May 1991

UILU-ENG-91-2230
CRHC-91-20

Center for Reliable and High-Performance Computing

LANGLEY GRANT
IN-61-CR
13936
P93

PERFORMANCE ANALYSIS OF GARBAGE COLLECTION AND DYNAMIC REORDERING IN A LISP SYSTEM

Rene Lim Llames

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-91-2230 CRHC-91-20	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois — 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION NASA NSF
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Ave. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) NASA Langley Research Center, Hampton, VA 23665 NSF, 1800 G St., Washington, DC 20552
8a. NAME OF FUNDING/SPONSORING ORGANIZATION NASA/NSF — 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NASA NAG-1-613, NSF (DCI) MIP-8604893
8c. ADDRESS (City, State, and ZIP Code) NASA Langley Research Center, Hampton, VA 23665 NSF, 1800 G St., Washington, DC 20552	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Performance Analysis of Garbage Collection and Dynamic Reordering in a Lisp System

12. PERSONAL AUTHOR(S)
Rene Lim Llames

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM ______ TO ______	May 1991	82

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
virtual memory, garbage collection, dynamic reordering, performance measurement and modeling, object management, locality of reference, Lisp, dynamic memory allocation.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Generation-based garbage collection and dynamic reordering of objects are two techniques for improving the efficiency of memory management in Lisp and similar dynamic language systems. An analysis of the effect of generation configuration is presented, focusing on the effect of the number of generations and generation capacities. Analytic timing and survival models are used to represent garbage collection runtime and to derive structural results on its behavior. The survival model provides bounds on the age of objects surviving a garbage collection at a particular level. Empirical results show that execution time is most sensitive to the capacity of

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) — 22c. OFFICE SYMBOL

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

the youngest generation. The existence of a range of optimum values demonstrates the potential for the tuning of garbage collection.

A new memory management system integrating dynamic reordering with garbage collection is described. The system supports schemes for preserving object order in virtual memory during garbage collection, both approximately and exactly.

We present a technique, called scanning for transport statistics, for evaluating the effectiveness of reordering, independent of main memory size. Reordering oldspace is scanned for the number of pages containing the transported objects and statistics on their sizes, from which is computed the reduction in working set size due to reordering. The relative reduction in working set size is a measure of the density with which the actively used objects are packed into pages. Since the technique can be applied selectively in space, the portions of memory which are suitable for reordering can be identified. The method can also be used to measure locality improvement due to garbage collection.

Results from two experiments, one involving an extensive interactive session and the other a large application, show overall reductions in working set size of 48% and 58% due to reordering, with up to 93% for individual memory areas. Relative reduction in working set size was found to be greater for list space than structure space, by a factor of about three overall. The large disparity between list and structure object fragmentation in certain areas suggests that the memory management system should be able to treat list and structure space differently.

PERFORMANCE ANALYSIS OF GARBAGE COLLECTION AND DYNAMIC REORDERING IN A LISP SYSTEM

BY

RENE LIM LLAMES

B.S., University of the Philippines, 1982
M.S., University of Illinois, 1985

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1991

Urbana, Illinois

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

DECEMBER 1990

WE HEREBY RECOMMEND THAT THE THESIS BY

RENE LIM LLAMES

ENTITLED PERFORMANCE ANALYSIS OF GARBAGE COLLECTION AND DYNAMIC REORDERING IN A LISP SYSTEM

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF DOCTOR OF PHILOSOPHY

[illegible]
Director of Thesis Research

[illegible]
Head of Department

Committee on Final Examination†

[illegible] Chairperson

[illegible]

[illegible]

[illegible]

† Required for doctor's degree but not for master's.

O-517

PERFORMANCE ANALYSIS OF GARBAGE COLLECTION AND DYNAMIC REORDERING IN A LISP SYSTEM

Rene Lim Llames, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1991
Ravi Iyer, Advisor

Generation-based garbage collection and dynamic reordering of objects are two techniques for improving the efficiency of memory management in Lisp and similar dynamic language systems. An analysis of the effect of generation configuration is presented, focusing on the effect of the number of generations and generation capacities. Analytic timing and survival models are used to represent garbage collection runtime and to derive structural results on its behavior. The survival model provides bounds on the age of objects surviving a garbage collection at a particular level. Empirical results show that execution time is most sensitive to the capacity of the youngest generation. The existence of a range of optimum values demonstrates the potential for the tuning of garbage collection.

