ;(0 2 3) + (3 5 6)
        ;; no, continue
        bne scanlp              ;(3 6 9) (branch taken)

                Best Case       Cache Case      Worst Case
Totals:            15              51              66
```

Figure 1: Page-scanning on the MC68020.

On our general-purpose computer, if every one of 1024 pages were found dirty at garbage collection time, but contained no ephemeral references, then there might be some 2.9 seconds of overhead per garbage collection. This is 25.2% of the BOYER benchmark time, and 153% of the compiler benchmark time. I stress that this is the worst possible case, with every page in physical memory found dirty at garbage collection time. We expect that in real applications a much smaller proportion of the pages would be found dirty (many of them, for example, might contain pure code), and, of those found dirty, many would not need to be scanned (for example, those in the youngest ephemeral level, or in the unscanned spaces).

Consider now the time required to update the ESRT. On our general-purpose computer, we use the virtual memory system's table of dirty bits for the GCPT, so that, if we wish to ensure that ESRT entries are always correct for pages on backing store, we must scan the page and potentially update or create an ESRT entry whenever the ejected page's dirty bit is set. The case where the 3600's special-purpose hardware will help it is that where a word was written into a page, but the word was not a pointer to an object in an ephemeral space. But here we assume that the virtual memory system on our machine has given the garbage collector a trap after initiating a seek on the disk; the scan time is easily entirely subsumed in the average seek time of any modern disk drive, just as on the 3600.

Thus the 3600's special-purpose hardware does indeed help it, but not nearly so much as one might think. Given our belief that the average overhead for page-scanning on stock hardware will not account for most of the time spent in garbage collection, we believe that with a slightly faster processor, such as an MC68020 clocked at 25 megahertz, the 3600's speed advantage will disappear. Note that the time estimates given are for time spent in the garbage collector; the overhead required at user program run time to keep track of ephemeral objects, estimated by Moon as 'at least 10% and possibly a factor of two or more,' [10, page 243] is actually limited to the nearly insignificant time added to object creation in order to update the table of offsets to the first tagged object in each page – provided one has the collusion of the virtual memory system.

2.3.3 Address Space Utilization

Both the Symbolics Ephemeral Garbage Collector and the stock-hardware derivative we propose here copy objects from one ephemeral level to the next; thus the pages used for the creation of new objects are the same before and after a garbage collection. We expect good virtual memory performance from this scheme.

2.3.4 Suitability

Moon's ephemeral garbage collector does not lend itself to easy criticism. While his paper does not anticipate the implementation of his scheme on general-purpose computers, and, indeed, discounts the idea, the author feels that, given the cooperation of the virtual memory system, Moon's garbage collection scheme would perform quite well on a general-purpose computer. General-purpose computers are at a disadvantage in the task of page-scanning, but even inefficient scanning of pages is a small price to pay to escape the need to explicitly record using additional software the storage of pointers into ephemeral levels.

Part II

A Lifetime-based Garbage Collector for LISP Systems on General-Purpose Computers

In what follows, I use Moon's terminology. I call objects that have not yet been moved to spaces intended to hold relatively permanent objects *ephemeral* objects. Moving an object from a space that holds newer objects to a space that holds older objects is called *advancement.* The several spaces that hold ephemeral objects, organized so that the ages of the objects within them vary monotonically, are referred to as *ephemeral levels.* The ephemeral level holding the youngest objects is called either the youngest ephemeral level or the first ephemeral level; that holding the oldest objects is called either the oldest ephemeral level or the last ephemeral level.

3 Desiderata

A number of constraints are forced by the use of general-purpose computers, and by the desire to write a portable garbage collector; that is, one that does not require the cooperation of the virtual memory system. A summary of the design goals follows:

- Performance. We wanted each garbage collection to be fast; that is, we wanted to minimize the amount of computation required for a garbage collection, so that the ephemeral garbage collector would be suitable for use in interactive systems.

  We also hoped not to unduly slow down user code. Because we do not have the cooperation of the virtual memory system, pointer-settings must be recorded explicitly, so that some slowdown is inevitable in execution time of code that does not allocate storage, and thus does not cause garbage collections. We wished to minimize this slowdown.

In code that creates and discards many objects, like the BOYER benchmark [7], we wished to realize overall performance improvements.

- Portability. The design was not to require cooperation from the virtual memory system, nor was it to be tied to a particular architecture. It also had to be easy to retrofit into existing LISP implementations for various machines.

- Predictability. Many users of LISP carefully code their programs to avoid any object creation, so that no unexpected delays will occur; for example, a robot control program cannot afford even a 20-millisecond delay. Programs that do not create objects should not cause garbage collections, or be subjected to unexpected delays, as for reorganization of internal tables.

- Flexibility. We wanted the ephemeral garbage collector to be tunable; the number of levels and their sizes were to be easily modifiable, because the parameter settings for best performance were likely to vary between applications.

- Robustness. The scheme we selected was not to be prone to failure during garbage collection. We wished to avoid schemes that could conceivably run out of memory when advancing objects from one ephemeral level to the next.

4 Early Decisions

With the goals discussed in the previous section in mind, it was possible to make many design decisions before committing to a specific scheme for recording pointers. These decisions are discussed below.

4.1 Optimizing the Task of Keeping Track of Ephemeral Objects

Without special support from the virtual memory system, the greatest source of inefficiency in lifetime-based garbage collection systems on general-purpose computers is the recording of pointers into ephemeral spaces. This recording must be performed in software, replacing what was formerly one instruction; this increases the size of the compiled code image, even if an out-of-line call is performed, and has a varying, but always negative, effect upon performance, dependent upon the dynamic frequency of pointer stores.

This effect is manifested most strongly in high-performance systems with native code compilation; it is not nearly so much a problem in, for example, Ungar's Berkeley Smalltalk system (discussed in Section 2.1, above), because this system utilizes a byte-code interpreter that executes only some 9,000 instructions per second. In [10, page 246], Moon makes the point that the performance of Ungar's generation-scavenging looked good because Berkeley Smalltalk takes about 50 machine instructions to do a store; the overhead of adding an object to the remembered set is not overwhelming by comparison.

One somewhat ameliorating factor is the possibility of performing certain compile-time optimizations; as noted below, the Lucid Common LISP compiler does in fact perform these optimizations for the benefit of the lifetime-based garbage collection scheme we implemented. The compiler optimizes out pointer-recording when the pointer being stored is a constant immediate quantity, such as a character or small integer, or points to a constant static entity, such as a symbol.

One more significant optimization is performed; when the current object creation area is the youngest ephemeral level, the object-creation subroutines used do not record storage of initial values in newly-created objects, as these will have been created in the youngest ephemeral level, and any pointers from them will be pointers 'backwards in time,' to use Lieberman and Hewitt's terminology.

4.2 Set-Associative Pointer-Recording

One possible pointer-recording scheme would use a set-associative table of locations holding pointers to ephemeral spaces.

Suppose we maintained a set-associative table of 256 lines, with 16 one-word entries per line, for each ephemeral level; this gives some 16 kilobytes of table per ephemeral level. Assume further that we worried only about updating the table for pointer settings; that we did not worry about removing entries in the table for pointers that were written over. When any line in the table was completely full, we could use one of several (expensive) strategies to reduce the problem; possibly we could begin allocating in the ephemeral level in question a list of pointers into it, or perform a scavenge of the level in question into the next older level, in which case we would need to add to that level's table only the references that were not in that level, and we would likely have enough space for them; etc. But leaving aside the question of dealing with entirely full lines, we depict in Figure 2 an MC68020 instruction sequence that performs a pointer-setting when using a set-associative table for recording pointers into ephemeral levels.[3]

The common case (set a pointer from the current ephemeral level to the current ephemeral level) requires nine instructions, not counting the pointer setting. Making the table entry requires ten instructions in the case where the first entry examined is empty, and five more instructions per time around the loop. This sequence of instructions is so large as to mandate an out-of-line subroutine call; this would entail further overhead at runtime. Note also that we are making free use of two address registers beyond those holding the pointer and the destination, and three data registers; thus the compiler will have fewer registers at its disposal, with the attendant negative impact on efficiency of surrounding code.