A new memory management system integrating dynamic reordering with garbage collection is described. The system supports schemes for preserving object order in virtual memory during garbage collection, both approximately and exactly.

We present a technique, called scanning for transport statistics, for evaluating the effectiveness of reordering, independent of main memory size. Reordering oldspace is scanned for the number of pages containing the transported objects and statistics on their sizes, from which is computed the reduction in working set size due to reordering. The relative reduction in working set size is a measure of the density with which the actively used objects are packed into pages. Since the technique can be applied selectively in space, the portions of memory which are suitable for reordering can be identified. The method can also be used to measure locality improvement due to garbage collection.

Results from two experiments, one involving an extensive interactive session and the other a large application, show overall reductions in working set size of 48% and 58% due to reordering, with up to 93% for individual memory areas. Relative reduction in working set size was found to be greater for list space than structure space, by a factor of about three overall. The large disparity between list and structure object fragmentation in certain areas suggests that the memory management system should be able to treat list and structure space differently.

Acknowledgments

The author would like to thank Professor Ravi Iyer for the advice, guidance, financial support, encouragement, patience, and good humor he provided during this research. The author also wishes to thank Professors Kenneth Forbus, W. Kent Fuchs, and Janak Patel for serving on his dissertation committee.

The support of NASA grant NAG-1-613 and NSF Grant (DCI) MIP-8604893 is gratefully acknowledged.

The author thanks his friends and colleagues for helping to make his stay a memorable one. Finally, the author thanks his parents for making his education possible and for their love and support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

One of the characteristics of systems which implement dynamic languages such as Lisp, in contrast to static languages such as C or Pascal, is the pervasiveness of runtime allocation and deallocation of "heap" memory. The resulting problem of efficient management of memory is compounded in a virtual memory environment, where it is important to maintain locality of reference. In such object-oriented, dynamically allocated, virtual memory systems, users rely on a garbage collector to reclaim unused storage to avoid the exhaustion of address space, as well as, and sometimes more importantly, to reduce working set size by compacting the accessible objects in memory. Modern generation-based garbage collectors can perform these functions less disruptively and more efficiently than the original techniques which often required suspending user computation and scanning all memory. However, the possibility or necessity of tuning or performance debugging of these collectors under particular program characteristics remains.

In generation-based garbage collection systems [14, 2, 15, 18, 19, 25, 30], objects are classified into age groups or generations, and younger generations are collected more frequently than older ones. This technique concentrates collection effort on the youngest objects since it has been empirically shown that the objects created most recently are the ones most likely to become garbage. The issues in such schemes are the number of generations to maintain, the threshold size of a generation (which determines its collection frequency), and the policy for promoting objects to older generations and eventually to tenured status. In general, time and space tradeoffs exist between these various configuration choices, and the optimum configuration depends on program characteristics.

Recently, it has been suggested that existing garbage collection functionality can be employed to dynamically reorder objects in memory, in effect compacting the subset of accessible objects which is being actively used [7]. The potential gain in performance or reduction in main memory requirements has been demonstrated to be quite substantial. However, appropriate measurement and analysis methods for this new memory management function remain to be developed.

This thesis is concerned with analyzing the performance implications of generations in a generation-based garbage collection system, and in dynamic reordering. The effects of varying the generation parameters are studied. The design of a new, integrated garbage-collecting and reordering memory management system is presented as well as a method for evaluating the intrinsic effectiveness of reordering, independent of main memory size.

1.1 Background

The particular Lisp system used in our study is the Symbolics Lisp machine. A detailed description of the this architecture and garbage collection system can be found in other references [15, 16]. Here, we summarize the relevant specific features and terminology.

1.1.1 Memory organization

Virtual memory is organized into *areas*, *spaces*, and *levels*. The address space for a given combination of area, space, and level is allocated in one or more blocks of contiguous addresses called *regions*. Specific kinds of objects are placed by default in their own areas, such as compiled functions, symbols, symbol property lists, and symbol print names. However, most of the objects created by an application are placed in an area named `WORKING-STORAGE-AREA`. An area may contain several kinds of spaces. Objects are created in newspace; regions to be garbage collected are atomically relabelled as oldspace during the *flip* phase of collection; during the ensuing *scavenging* phase, accessible objects in oldspace are copied into copyspace. Scavenging can occur incrementally, allowing user and other system processes to run simultaneously, or nonincrementally, in which case other processes are locked out. Static space is intended to contain objects not normally subject to collection.

An area, notably `WORKING-STORAGE-AREA`, may contain several levels, which implement the generations or age groups for generation-based garbage collection. Levels $l > 0$ are the *ephemeral* or nontenured levels, while level 0 contains tenured objects. An *ephemeral* garbage collection collects one or more ephemeral levels. It is normally triggered when the top (youngest) level in an area containing ephemeral levels exceeds a specified capacity; any other lower ephemeral levels which have also exceeded their respective capacities are then also simultaneously collected; surviving objects are copied to the next older level. A *dynamic* garbage collection collects all the ephemeral levels as well as level 0 new and copyspace. It is typically invoked infrequently to reclaim memory taken by objects which have been tenured but have become garbage (tenured garbage). A *full* garbage collection collects all that a dynamic garbage collection collects including selected static spaces.

1.1.2 Incremental copying garbage collection

Garbage collection in our measured system employs the incremental copying technique, based on modified versions of the Cheney [5] and Baker [4] algorithms. The Cheney algorithm performs breadth-first copying of linked structures without requiring an explicit stack. The Baker algorithm interleaves collection with normal processing, avoiding long, unpredictable delays to the user that would result if garbage collection were to be performed without interruption. In the Baker algorithm, the heap is divided into two spaces of equal size, *fromspace* and *tospace*. A garbage collection involves copying all accessible objects in fromspace to tospace. An object is accessible if it can be reached starting from some set of root objects, called the *root set* or *base set*. After all accessible objects have been copied, fromspace can be reused. To begin another garbage collection, the labels of the two spaces are interchanged or *flipped*. The copying technique enhances locality by removing interspersed garbage.

In the Symbolics system, the heap is divided into static and dynamic areas. Only dynamic space (or some portion of it) is garbage collected; static space is assumed to contain objects that are unlikely to become garbage. During a collection, three kinds of dynamic space become meaningful:

- The portion of dynamic space to be garbage collected is turned into *oldspace*.
- Objects in oldspace discovered to be nongarbage, by a procedure to be described shortly, are copied to *copyspace*.
- New objects created during the collection are allocated in *newspace*.

After all accessible objects in oldspace have been copied, oldspace may be reclaimed. Another collection may then begin by flipping copyspace and newspace into oldspace, and allocating a fresh copyspace and newspace. Hence, oldspace corresponds to fromspace in the Baker algorithm, and copyspace/newspace corresponds to tospace. Unlike the Baker algorithm, the three spaces are not fixed in size or location. Whatever portion of dynamic space is desired to be collected is turned into oldspace, and copyspace and newspace are allocated as necessary from free virtual address space.

The garbage collector consists of two threads of control, the *scavenger* and the *transporter*, which are interleaved with the user program and other system processes, collectively called the *mutator*. The scavenger's job is to scan through memory containing all possible references to oldspace from nongarbage objects not in oldspace. Initially, the only place where such references can exist is the root set, by definition. When the scavenger encounters an oldspace reference, the transporter is called. The transporter

(1) copies the oldspace object to copyspace and installs a forwarding pointer (in the oldspace object pointing to the version in copyspace); and

(2) changes the oldspace reference to point to the copyspace version.

If the transporter is called due to a reference to a previously copied object, it has to do only (2), i.e., use the forwarding pointer to redirect the oldspace reference. As nongarbage objects are transported, copyspace will potentially contain references to oldspace. Thus, after scanning the root set, the scavenger needs to scan copyspace as well, to "pull in" any accessible structures still in oldspace. The root set and copyspace together constitute *scavenge space*, and after both have been scavenged, no references to oldspace exist and oldspace can be reclaimed.