[3] We do not move using the MC68020 memory indirect post-indexed addressing mode (although it would save one instruction), as it is slower than the combinations of instructions we do use.

```
        ;; on entry, a0 holds the destination, and a1 the
        ;; pointer.  Fetch byte table of ephemeral levels
        ;; from systemic quantities vector (SQ), which is
        ;; held in an address register.
        move.l (elevel, SQ), a2
        ;; prepare to fetch segment index of destination
        move.l a0, d0
        swap d0         ;get segment index in low 16
        ;; store in d1 ephemeral level of destination
        move.b (0, a2, d0.w), d1
        move.l a1, d0
        swap d0
        ;; d2 gets ephemeral level of pointer
        move.b (0, a2, d0.w), d2
        ;; now we compare ephemeral levels to see if we need
        ;; to make a recording table entry.
        cmp.b d1, d2
        bne hktabl      ;different levels, make entry
        ;; same, just set pointer (most common case)
        (set pointer and exit)
        ...
        ;; fetch table of tables from SQ
hktabl: move.l (extbls, SQ), a2
        ;; index by ephemeral level of pointer
        move.l (0, a2, d2.w), a2
        move.l a0, d0   ;get dest in data register again
        ;; lines are at 64-byte intervals; want bits 14-0
        ;; with low 6 cleared for index of our line.  Get
        ;; line index in d0.
        ori.l #$00003FC0, d0
        add.l d0, a2    ;line address in a2
        ;; 16 entries per line, but testing at bottom
        move.l #15, d1
        ;; find entry that's empty or same
loop:   move.l (a2)+, a3
        beq found       ;0 is empty entry
        cmp.l a0, a3    ;if same, done, go to setit
        beq setit
        dbra d1, loop
        ;; no empty entry in this line
        (deal with the problem, possibly by creating an
        extension to the line)
        ...
found:  move.l a0, -(a2)    ;make entry
        bra setit       ;go set pointer
```

Figure 2: MC68020 code to record ephemeral reference locations in a set-associative table.

This scheme does have the advantage of compactness of representation of the recorded information, but has two major disadvantages. The first is that the number of instructions executed to set, for example, a special variable, is at least nineteen, and likely more. The second disadvantage is that it is not clear how to proceed when a line is filled; the delays necessary for compaction or garbage collection might be too large. Two design goals, performance and predictability, are violated; the scheme was not considered further.

4.3 Avoiding the Overhead of Determining Spaces When Storing Pointers

We noted that set-associative pointer-recording had two distinct disadvantages; the second had to do exclusively with the structures used for recording the storage of pointers into ephemeral levels, but the first disadvantage lay partly in the expense of that recording, and partly in the expense of determining the spaces for a pointer being stored and the location it is stored in.

On the 3600, when a word is stored into memory, it is examined (in parallel with the memory access) to see if it is a reference to an ephemeral area being stored into either another ephemeral area, or into a non-ephemeral area; if it is, the fact is recorded by setting a bit in the GCPT. Moon states that the reason custom hardware is required to implement a lifetime-based garbage collector is that this examination would have to be performed in software on a general-purpose machine, and would take between 2.5 and 25 microseconds. As we saw in Figure 2, which showed MC68020 code for maintaining set-associative tables of pointers into ephemeral spaces (the determination of spaces would be the same), it would also require the use of several registers, thus slowing down execution of the surrounding code. Finally, the nine instructions required simply to determine the ephemeral levels of the pointer and the location in which it is stored, before ever recording its storage, would have a serious impact upon performance of code that did not garbage collect.

What we wish to do is move some of the overhead of this operation from pointer-setting time to garbage collection time. The critical portions of the garbage collector can be coded in assembly language, and can use as many registers as necessary; there will be only one copy of this code, so the number of instructions used will not be critical to the image size, as it would be if the operation were being coded in-line at every pointer storage. Two other important points are:

- Many of the pointers stored during the execution of a program will be stored in locations in the youngest ephemeral level. The garbage collector will never have to determine the ephemeral level to which these pointers point, because all pointers stored in the youngest level are pointers backwards in time, in the terminology of Lieberman and Hewitt.
- In many programs, a single set of locations is repeatedly written. If there are several pointer stores to a single location between garbage collections, the ephemeral level of only the last pointer stored matters; thus the work done to determine ephemeral levels in the other pointer stores is wasted.

These considerations suggest that, rather than determining the spaces of the pointer and the location it is stored in at pointer-storage time, we should adopt some sort of scheme whereby we record only that a location has been modified, and postpone until garbage-collection time the determination of the space the pointer was stored in and the space it pointed into. We do not even attempt to determine at runtime whether the pointer is actually an immediate constant, such as a character or fixnum.[4]

In using this scheme, we will often have to examine at garbage-collection time locations that do not contain ephemeral references at all. This examination will cost very little if the page containing the location in question is in main memory; if it is on backing store, the cost will be much greater. Our hope is that the technique of lifetime-based garbage collection so improves locality of reference as to decrease substantially the number of cases where the working set exceeds the available physical memory.[5]

[4] Although, as mentioned in Section 4.1 above, compile-time optimizations may be exploited.

[5] Measurements of the number of pointer stores recorded, but not containing ephemeral references, during the execution of various symbolic processing tasks would be useful in evaluating this scheme, as would a measurement of the number of pages containing

4.4 The Organization of Ephemeral Spaces in Memory

We concluded that the maintenance of generation counts was undesirable in LISP systems.[6] Without generation counts, we have two alternatives when performing lifetime-based garbage collection:

- We can organize each ephemeral level into semispaces, and copy objects from one semispace to another until a certain number of garbage collections have been completed. This gives address space utilization much like that of Tektronix's Large Object Space Smalltalk garbage collector.
- We can garbage-collect each ephemeral level into the next older level, as in the Symbolics ephemeral garbage collector.

We concluded that the first alternative is more likely to result in poor virtual memory performance than the second;[7] thus we chose to use a division of memory into successive ephemeral levels, each of which is garbage-collected into the next older level.

4.4.1 Pointers Backwards in Time

Lieberman and Hewitt's garbage collector (discussed in Section 1.2.1) recorded only 'pointers forwards in time,' that is, those from either non-ephemeral spaces to ephemeral spaces or from older ephemeral levels to younger ephemeral levels. Thus the garbage collection of a level required scanning all younger levels as members of the root set; otherwise pointers 'backwards in time,' from younger levels to older levels, would not be updated to point to the newly copied referents, and might possibly lose their referents altogether.

Moon's ephemeral garbage collector is also incremental, and can scavenge several levels at once while running user code. His solution is to record

recorded locations ejected to backing store between ephemeral garbage collections on systems with varying amounts of memory. These and other recommendations for future analysis are described in Section 8.

[6] See Section 2.1.3.

[7] See Sections 1.2.2 and 2.2.2.

pointers backwards in time by the same means as pointers forwards in time; because his representation of recording information is very compact, this is inexpensive.

In a stop-and-copy scheme, user code may not run until a garbage collection has completed. When we have finished copying from a younger level to an older level, the younger level will be empty. If the older level is now full, we may scavenge it without scanning younger levels, as these will be empty.

4.4.2 Implications for Memory Organization

This ordering of events allows us to organize our ephemeral spaces in a convenient fashion. Many operating systems on general-purpose computers require contiguous memory allocation; thus, if garbage-collecting an ephemeral level would require filling the next level beyond its capacity, space must be set aside for the overflow. Suppose we order our ephemeral spaces as depicted in Figure layout.[8] The size of the odd-level overflow segment pool (OSP) is the sum of the sizes of all the even levels except for the last ephemeral level, if it is an even level; similarly, the size of the even level OSP is the sum of the sizes of all the odd levels except for the last ephemeral level, if it is an odd level. Thus the total space occupied by overflow segments is less than the space occupied by ephemeral data.