Besides the scavenger, the mutator could also attempt to reference objects in oldspace, which will also trigger the transporter. Transporter calls can therefore be either scavenger-induced, or mutator-induced.

The scavenger is allowed to run if the system is idle. Otherwise, the rate of performing collection work (scanning and transporting) is constrained to be proportional to the rate of allocation, i.e., the garbage collector is allocation-driven, to ensure that consumption does not outpace production of free space.

1.1.3 Approximately depth-first copying

Since the garbage collector can copy objects in whatever order it chooses, this degree of freedom can be exploited to improve spatial locality of the surviving objects. The Symbolics garbage collector modifies the Cheney algorithm such that an approximately depth-first order is realized. Whenever it is likely to result in the discovery of oldspace references, the scavenger temporarily suspends its normal linear scan of the root set and copyspace to scan the partially filled page at the growing end of copyspace. This "last page" scavenging of copyspace tends to place objects on the same page as their parent. Another technique suggested by Courts [7], in which objects evacuated by mutator-induced transporting are separated from those evacuated by scavenger-induced transporting, is also possible but not implemented in our measured system.

1.1.4 Generational garbage collection

The system provides two forms of garbage collection—the original *dynamic*, and the more recently developed *ephemeral* collector. In dynamic collection, all dynamic space is garbage collected and the root set is taken to consist of all objects in static space. The policy for initiating collections is safety-based: a collection is begun when the system decides it has reached the latest time at which a collection, if begun, could safely complete without running out of free memory space.

A dynamic collection typically requires much runtime and paging time due to the enormous size of static space and the large amount of objects that have to be transported. Although collection is interleaved with the user program, response time increases considerably due to paging. Consequently, most users turn off the dynamic collector during interactive usage.

The ephemeral garbage collector is an implementation of generational collection, which is based on two heuristics about objects:

- younger objects are more likely to become garbage than older objects (infant mortality); and
- there are many fewer references from older to younger objects than from younger to older objects.

The first heuristic suggests that we stratify dynamic space into several independently collectible *generations* or *levels*; place newly created objects in the first generation; advance surviving objects to the next higher generation; and garbage collect the younger generations more frequently. Collecting the younger generations will be more efficient since effort is expended on reclaiming areas with a high percentage of garbage, and thus little transporting work is required. When collecting all generations younger than a given level, the root set must include all references from older generations to the generations being collected. The second heuristic greatly reduces the size of the root set and suggests that it is not impractical to keep track of these backward intergenerational references.

In ephemeral collection, ephemeral (assumed to be short-lived) objects are created in the first level. The policy for initiating collections is capacity-based: a collection is begun when the first level exceeds its prespecified *capacity*. The first level is flipped simultaneously with higher levels that have also exceeded their capacities. Objects that survive a garbage collection graduate to the next level. Those surviving a collection of the last level become normal, "tenured" dynamic objects and may be collected by dynamic collection. Two tables remember the pages into which ephemeral object references have been written. These tables determine the root set for garbage collecting a particular level. The tables are called

- the Garbage Collector Page Tags (GCPT) for in-main-memory pages, and
- the Ephemeral Space Reference Table (ESRT) for on-disk pages.

A greater effort is made to minimize the size of the ESRT to avoid unnecessarily fetching on-disk pages during scavenging.[1]

1.1.5 Tagged architecture and barrier hardware

To allow the above techniques to be implemented with acceptable overhead, the Symbolics computer relies on its tagged architecture and special hardware. The processor detects in hardware

- an attempt to read into the processor a pointer to oldspace (the *read barrier*); and

[1]In other generational collection schemes, the entity serving the function of the GCPT and ESRT has been called *entry vector*, *remembered set*, and *indirection cells*.

- an attempt to write to memory a pointer to an ephemeral space (the *write barrier*).

The read barrier is required for incremental garbage collection, whereas the write barrier is required for generation-based garbage collection.

The implementation of these hardware barriers between the processor and memory relies primarily on a table for mapping a virtual address to a space type and ephemeral level. Also, the GCPT is implemented in hardware. Such support avoids the performance degradation that would result from performing address checks in microcode or Lisp.

Chapter 2

Memory Monitoring Tools

To support the analysis work, a number of software tools have been developed, including tools for providing descriptions of virtual memory usage, for collecting statistics on main memory occupancy, intergeneration references, and object populations, and for page fault tracing. The largest of these tools is a facility for dynamically collecting, analyzing, and visualizing memory usage and performance data. In this chapter, we describe the dynamic monitoring facility, discuss the salient features of the instrumentation and analysis capabilities, and show examples of its use in characterizing Lisp program behavior and tuning garbage collection. We also demonstrate how the facility can be used to observe object lifetimes as a function of the time the objects are created.

Recent papers involving measurements on Lisp programs have been concerned with processor architectures for high Lisp performance [22, 23, 18]; cache performance [17]; and garbage collection algorithms [18, 31]. These studies involved the simulation of traces at the instruction, memory reference, or object reference level. Because of the large number of events simulated, there is a practical limit on the length of programs measured, with CPU times on the order of tens or hundreds of seconds. Wilson [29] discusses the design of a memory system capable of recording a history of detailed changes made to it over a very long period of time, and reconstructing a previous state.

Our instrumentation is based on sampling memory system activity timers, event counters, and memory occupancy at epochs defined by and synchronized with garbage collection. While the data does not contain high resolution information, such as an instruction or address trace, the relatively low frequency of sampling and low data rate make it possible to monitor programs with minimal overhead for long periods of time, e.g., many hours. The long duration enables time-varying characteristics, such as program phase behavior, to be observed. The primary program and system characteristics measured are object allocation, lifetime, paging characteristics, and collector performance.

The facility described here is useful in evaluating and experimenting with different generation configurations and in guiding other tuning efforts, such as user management of objects and user control of paging policies.

2.1 Instrumentation and Analysis

The software monitor collects data which makes it possible to (1) evaluate various performance measures, such as page fault rates and garbage collection efficiency over any specified portion of a measured program; and (2) to construct two views of memory: a global (or non-area-specific) view showing the time variation in the usage of the various spaces and distinguishing only between ephemeral (level $l > 0$) and tenured (level 0) spaces; and a by-area view, showing the time variation in the usage of each space-level combination within one or more selected areas.

At a minimum, by "usage" we mean the amount currently used (in words), distinguishing between list objects (also called *conses*) and structure objects. This information is inexpensively available from the system. Additionally, for the by-area view, the monitor is capable of recording statistics on the types of objects present. This more detailed usage data is obtained by scanning the regions constituting each space-level combination; the overhead in scanning for object-level information is reduced by caching the statistics obtained for a region and, where possible, scanning only the portion of a region that has been used up by new objects since the last time the region was scanned.

2.1.1 Monitoring

The requirements for the monitor are summarized as follows:

- Low time overhead. The execution time of monitored programs should not be unduly lengthened as a result of data collection activity.
- Low space overhead. The rate at which data is collected should be low enough to make it possible to monitor for long periods of time, e.g., programs running for many hours, without requiring the storage of overwhelming amounts of data.

- Flexibility. This refers to being able to select the areas of memory to be monitored for producing by-area views, and the level of detail in the usage data collected for those areas. In the case where statistics on the types of objects in memory are being collected, it should be possible to specify how objects are classified into types. This would make it possible to map objects into meaningful, application-level data types, rather than, or in addition to, language-level types.

- Generality. Monitoring and analysis should be able to handle the occurrence of all types of garbage collections: ephemeral collections (the most frequent), dynamic collections of all or some areas, and full garbage collections.

The approach adopted was to sample information at three distinguished epochs during a garbage collection cycle. These are: before the flip (BF), which defines the start of a cycle; after the flip (AF), which is after all the regions to be collected have been turned into oldspace and scavenging is about to begin; and after oldspace has been reclaimed (ARO), which is when scavenging has finished, no pointers to oldspace exist, and all oldspace regions have been turned into free regions. Basically, garbage collection is "in progress" during the BF-AF-ARO portion of a cycle, and not in progress during the period from ARO to the next BF.