Suppose we perform a copying garbage collection from level 0 to level 1, and level 1 is full, and all data in level 0 are retained; we may allow data to overflow from level 1 into the odd-level overflow segment pool. Because the size of the odd-level OSP is at least that of the level 0, we are guaranteed that there will be room to copy the data from level 0 into level 1 and the odd-level OSP, so our copying garbage collector may simply continue copying past the end of level 1 into the OSP. When we garbage-collect level 1 into level 2, level 1 may contain as much data as level 1 and level 0 put together. If level 2 were also entirely filled, we are guaranteed room to complete the garbage collection, because level 1's size was included in the size of the even-level OSP, and level 0 is now empty, so we simply continue copying past the end of level 1 into level 0.

[8] I am indebted to James Boyce, of Lucid, Inc., for this suggestion.

```
+-------------+-------+-------+----------------------+
| other even- |       |       | even-level overflow  |
| numbered    | level | level | segment pool         |
| ephemeral   |   2   |   0   | Size = sum of sizes  |
| levels      |       |       | of all odd levels    |
|             |       |       | except last, if odd. |
+-------------+-------+-------+----------------------+
 low addresses                      high addresses

+-------------+-------+-------+----------------------+
| other odd-  |       |       | odd-level overflow   |
| numbered    | level | level | segment pool         |
| ephemeral   |   3   |   1   | Size = sum of sizes  |
| levels      |       |       | of all even levels   |
|             |       |       | except last, if even.|
+-------------+-------+-------+----------------------+
 low addresses                      high addresses
```

Figure 3: Layout of ephemeral spaces and overflow segment pools in Lucid Common LISP.

Note that this strategy has good implications for virtual memory usage. The pages overflowed into are pages that recently held other ephemeral data; thus we have a good chance that they will still be present in physical memory.

The garbage collection strategy described above, in which no ephemeral level is garbage-collected until all younger levels have been emptied, has a number of implications. First, note that, occasionally, a garbage collection will require garbage-collecting all the ephemeral levels. The defaults on the 3600 are to use semispaces that decrease in size by a factor of two; this may or may not reflect something like the "average" persistence of objects, but with this progression of sizes, the maximum amount of work required to garbage collect m levels would be no more than $2m$ times the amount of work to garbage-collect level 0 (for $m = 4$ levels the factor is actually about 6).[9]

For five levels, we approach an order of magnitude; one risk in emptying younger levels before garbage-collecting older levels, then, is that the pauses for garbage collection may become noticeable. In practice, this has not been a problem. A more serious risk lies in the way that a garbage collection that causes the objects in ephemeral space to be advanced all the way to dynamic space (as some garbage collections inevitably do) cannot help but advance, at the same time, all the live objects that were in the youngest ephemeral level at the time of the garbage collection. These objects may in fact become garbage very soon after their advancement; they have had less than one garbage-collection period in which to mature before being advanced to dynamic space. However, garbage collections that proceed all the way to dynamic space are much more rare than those that do not; we do not expect this "premature tenuring" (to use Ungar's term) to be a problem.

4.5 Allocation of Very Large Objects

Most very large objects have long lifetimes. These objects may be, for example, bitmaps being processed by image understanding programs, or arrays of cellular automata, or data collection buffers for input devices. Sometimes these objects are so large that they exceed the capacity of the youngest ephemeral level, which is typically a small fraction of the space

[9] See note A.3 for a precise formulation.

allocated to the process; then they cannot be created in ephemeral space at all. Some other very large objects will fit in the youngest ephemeral level, but, because they are permanent, will simply be copied on succeeding garbage collections through all the ephemeral levels until they are finally advanced to dynamic space.

This sort of successive copying is inefficient; we would like to provide the user with a means of causing objects larger than a given size to be created in dynamic space. This is easily accomplished with a user-modifiable parameter that is checked by the object creation routines.

Allocating objects in dynamic space when the normal allocation space is ephemeral space will eventually result in the filling of dynamic space, and the attendant dynamic garbage collection; and there will be objects in ephemeral space at this point. In fact, this situation is not unique to the allocation of very large objects; whenever we are about to scavenge the oldest ephemeral level into dynamic space, we must insure that there is enough room in dynamic space to hold its contents. The allocation of this space may result in a dynamic garbage collection, and there will be live data in ephemeral space during the dynamic garbage collection.

4.6 Dynamic Garbage Collection in the Presence of Ephemeral Objects

When garbage-collecting dynamic space, if ephemeral space holds live objects, we must somehow arrange for these objects to have their references to objects in dynamic space updated; possibly this could be done by garbage-collecting the ephemeral spaces as well. Clearly we can not proceed as usual, and simply not move ephemeral data when we encountered it while walking the tree from the roots; the Cheney algorithm uses copying and reordering in order to record pending branches, in the same way that other algorithms use a stack, or pointer reversal. We also do not want to use these other algorithms; the use of a stack has the well-known problem of deep nesting causing overflow, and Deutsch-Schorr-Waite pointer reversal requires visiting twice each node encountered.

One way of approaching the problem is to subdivide it into two cases: one may either leave the ephemeral data in ephemeral space, or move them all into dynamic space.

1. The obvious way to proceed if we wish to leave the data in ephemeral space is to treat ephemeral space as roots for the dynamic garbage collection. This has the disadvantage of quite possibly causing preservation of structures that are only pointed to by garbage in ephemeral space, but we expect that these will be rare.

   The algorithm is slightly complicated by the necessity of updating pointer storage recording structures. We do this by first clearing the structures recording pointers for the locations in dynamic fromspace. As scavenging is performed, when a pointer is stored in dynamic tospace, if it points into ephemeral space, it is recorded in the proper structure.

2. The other possibility, that of moving all the data into dynamic space, causes premature tenuring, to use Ungar's term, but has the advantage of being simpler. We simply treat data in ephemeral space in the same way we treat data in oldspace; we copy them all into newspace. At the end of the garbage collection, ephemeral space is empty, and all recording structures are cleared.

The problem with scheme 2 is that, although it is indeed simpler than scheme 1, there is no obvious way to proceed if one exceeds the capacity of tospace during the garbage collection and virtual memory is exhausted. Exceeding the capacity of tospace is indeed possible, as the entire live contents of ephemeral space are being added at once to dynamic space.

If we instead treat the ephemeral spaces as roots, we may perform the dynamic garbage collection without advancing ephemeral data. If we were in the midst of performing an ephemeral garbage collection, and the dynamic garbage collection freed enough space to allow the advancement of ephemeral objects into dynamic space, we would simply continue. If we found that an amount of space insufficient to allow advancement of ephemeral data was freed, we might disable dynamic garbage collection, copy the ephemeral data into the unused semispace, and signal an error to the user, who could respond by, for example, suspending the LISP process until more virtual memory is available.

The ability to continue in this fashion requires that the sum of the sizes of the ephemeral levels be no larger than a dynamic semispace.

We cannot use the same sequence of operations with scheme 2, because we cannot perform the dynamic garbage collection without also beginning to copy in the ephemeral data, and, at that point, we no longer have the option of using the other semispace if we should run out of space; we are already using both semispaces.

Thus we chose to use scheme 1 in our garbage collector.[10]

[10] For simplicity, the pointer storage recording algorithms given in sections 5 and 6 do not show the cases in which copying into dynamic space requires dynamic garbage collection, and thus do not depict the necessity of examining recording structures that the dynamic garbage collector may have updated during during garbage collection. These extensions are, however, straightforward.

5 Card-Marking

Two schemes for recording pointer stores were explored at length; only one of these was implemented. The scheme that was discarded was called *card-marking.* It was similar to the Moon's ephemeral garbage collector, as one might implement it without virtual memory system cooperation; the significant differences had mostly to do with a desire to keep the pointer-setting time low.

5.1 Division of Memory; Determination of the Root Set

The root set is recorded through a scheme much like Moon's. Memory is divided at a fine level into pieces called *cards*; these correspond to the 3600's pages, but we do not call them pages in order to avoid confusion with virtual memory pages on the machine in question.

There are two tables used in determining the root set; these are called the primary card mark table and the secondary card mark table. The primary card mark table is a table of bits directly mapped to all the cards in the address space. There is one bit per card; the bit is set whenever a pointer is stored in the card. Thus the bit is a sort of dirty bit for the card; it indicates that the card has been modified and must be scanned to determine whether it contains a reference to some ephemeral level.

The secondary card mark table is implemented sparsely as a collection of tables; every segment that can contain pointers has associated with it a portion of the secondary card mark table. The secondary card mark table contains one byte per card; each of these card entries contains one bit for each ephemeral level (thus we have a maximum of eight ephemeral levels). An ephemeral level's bit is set to indicate that the card contains a reference to that particular ephemeral level. The secondary card mark table is updated at garbage collection scan time.

The size of cards is a compromise between a desire to keep the card mark tables small (which argues for a large card size) and a desire to minimize the amount of time spent in scanning a card already in main memory for a possibly nonexistent reference to an ephemeral object (which argues for a small card size). 256 words is probably a good card size on machines with, say, 28 bits of address space, like the Sun-3; this would give a primary card mark table size of 32 kilobytes, and the per-segment secondary card mark table portions would be 64 bytes each. Machines with 32 bits of address space may motivate larger card sizes, but in no case should the card size be larger than a virtual memory page frame, because of the much increased likelihood of unnecessary page faults at scan time.

As in the proposed stock hardware implementation of Moon's ephemeral garbage collector (Section 2.3.2), card-marking requires that the object creation routines be modified to record in a table the location of the first tagged word on each card; this allows scanning of cards containing untagged data.

Card-marking allows a compact representation of recorded pointer stores. What are the dynamic characteristics of this technique; i.e., what does it save us at pointer storage time? Assuming we have 256-word cards and a 28-bit address space, a card's byte in the primary card mark table is determined by the high 15 bits (13-27) of the 28-bit address; the bit in question is given by bits 10-12 of the address. If we place the primary card mark table in a register, or at some constant offset from a register, we can mark a card in six instructions on the MC68020, with two temporary data registers; the MC68020 code for this is shown in Figure 4.

Here is the advantage of card-marking, then: with only six instructions required to make an entry in the primary card mark table, pointer storage can probably continue to be coded in-line, so that the expense of an out-of-line subroutine call is saved; performance on pointer-storage-intensive benchmarks that do not discard much storage is likely to be quite good.

5.2 Performing a Garbage Collection

Garbage collection in a card-marking scheme is similar to, although simpler than, Moon's ephemeral garbage collection; first in being stop-and-copy, as opposed to incremental, and second in not being directly concerned with the virtual memory system. A Pidgin ALGOL description of card-marking garbage collection is given in Figures 5 and 6.

Following is a description of how a card-marking garbage collection proceeds.

```
;; Assume that the location to be stored in is in
;; a0, and that the primary card mark table is at a
;; constant offset crdtbl from the systemic
;; quantities vector (SQ), held in an address
;; register.
;; Store the reference location in temporary that
;; will get byte pointer.
move.l a0, d0
;; 68K can only immediate shift 8 places; we need 10
lsr.l #8, d0
move.l d0, d1   ;other temporary for bit field
lsr.l #5, d0    ;now we have the byte index in d0
lsr.l #2, d1    ;and the bit index in d1
;; note bset will mask all but the low three bits of
;; the bit index
bset.b d1, (crdtbl, sq, d0.w)   ;set the bit
```

Figure 4: Marking a card on the MC68020.

When we garbage-collect an ephemeral level, we first scavenge the stack and mark registers, just as with a dynamic GC. Then we consult the secondary card mark table, and scan all cards whose secondary marks state that they contain pointers to this level, except those within the level itself; these need not be scanned, because they cannot contain roots for this level.

The scan proceeds as follows: each word in the card is fetched. If the word is a pointer to the ephemeral level being garbage-collected, it is treated as a root, and a scavenge is performed on the object it points to; but, in any case, if after the potential scavenge the word contains a pointer to any ephemeral level at all, a flag for that level is set (note that the placement of this operation after the scavenge guarantees that the appropriate level's flag is set – the datum has changed levels in the scavenge). When the scan of the card is finished, the secondary card mark table entries for this card are reset from these flags, and the primary mark for this card is cleared, so that we can avoid scanning the card twice.[11]

[11] Note A.4 gives a comparison with the Symbolics approach.

```
procedure card_marking_gc ();
  begin
    push marked registers on stack;
    from_level := 0;
    done := false;
    while (not(done))
      begin
        to_level := from_level + 1, or dynamic space, if
                    from_level is the last ephemeral level;
        scavenge_stack(from_level, to_level);
        for each card number c
          if card_ephemeral_level[c] = from_level
            then begin
              for each level
                secondary_card_mark_table[c, level] :=
                  false;
              primary_card_mark[c] := false
            end
            else
              if secondary_card_mark_table[c, from_level] =
                  true
                then begin
                  scan_and_scavenge_card(c, from_level,
                                         to_level);
                  primary_card_mark[c] := false
                end
        if from_level = 0
          then for each card number c
            if primary_card_mark[c] = true
              then begin
                  scan_and_scavenge_card(c, from_level,
                                         to_level);
                  primary_card_mark[c] := false
              end
        if to_level is not full, or is dynamic space
          then done := true
          else from_level := from_level + 1
      end
  end
```

Figure 5: The card-marking garbage collection algorithm.

```
procedure scan_and_scavenge_card
  (card_number, from_level, to_level);

  boolean vector level_flags[number_of_ephemeral_levels];

  begin
    for i from 0 until number_of_ephemeral_levels
      do level_flags[i] := false;
    for each word in the card, beginning at the first tagged
        word
      begin
        if the word is an immediate constant
          then continue at the next word;
        if the word is the header of a vector of untagged
            data
          then continue at the first word after the vector;
        if the word points into from_level
          then scavenge_word(word, from_level, to_level);
        if word points to an ephemeral level
          then begin
            l := the level pointed to;
            level_flags[i] := true
          end
      end
    for i from 0 until number_of_ephemeral_levels
      do secondary_card_mark_table[card_number, i] :=
        level_flags[i]
  end
```

Figure 6: Scanning and scavenging a card.

When we have finished scanning all the cards whose secondary marks state that they contain references to the ephemeral level being garbage-collected, we scan all the cards whose primary marks are still set, except those within the ephemeral level being garbage-collected.[12] The scan is exactly like that performed for cards found through the secondary card mark table; note that this means that secondary card marks are updated and primary card marks are cleared.

When we have finished scanning all the cards whose primary marks were set, we have scavenged all the data in the ephemeral level that was being garbage-collected, and we can clear its primary and secondary card mark tables. Note that this means that the entire primary card mark table is now clear.

5.3 The Problem with Card-Marking

The inner loop of the piece of code that scans a card on the MC68020 was presented in Figure 1, on page 13. The time required to scan a 256-word card on the MC68020 was estimated at 816 microseconds. The gain in pointer-storage speed afforded by card-marking is substantial, but it was estimated at less than the loss due to card-scanning.[13]

Furthermore, the gain in pointer-storage speed was likely to be lost on machines with thirty-two bit address spaces, especially those in which the LISP address space was to be organized sparsely. Here a contiguous primary card mark table would be prohibitively large, and so a two-level map would be necessary; but this would be nearly as expensive at pointer-storage time as the scheme we actually implemented, which we shall call, for want of a better name, *word-marking.*

[12] Note A.4 discusses the virtual memory behavior that will result from this scanning order.

[13] This was only an estimate, however, and we have come to wonder whether it was correct. A prototype card-marking implementation would certainly answer these questions; see Section 8, however, for other possibilities.

6 Word-Marking

6.1 Recording the Root Set

The scheme used to record the root set in the Lucid Common LISP Ephemeral Garbage Collector is called *word-marking.* Word-marking uses two different data structures to record the root set. The first is a table of modification bits; the second is a set of explicitly-managed lists.

6.1.1 Modification Bit Tables

We divide the address space into large pieces called *segments;* on the MC68020, these are 64 kilobytes in length. Their exact size is not critical; making them 64 kilobytes in length allows a simple instruction sequence to extract a segment number from a pointer.

There is for each allocated segment a table of bits, called a *modification bit table* (MBT). The MBT contains one bit for each longword in the segment; thus, on the MC68020, MBTs will be 2 kilobytes in length. Every segment has associated with it an MBT, but the MBTs are sparsely allocated, in that there will be a single MBT shared by all the segments for which we do not need to record pointers into ephemeral space; these include the segments in the youngest ephemeral level, the unscanned segments, and segments holding non-pointer data. This MBT is called the *non-recording* MBT, and is specially recognized by the garbage collector.