The information sampled at these epochs includes a time stamp; selected paging and garbage collection counters; and data on memory usage as contained in the area and region tables, and, optionally, as obtained by scanning regions for object-level statistics. The paging and garbage collection counters can be classified as either activity timers (e.g., page fault time, scavenging time), event counters (e.g., page fetches), or work counters (e.g., words consed, words scanned, words copied) relative to some arbitrary time in the past. These counters are accessible as Lisp global variables and are maintained by the system. The area and region tables are also maintained by the system; they implement the memory organization shown in Figure 2.1. Not all the information is collected at all three epochs. For example, the garbage collection-related counters are sampled only at ARO, when they reflect the collection just completed. As another example, the sizes of oldspace regions are noted only at AF, since oldspace does not exist at the other epochs.

Figure 2.2 depicts the overall organization of the monitoring and analysis facility. Data logging is accomplished by three functions corresponding to the three interesting epochs; these

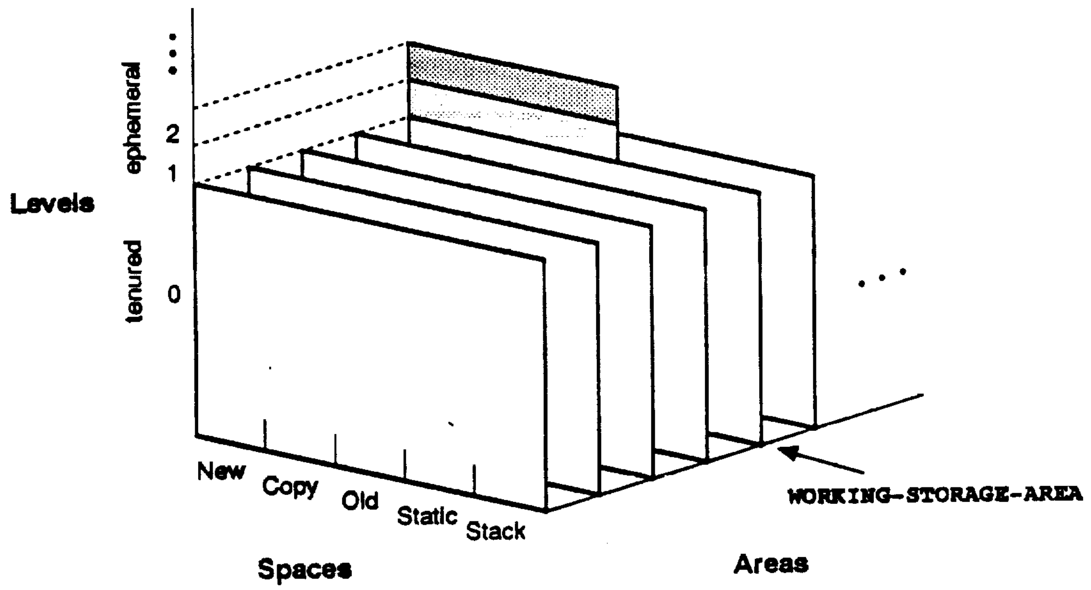

Figure 2.1 Virtual memory organization in the Symbolics Lisp system.

functions are hooked onto the garbage collector to be run at their respective times. The raw data is stored in an object called a *runlog*, which also serves as a repository of all information on the context of a measurement session.

Runlogs can be saved and restored from disk files, and multiple runlogs can exist in memory for comparative analyses. Unlike a trace file, a runlog contains data in structured form, so it is not necessary to "parse" the raw data to uncover its structure. A runlog contains:

- Identification of self (e.g., a name), and the relevant details of the system environment (e.g., main memory size, generation configuration) in effect during the measurement.
- Parameters for effecting user control over the monitoring process, such as for specifying the termination condition, which areas of memory (if any) to monitor, and what kind of memory usage data to collect.
- State variables.
- A global log, which contains the samples of the activity timers, event and work counters, and usage data on all memory (non-area-specific).
- A set of area logs, one for each area being monitored. An area log contains usage data on each space-level combination present in the area.