The MBTs reside in static space, and are explicitly managed by the memory manager. They are allocated in groups, and stored contiguously, for slightly better locality on systems with page frames larger than 2 kilobytes.

The MBT stores the same sort of information that the primary card mark table was to hold in the card-marking scheme, but the information is stored on a per-word basis. That is: in card-marking, when one modifies a location, the bit for the card in which the location resides is set. In word-marking, when one modifies a location, the MBT for the segment in which the location resides is fetched, and the bit within it corresponding to the location is set.

So that the garbage collector need not examine the MBT for each segment that might have been modified, there is a table, called the segment modification cache, which contains one byte for each segment; the byte for a segment is set nonzero whenever an entry is made in its modification table. A byte is used for each segment because the table must be modified quickly when a pointer is stored.

The segment modification cache must also be read quickly at garbage collection time. On a machine with a 28-bit address space, the segment modification cache is 4 kilobytes in length.[14] With a longword test, the entries for 4 segments can be checked at once. A table of bits would have allowed quicker examination, but would make pointer-setting slower by several instructions.

Note that the segment modification cache cannot practically be spread among the modification bit tables, through some technique where a designated location in the MBT held a value indicating whether the MBT had been modified since the last garbage collection. This would allow us to save an instruction at pointer storage time, but would degrade virtual memory performance at garbage collection time, as every allocated MBT would have to be examined. The MBTs may occupy some 3% of allocated storage; examining each one could significantly increase virtual memory traffic.

Updating both the modification bit and the segment modification cache requires some ten instructions,[15] a temporary address register, and two temporary data registers on the MC68020; the code is shown in Figure 7.

The length of this instruction sequence is sufficiently great that it must be coded as an out-of-line call, or significantly increase the amount of code in the LISP image.

[14] Of course, in most applications, only a fraction of the address space of the processor is allocated; the table is only searched as far as the last allocated segment.

[15] By using the MC68020's memory indirect post-indexed addressing mode, we can shorten this to nine instructions, and this would be faster on the MC68030; however, it will be considerably slower in most cases on the MC68020.

```
;; SQ is an address register holding the base
;; address of a vector of systemic quantities.
;; SMCACHE is the constant offset from the base of
;; the SQ vector to the base of the segment
;; modification cache.  MBTTBLS is the constant
;; offset from the base of the SQ vector to the slot
;; holding a pointer to the base of the
;; segment-number indexed table of MBTs.
move.l a0, d0   ;location being modified is in a0
move.l a0, d1   ;d0 and d1 are temporaries
swap d1         ;low 16 bits of d1 now hold segment
;; the segment modification cache lies directly
;; after the SQ vector.  Set location to indicate
;; segment modified.
move.b #-1, (smcach, SQ, d1.w)
;; For compactness, the next two instructions
;; may be replaced by the single instruction
;; move.l ([mbttbls, SQ], d1.w*4, 0), a1,
;; but this would be slower on the 68020.
;; Get address of table of modification tables in a1
move.l (mbttbls, SQ), a1
;; get address of this segment's MBT in a1
move.l (0, a1, d1.w*4), a1
;; get bit field in d0 (low 3 are bit to set)
lsr.w #2, d0
move.l d0, d1   ;d1 will be byte in table
;; make 11 bits in low half be byte in table
lsr.w #3, d1
bset.b d0, 0(a1,d2.w)   ;set the bit
```

Figure 7: MC68020 code to update an MBT and the segment modification cache in a word-marking scheme.

6.1.2 Entry Backpointer Lists

The modification bit tables hold very little information; we must examine the locations that were modified to determine whether they contain pointers to an ephemeral level. We do this at garbage collection time. The garbage collector examines modified locations and adds the addresses of locations that point into an ephemeral level to that ephemeral level's entry backpointer list, or EBPL. The EBPLs are explicitly-managed lists maintained in a block of static space by the garbage collector.

The EBPLs are managed in such a fashion that their entries are unique; there is no duplication of entries. They are lists, rather than queues, because those for different levels grow and shrink dynamically. They are grown only when the youngest ephemeral level is garbage-collected and modification bit tables are examined. They will also shrink at this time; a modification may mean that a pointer to an ephemeral level was replaced by a pointer to some other ephemeral level, or to a location outside of ephemeral space. The management of the EBPLs is explicit: when an entry is removed, its cons cell is returned to a freelist. When the oldest ephemeral level is garbage-collected into dynamic space, its entire EBPL is linked into the freelist.

Because the EBPLs are updated at garbage collection time, if a program does not create objects, it will not pause. Also because the EBPLs are updated at garbage collection time, it is possible in a linear pass through them to maintain unique entries; such a linear pass would not be possible at pointer-setting time.

Note that the use of EBPLs for level-specific information means that word-marking imposes no limitation on the number of ephemeral levels allowed.

6.2 Performing a Garbage Collection

A Pidgin ALGOL routine that performs a word-marking garbage collection is shown in Figures 8 and 9. An explanation of its working follows.

Initially, we scan the stack and mark registers, just as with a dynamic GC. Then we make a pass over the EBPLs. Any EBPL entry corresponding to a location whose MBT entry is set is elided; the scavenge from MBTs

```
procedure word_marking_gc ();
  begin
    push marked registers on stack;
    from_level := 0;
    done := false;
    mbts_not_empty := true;
    failed_to_clear_mbt := false;
    while (not(done)) begin
        to_level := from_level + 1, or dynamic space, if
                    from_level is the last ephemeral level;
        scavenge_stack(from_level, to_level);
        for each location in each EBPL
          if mbt_entry_set(location)
            then remove the location from the EBPL;
        for each location in the EBPL for from_level begin
            if location is in from_level or to_level
              then remove the location from the EBPL;
            if location is not in from_level
              then scavenge_word(location, from_level,
                                           to_level)
          end
        if mbts_not_empty then begin
            for each segment whose segment modification
                  cache entry is set
              if segment_mbt(segment) = non_recording_mbt
                then clear_segment_mod_cache_entry(segment)
                else if scavenge_segment_from_mbt
                        (segment, from_level, to_level);
                    then clear_segment_mod_cache_entry(segment)
                    else failed_to_clear_mbt := true;
            mbts_not_empty := failed_to_clear_mbt
          end
        if to_level is dynamic space
          then link from_level's EBPL to the EBPL freelist
          else link from_level's EBPL to to_level's EBPL;
        set from_level's EBPL to nil;
        if to_level is not full, or is dynamic space
          then done := true
          else from_level := from_level + 1
      end
  end
```

Figure 8: The word-marking garbage collection algorithm.

```
procedure scavenge_segment_from_mbt(segment, from_level,
                                             to_level);
  begin
    mbt := segment_mbt(segment);
    entries_not_cleared := false;
    for each location corresponding to a set entry in mbt
      if the location contains immediate constant data, or
          does not point into an ephemeral level
        then clear_mbt_entry(location)
        else begin
          level := the ephemeral level into which location
                   points;
          if level = from_level
            then scavenge_word(location, from_level,
                                         to_level);
          if location itself is in to_level
            then clear_entry := true
            comment add_to_ebpl returns true if successful
            else clear_entry := add_to_ebpl(level, location);
          if clear_entry
            then clear_mbt_entry(location)
            else entries_not_cleared := true;
        end
    return(not(entries_not_cleared));
  end
```

Figure 9: Scavenging the words in a segment that point into ephemeral space by examining its modification bit table.

performed later will examine the contents of the location and determine which EBPL it should now be placed on, if any. This operation guarantees uniqueness of entries on EBPLs, and also guarantees that for the rest of the garbage collection, the locations listed in EBPLs will be known to contain pointers into the corresponding ephemeral level.

Then we fetch the EBPL for this level.[16] For each location listed in the EBPL, we determine whether the location is in the space being garbage-collected or the space being garbage-collected into; in either case we reclaim the EBPL cons. We can do so because pointers stored within an ephemeral level are not considered part of the root set for that ephemeral level; thus they should not be on the level's EBPL.

If the location is not in fromspace,[17] we scavenge it.

Now we scan the segment modification cache entries for the segments that may contain pointers to this ephemeral level. When we find a segment that has been modified, we fetch and examine its MBT. If the MBT is the unique non-recording MBT, we need not examine it further, as it is associated only with segments that cannot contain ephemeral references. Otherwise, the MBT is examined a word at a time for nonzero entries; because this is simply a check for nonzero entries, it is a two-instruction `dbne` loop on the MC68020, performed for 512 words. The scan could be optimized by recording the least and greatest locations modified in the segment, but this would make pointer storage still slower.

When a modified location is found, its contents are examined; if these do not point into an ephemeral level or are constant immediate data, the MBT entry for the location is cleared, and the search continues at the next word. If the location contains a pointer to an ephemeral object, then, if the object is in the level being garbage-collected, that is, fromspace, the location is

[16] Note that, if this is a garbage collection of the youngest ephemeral level, the EBPL will be empty, because the modification table is scanned only when a garbage collection happens (but see note A.5).

[17] This check is redundant if the algorithm is implemented exactly as shown, as we reclaim EBPL entries for locations in tospace before linking fromspace's EBPL into tospace's EBPL, to reduce the chance of running out of EBPL conses. If the obvious simplification is made, however, this step would be necessary to allow the reclamation of EBPL conses and ephemeral objects pointed to only by dead ephemeral objects at later levels; for example, the ephemeral garbage collector could not otherwise reclaim circular structures spread across more than one ephemeral level.

scavenged. If the location itself was in tospace, we simply clear its MBT entry; this guarantees that MBTs for empty levels are cleared. Otherwise we attempt to add it to the appropriate EBPL. If we were not out of EBPL conses and thus succeeded in adding the location to the appropriate EBPL, we may clear the location's MBT entry, but otherwise we must leave it set, so that we examine the location at the next garbage-collection, as it is known to contain an ephemeral reference. This state, in which the EBPL freelist is empty, will not persist, because eventually a garbage collection of the oldest ephemeral level will happen; when it completes, all EBPLs will again be null.

When we have finished scavenging ephemeral references recorded in the modification tables, we have copied all the live data out of the ephemeral level being garbage-collected. If we succeeded in clearing all MBTs, we note the fact, so that we need not examine the segment modification cache on the next garbage collection. Note that we have necessarily cleared the segment modification cache entries and MBTs for the segments in fromspace; we have also elided from the EBPL for fromspace all entries whose locations were in tospace. We must now update the EBPL for tospace to include entries for locations that point to the data just copied into it; this we do by linking to its end the EBPL for fromspace, and setting the EBPL for fromspace to nil.

7 Performance Measurements and Analysis

We describe two sets of performance measurements. The first was collected in January of 1988, on a preliminary release of the ephemeral garbage collector running on various Sun workstations, and measures performance on several of the Gabriel benchmarks [7]. The second set of measurements measures performance of a prototype version of the system, running the Lucid Common LISP production compiler on Apollo workstations; these measurements were collected in the summer of 1987.

7.1 Performance on the Gabriel Benchmarks

The Sun benchmarks are summarized in Tables 1, 2, and 3. Timings were measured on single-user machines with network paging over a 10-megabaud Ethernet to a Sun-3/180 file server; the paging devices on the file server were fast disks operating through SMD interfaces. There was essentially no network contention when the timings data were collected.

All three machines used in benchmark timings used MC68020 processors clocked at 16 megahertz. The Sun-3/75 was configured with 4 megabytes of physical memory; the Sun-3/110, with 8 megabytes of physical memory, and the Sun-3/180 with 16 megabytes of physical memory.

The benchmarks were run with the operating system in single-user mode to avoid any anomalies from running daemons. They were compiled with the Lucid Common LISP/Sun 3.0 production compiler, with speed and safety settings of 3 and 0, respectively. The version of LISP used was a beta-test version of Lucid Common LISP/Sun Release 3.0. It was configured with 82-segment dynamic semispaces (5.4 megabytes each), and three ephemeral levels; level 0 was 8 segments (512 kilobytes) in length, and levels 1 and 2 were each 10 segments (640 kilobytes) in length. In each case the best timing from several runs is given. These benchmarks are described in detail in [7].

In most cases, ephemeral garbage collection reduced the elapsed real time for execution of these benchmarks; this is especially so in cases where several dynamic garbage collections had to be performed. The difference is most

(dotimes (i 10) (boyer-setup) (boyer-test))			
Processor	**Parameter Measured**	**Garbage Collector**	
		Dynamic only	**Ephemeral**
Sun-3/75 16mhz MC68020 4mb main memory	Elapsed Real Time	235.5	206.7
	CPU Time	162.7	190.46
	Dynamic Bytes Consed	18,139,280	1,665,216
	Dynamic Garbage Collections	3	0
Sun-3/110 16mhz MC68020 8mb main memory	Elapsed Real Time	244.2	176.5
	CPU Time	166.4	174.4
	Dynamic Bytes Consed	18,139,280	1,665,216
	Dynamic Garbage Collections	3	0
Sun-3/180 16mhz MC68020 16mb main memory	Elapsed Real Time	148.6	171.9
	CPU Time	144.0	171.9
	Dynamic Bytes Consed	18,139,576	1,533,040
	Dynamic Garbage Collections	4	0

Table 1: BOYER benchmark timings. Times are in seconds.

`(dotimes (i 20) (deriv-run))`			
Processor	**Parameter Measured**	**Garbage Collector**	
		Dynamic only	**Ephemeral**
Sun-3/75 16mhz MC68020 4mb main memory	Elapsed Real Time	389.0	98.4
	CPU Time	172.9	89.0
	Dynamic Bytes Consed	39,205,320	0
	Dynamic Garbage Collections	7	0
Sun-3/110 16mhz MC68020 8mb main memory	Elapsed Real Time	387.8	89.8
	CPU Time	177.0	89.2
	Dynamic Bytes Consed	39,205,320	0
	Dynamic Garbage Collections	7	0
Sun-3/180 16mhz MC68020 16mb main memory	Elapsed Real Time	103.8	89.3
	CPU Time	101.2	89.3
	Dynamic Bytes Consed	39,205,320	0
	Dynamic Garbage Collections	7	0

Table 2: DERIV benchmark timings. Times are in seconds.

(dotimes (i 100) (destructive 600 50))			
Processor	**Parameter Measured**	**Garbage Collector**	
		Dynamic only	**Ephemeral**
Sun-3/75 16mhz MC68020 4mb main memory	Elapsed Real Time	415.8	196.3
	CPU Time	249.0	178.1
	Dynamic Bytes Consed	34,489,080	0
	Dynamic Garbage Collections	6	0
Sun-3/110 16mhz MC68020 8mb main memory	Elapsed Real Time	437.2	176.9
	CPU Time	259.6	176.9
	Dynamic Bytes Consed	34,489,080	0
	Dynamic Garbage Collections	6	0
Sun-3/180 16mhz MC68020 16mb main memory	Elapsed Real Time	194.5	177.5
	CPU Time	191.5	177.5
	Dynamic Bytes Consed	34,489,080	0
	Dynamic Garbage Collections	6	0

Table 3: DESTRUCTIVE benchmark timings. Times are in seconds.

dramatic on the machines configured with less memory; this is because ephemeral garbage collection drastically reduces the size of the working set. Note that the differences between central processing unit time and real time on these machines is large under dynamic garbage collection, and much smaller under ephemeral garbage collection; as the machines were running in single-user mode, the discrepancy between real and central processing unit times will be due almost totally to virtual memory system overhead.

It is interesting to note that, in some cases, ephemeral garbage collection reduced the amount of central processing unit time required for the execution of a benchmark. We expect that the reduced size of the root set accounts for much of the performance improvement, as, among these benchmarks, only BOYER retains for long the large structures created. Thus it seems unlikely that much transporting occurred in dynamic garbage collection.

The DESTRUCTIVE benchmark timings (Table 3) show particularly good performance under ephemeral garbage collection. Reference to the source code reveals one reason: only two of the six destructive operations used will result in invocation of the out-of-line subroutine that records pointer stores. The others are stores either of declared fixnums or constant symbols; as noted in Section 4.1, the compiler can optimize out pointer-recording in these cases.