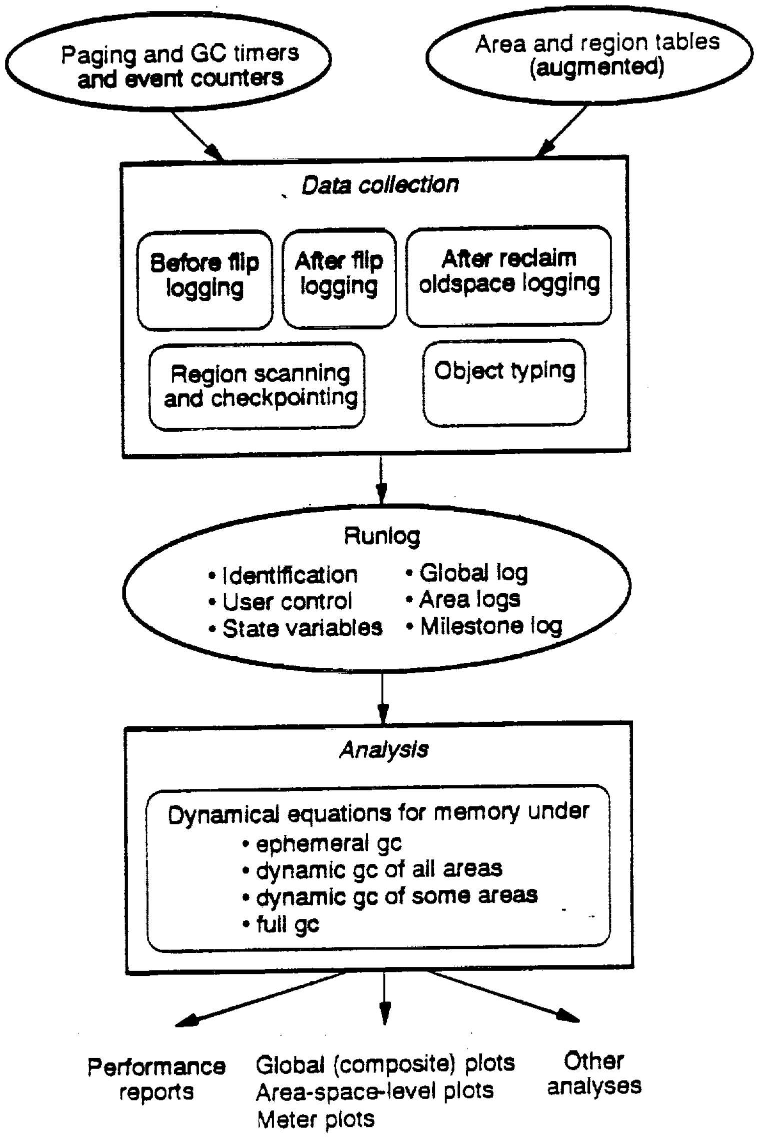

Figure 2.2 Block diagram of dynamic monitoring and analysis facility.

- A set of "milestones," which are essentially time-stamped samples of the counters taken at instants meaningful to a user, such as immediately before and after execution of a top-level function and at major program state changes. This data allows performance to be characterized over an interval whose endpoints are defined at other than the epochs associated with a collection cycle.

2.1.2 Analysis

Analysis of data in a runlog can occur concurrently with monitoring. This is possible since no conflicts arise in writing and reading the data. A trivial kind of analysis involves taking the difference between two counter values, yielding the total amount of time spent in an activity (for example) in the interval between sampling instants.

An example of the kind of performance summary generated is given in Figures 2.3 and 2.4, which result from the monitoring of two large programs. The SRW program is a parser written in REFINE which is then compiled into Lisp. The program was measured while parsing five mainframe assembly language files into a knowledge base. It incurs a modest amount of garbage collection time. The QPE program is a simulator for qualitative process theory. Garbage collection overhead is negligible, but paging is a significant problem.

Options to the reporting function can request that analysis be confined to the the interval from cycle i to $j >= i$, or between any two recorded "milestones."