In the BOYER timings (Table 1), we see a pattern characteristic of the Lucid ephemeral garbage collector: enhanced virtual memory performance is gained at the expense of increased central processing unit load. The size of the working set has been decreased; the improved virtual memory performance has resulted in reduced elapsed times to perform a task, but at a cost of more work for the central processing unit. On machines with better virtual memory performance, use of the ephemeral garbage collector is less attractive.

7.2 Performance of the Compiler Under Ephemeral Garbage Collection

The Apollo performance measurements are summarized in Table 4. These measurements were taken on single-user machines with local paging and file disks; network traffic was virtually nil. The Apollo DN4000 processor is an

Processor	Parameter Measured	Garbage Collector	
		Dynamic only	Ephemeral
DN570-T 20mhz MC68020 8mb main memory 154mb disk, ST506	Elapsed Real Time	32,171.5	28,554.7
	CPU Time	16,429.0	20,275.1
	Process Disk I/O	499,881	285,034
	Dynamic Bytes Consed	722,183,608	87,741,088
DN3000 12mhz MC68020 8mb main memory 348mb disk, ESDI	Elapsed Real Time	30,269.8	31,535.6
	CPU Time	19,768.3	26,316.8
	Process Disk I/O	485,096	256,155
	Dynamic Bytes Consed	722,163,960	90,460,272
DN4000 25mhz MC68020 32mb main memory 348mb disk, ESDI	Elapsed Real Time	12,358.1	15,693.7
	CPU Time	11,680.3	15,012.1
	Process Disk I/O	8,226	6,484
	Dynamic Bytes Consed	722,121,048	92,014,816

Table 4: Global recompilation performance measurements on Apollo workstations. Times are in seconds.

MC68020 clocked at 25 megahertz; the DN4000 in question was configured with 32 megabytes of physical memory and a fast 348 megabyte disk drive operating through an ESDI interface. The DN3000 processor is an MC68020 clocked at 12 megahertz; the DN3000 used was configured with 8 megabytes of physical memory and a fast 348 megabyte disk drive with the same ESDI interface as on the DN4000. Finally, the DN570-T processor is an MC68020 clocked at 20 megahertz; the DN570-T used for performance measurements was configured with 8 megabytes of physical memory and a relatively slow 154 megabyte disk drive operating through an ST506 interface.

The task executed was a recompilation of all the files in the LISP system; it also served as a testbed for debugging the ephemeral garbage collector prototype.[18]

[18] In its default configuration, the Apollo version of the Lucid Common LISP compiler makes use of a facility that allows block allocation and deallocation of temporary storage. In the results shown here, this facility has been disabled, as such block allocation and deallocation is not available to user programs, and our intention is to provide an analysis of the behavior under ephemeral garbage collection of large programs utilizing the usual storage allocation facilities.

Use of the ephemeral garbage collector degraded performance in the global recompilation task on both the DN4000 and the DN3000. On the DN4000, the task took 27% more elapsed real time under ephemeral garbage collection than under dynamic garbage collection; on the DN3000, the figure was about 4.2%. On the DN570-T, however, the elapsed real time under ephemeral garbage collection is 11.2% less than under dynamic garbage collection.

These results confirm the conclusion we reached in examining our timings on Sun workstations: the Lucid Ephemeral Garbage Collector improves virtual memory performance at the expense of central processing unit time. Examination of the "Disk I/O" figures show a reduction in virtual memory traffic on all three systems; on the DN3000 and DN570-T this reduction was in excess of 42% of that observed under dynamic garbage collection; on the DN4000, configured with 4 times the amount of physical memory, the reduction in disk I/O was only 21%. The DN3000 and DN570-T disk I/O figures are very close, as would be expected from machines with identical amounts of physical memory; however, the DN570-T's faster processor gives it a lower elapsed time under ephemeral garbage collection. Under dynamic garbage collection, this advantage is reversed by the DN3000's faster paging device.

On the whole, however, these measurements show much worse performance under ephemeral garbage collection while performing a global recompilation on Apollo workstations than while running benchmarks on Sun workstations. Leaving aside momentarily the fact that the tasks being performed are different, we would still expect some discrepancy in performance, due to the different virtual memory characteristics of the systems being compared.

The Sun-3 has 8-kilobyte page frames, as compared to the Apollo's 1-kilobyte page frames; the coarser page size hurts performance in a pointer-oriented language with heap allocation, like LISP. Furthermore, the Apollos whose performance we measured had more memory and faster paging devices than did the Suns. But we also see wide discrepancies in factors besides virtual memory behavior; in particular, the central processing unit time expense of ephemeral garbage collection is far greater in the Apollo performance measurements.

Of course, we are comparing apples and oranges; the tasks being performed were different. What is interesting is how they are different. The compiler makes heavy use of hash tables, especially when reading input files, and hash tables are invalidated by copying garbage collection, as the hash values of objects stored in them are computed from their addresses, which are assumed to have changed. Thus references to hash tables between garbage collections require recomputing hash values for all objects in the tables. Because ephemeral garbage collections occur so much more often than do dynamic garbage collections, this task will have to be performed many times more often under ephemeral garbage collection; we believe that this accounts for a lot of the central processing unit time.

Other measurements have led us to believe that the default configuration of ephemeral spaces is less than ideal for use of the compiler. The compiler creates large data structures while compiling a file, and retains some of them through the entire compilation of the file; thus there is the possibility that these large structures will be moved several times by ephemeral garbage collection, and finally advanced into dynamic space, where they are released. We have in fact observed this behavior by metering compilation.

This is not a problem peculiar to the compiler; we expect that many programs that build large temporary data structures will exhibit similar behavior. Note that, because these structures are built out of small parts, the automatic allocation of very large objects in dynamic space (see Section 4.5) is of no help here. This is called the *pig-in-the-snake* problem.[19] In general, it can be solved only by tuning the number and the sizes of ephemeral levels for optimal behavior on the problem at hand. In the case of the compilation benchmark, we can see that a greater delay between garbage collection times, as occurs in dynamic garbage collection (because semispaces are larger), would result in moving these structures less often or possibly not at all; they might perish first. We expect that a larger first ephemeral level would have much the same effect.

Finally, the compiler performance measurements were made on an earlier version of the system; some of the continued development in the interim may have led to better performance in the later benchmarks. Further testing is planned to analyze compiler performance with the current system.

[19] I am indebted to Jon L. White for this terminology.

8 Conclusions and Future Work

8.1 Conclusions

The performance analysis presented in Section 7 may be summarized broadly and in brief:

- At tasks in which large amounts of data are allocated and then discarded, the Lucid Ephemeral Garbage Collector reduces both the elapsed real time and the central processing unit time required.
- At tasks in which large amounts of data are allocated and retained, the ephemeral garbage collector will enhance performance by reducing the size of the working set, gaining virtual memory performance (and thus elapsed real time) at the expense of central processing unit time.
- On processors with very good virtual memory performance (those configured with large amounts of physical memory and fast paging devices) the ephemeral garbage collector may degrade performance significantly. We believe that this is mostly due to the overhead of recording pointer storage.
- Ephemeral space configuration should be tuned to individual problems to avoid extra transporting work.

Additionally, ephemeral garbage collections do not cause noticeable pauses. This, and the performance characteristics described above, promise much for the manufacturer of interactive systems built in LISP and delivered on workstations. Ephemeral garbage collection allows smaller workstations without local disks to be used as symbolic processing delivery vehicles without prohibitive effects on performance.

8.2 Future Work

We did not establish conclusively that our means for recording the root set was superior to card-marking. More performance measurements should be made to support or controvert this argument.

We do not know for certain that overall performance is actually enhanced by our strategy of delaying until garbage collection time the determination of the spaces in which a pointer is stored and to which it points. Instrumentation of the pointer storage routine and careful metering of paging behavior will yield an answer to this question.

A scheme by which repeated scanning of the stack on scavenges of successive levels could be avoided would be useful; the stack often grows to considerable size during the execution of LISP programs, as recursion is encouraged. The obvious means by which to avoid scans after the initial one would be to use the EBPLs to hold backpointers to ephemeral references on the stack; however, this would mean less efficient use of EBPL conses, as many EBPL entries for the stack would become invalid between invocations of the garbage collector. As we would need to scan the stack at the inception of each invocation of the garbage collector, we could modify the initial scan of EBPLs (which elides entries for locations whose MBT entries are set) to remove entries for locations on the stack.

However, because most ephemeral garbage collections are only of the first ephemeral level, the additional bookkeeping overhead of this scheme might not pay off. Possibly we could invoke it only when the succeeding level was filled nearly to capacity, as this would signify increased likelihood of a scavenge of that level.

It would be useful to have a means of determining the best configuration of ephemeral space for a given problem. The configuration of ephemeral spaces is a compromise between several conflicting constraints. Increasing the number of ephemeral levels causes fewer objects to be advanced to dynamic space, decreasing the number of dynamic garbage collections; however, it increases the number of times a relatively permanent object must be copied before it is advanced to dynamic space. Increasing the size of the first level will give ephemeral objects more time to perish before we ever transport them; but it also reduces locality of reference. A good model of these constraints would allow us to build a tool that could analyze the dynamic behavior of a program and make configuration recommendations, or possibly even vary dynamically the configuration of ephemeral space.

Some means of performing approximately depth-first copying (either that used by Moon [10, page 238], or some other scheme) would improve locality of reference.

Of course, the availability of support from virtual memory systems would make a scheme like the one described in Section 2.3.2 more attractive than the one we implemented.

Part III

Appendices

A Notes

A.1 Performance of incremental garbage collection

Baker [1, page 26] noted that his incremental garbage collector would require a larger working set size than would a simple stop-and-copy garbage collector, as the user computation running would require a certain amount of working storage, made larger by the necessity of following forwarding pointers through evacuated objects in fromspace, and the scavenger would also constantly be cycling through memory in tospace and fromspace unrelated to that currently in use by the user computation.

Empirical evidence bears out Baker's concern. Moon [10, page 236] reports that the poor virtual memory performance of the Baker-style garbage collector on the Symbolics 3600 resulted in users' avoidance of its use whenever possible.

A.2 Shaw's suggested extension to virtual memory systems

Shaw [12] suggests a scheme in which the virtual memory system allows the LISP process to clear the dirty bits actually maintained by the hardware; prior to actually clearing the bits, the virtual memory system saves their state away, so that two tables are consulted by the virtual memory system when a page is ejected from physical memory.

Such a facility would allow the LISP process to remember and clear mark bits just before a garbage collection. During the garbage collection, it would scan the pages remembered to have been marked; it would write them if they had ephemeral references, as these references would need to be updated. Thus only the pages with ephemeral references would be marked dirty immediately after a garbage collection; this would eliminate some useless page-scanning.

Note, however, that the facility Shaw suggests, while potentially valuable, is not essential to the success of Moon's garbage collection scheme on a general-purpose computer. Dirty bits alone, maintained in the usual sense, will work, because the wasted page scans are of in-core pages; the expense of this operation, even on general-purpose machines, is dwarfed by any backing store operations that garbage collection may require.

It is interesting to note that Shaw's scheme does not provide any means to verify whether ejected dirty pages contain pointers to ephemeral levels before they are written to backing store; thus unnecessary backing store operations will occur at garbage collection time when a dirty page ejected from physical memory does not contain any references to ephemeral spaces.

A.3 Time required to garbage-collect all levels

Call the oldest level level 1, and the youngest level level m. If each succeeding level is half the size of the next younger level, and level m is of size x, the nth level is of size $2^{n-m}x$. The data in the mth level might have to be copied m times. Say that the work required to copy x words is x; then, if every level is entirely full of live data, the work required for m levels is $\sum_{n=1}^{m} 2^{n-m}mx$, or $mx\sum_{n=1}^{m} 2^{n-m}$. In the limit, this is $2mx$, where x was the work required to copy the youngest level.

A.4 Scanning order and virtual memory performance

On the 3600, an ephemeral garbage collection first scavenges the pages with GCPT bits set, and then pages whose ESRT entries are set. This probably results in many pages being scanned twice, but, as Moon comments, it is very cheap to scan a page with no ephemeral references on it, and the second scan will encounter no references in fromspace.

In our card-marking scheme, we scan the cards with set secondary marks first, and then the cards whose primary marks are set, but whose secondary marks were not set; thus we avoid re-scanning some cards. We perform the scanning in this order to optimize the overall paging behavior of the algorithm. If we wished, we could first scan the cards whose primary marks were set; when we were finished scanning them, the data would have changed levels and the secondary card mark table would reflect that, so we would also not scan any cards twice with this approach.

Scanning the cards with set secondary marks first optimizes the paging behavior in this way: the cards whose primary card marks were set are those that were recently written, and thus we assume that they may be written again soon and should be in physical memory after a garbage collection; if we were to scan them first, the cards whose secondary marks were set would be left in physical memory after a garbage collection; this would mean that the user program would first have to page them out.

One reason why Symbolics might perform the scan in the opposite order is that, because their garbage collector is incremental, the user program continues to execute very early in the garbage collection process, and thus they would like to delay disturbing the pages in physical memory as long as possible.

A.5 Updating EBPLs between garbage collections

It would be possible to update the EBPLs from the modification tables (and clear the modification tables) between garbage collections; for example, we could cause the object creation routines to scan modification tables whenever we began allocating objects in a new segment. This would not violate the design goal of predictability, which states that programs that do not create objects should not cause garbage collections, because the scanning would be motivated only by object creation. This might lead to better virtual memory performance, as the garbage collector would not then have to examine such chronologically distant locations. One result of making this modification to the system would be the possibility that level 0's EBPL would have entries at the inception of garbage collection.

B Bibliography

References

[1] Henry G. Baker, Jr. *List Processing in Real Time on a Serial Computer.* Massachusetts Institute of Technology Artificial Intelligence Laboratory Working Paper 139, Cambridge, Massachusetts, 1977.

[2] Stoney Ballard and Stephen Shirron. The Design and Implementation of VAX/Smalltalk-80. In *Smalltalk-80: Bits of History, Words of Advice,* pp. 127–150. G. Krasner (ed.), Addison-Wesley, Reading, Massachusetts, 1983.

[3] Rodney A. Brooks, Richard P. Gabriel, and Guy L. Steele. S-1 Common LISP Implementation. In *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming,* 108–113, Pittsburgh, Pennsylvania, August, 1982.

[4] Patrick J. Caudill and Allen Wirfs-Brock. A Third Generation Smalltalk-80 Implementation. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings,* ACM SIGPLAN Notices 21(11):119–129, Portland, Oregon, 1986.

[5] C. J. Cheney. A Nonrecursive List Compacting Algorithm. In *Communications of the ACM,* 13(11):677–678, November, 1970.

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

[7] Richard P. Gabriel. *Performance and Evaluation of LISP Systems.* MIT Press Series in Computer Systems, MIT Press, Cambridge, Massachusetts, 1985.

[8] Richard Greenblatt, et al. *LISP Machine Progress Report.* Massachusetts Institute of Technology Artificial Intelligence Laboratory Memo 444, Cambridge, Massachusetts, 1977.

[9] Henry Lieberman and Carl Hewitt. *A Real Time Garbage Collector Based on the Lifetimes of Objects.* Massachusetts Institute of Tech-

nology Artificial Intelligence Laboratory Memo 569, Cambridge, Massachusetts, 1981.

[10] David A. Moon. Garbage Collection in a Large LISP System. In *ACM Symposium on LISP and Functional Programming,* 235–246, Austin, Texas, 1984.

[11] Motorola, Inc. *MC68020 32-bit Microprocessor User's Manual,* Second Edition. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

[12] Bob Shaw. *Improving Garbage Collector Performance in Virtual Memory.* Hewlett-Packard Laboratories STL-TM-87-05, Palo Alto, California, 1987.

[13] David Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *ACM SIGSOFT/SIGPLAN Practical Programming Environments Conference,* 157-167, April, 1984.

[14] Jon L. White. Address/Memory Management for a Gigantic LISP Environment or, GC Considered Harmful. In *Conference Record of the 1980 LISP Conference,* 119–127, Redwood Estates, California, July, 1980.