The sample reports in Figures 2.3 and 2.4 also illustrate an interesting kind of analysis involving the global memory space data. Consider the tables bordered by vertical bars, which provide a breakdown of the memory flipped into oldspace, a breakdown of the amount of surviving objects, and a breakdown of new object allocation. The values in these tables are not directly contained in the raw data, but are derived by solving a system of linear equations involving the samples of memory space usage at the three epochs in a cycle, and some counters, notably the total number of words flipped into oldspace, copied during garbage collection, and consed during a cycle. In general, if S_{bf}, S_{af}, and S_{aro} denote the sizes of a given tenured or ephemeral space at each of the three epochs, the equations for a particular cycle c are of the form

$$S_{\mathrm{af}} = S_{\mathrm{bf}} - \mathit{flipped}_S + \mathit{survived}_{S,\mathrm{bf}-\mathrm{af}} + \mathit{consed}_{S,\mathrm{bf}-\mathrm{af}}$$

$$S_{\mathrm{aro}} = S_{\mathrm{af}} + \mathit{survived}_{S,\mathrm{af}-\mathrm{aro}} + \mathit{consed}_{S,\mathrm{af}-\mathrm{aro}}$$

```
INTERVAL BF(1) TO ARO(86)
  86 gc cycles: 86 ephemeral, 0 dynamic, 0 full

SYSTEM ANALYSIS          Total           |        Mutator                  GC            |      BF-AF-ARO              ARO-BF
  Real time         2:18:48.493 (100.00%) |   2:01:01.938 ( 87.19%)   0:17:46.555 ( 12.81%) | 0:21:23.297 ( 15.41%)  1:57:25.196 ( 84.59%)
    Runtime         1:59:38.141 ( 86.19%) | * 1:41:56.991 ( 73.45%) * 0:17:41.149 ( 12.74%) | 0:13:20.848 (  9.62%)  1:46:17.293 ( 76.57%)
    Page faults     0:09:24.001 (  6.77%) | * 0:09:18.594 (  6.71%) * 0:00:05.406 (  0.06%) | 0:02:57.944 (  2.14%)  0:06:26.057 (  4.64%)
    Page creation   0:04:37.934 (  3.34%) |             -                       -           | 0:03:29.547 (  2.52%)  0:01:08.387 (  0.82%)
    Other           0:05:08.418 (  3.70%) |             -                       -           | 0:01:34.959 (  1.14%)  0:03:33.459 (  2.56%)
                                          |                                                 |
  Page faults             9,702 (100.00%) |         9,609 ( 99.04%)            93 (  0.96%) |         444 (  4.58%)        9,258 ( 95.42%)
  Page creations        119,796 (100.00%) |             -                       -           |      50,347 ( 42.03%)       69,449 ( 57.97%)

GC ANALYSIS            GC real time          GC page faults      Words scanned
  Scavenging        0:17:45.824 ( 99.93%)        88 ( 94.62%)   178,969,781 (100.00%)
    GCPT            0:10:15.880 ( 57.74%)         5 (  5.38%)   155,642,368 ( 86.97%)
    ESRT            0:00:18.685 (  1.75%)         6 (  6.45%)     2,328,320 (  1.30%)
    First pass      0:02:36.071 ( 14.63%)        18 ( 19.35%)     8,322,079 (  4.65%)
    Final pass      0:04:35.189 ( 25.80%)        59 ( 63.44%)    12,677,014 (  7.08%)
  Mutator transp.   0:00:00.731 (  0.07%)         5 (  5.38%)

  Scanning          0:11:05.982 ( 62.44%)        40 ( 43.01%)   178,969,781 scanned
  Transporting      0:06:40.573 ( 37.56%)        53 ( 56.99%)    12,674,506 transported

GC EFFICIENCY
  Oldspace            24,461,761 (100.00%) | During         Words flipped into oldspace from                    |
                                           |          Ephe new   Ephe copy    Lev0 new   Lev0 copy      Total |
                                           | BF-AF  17,561,464   6,900,297           0           0 24,461,761 |

    Survived          12,674,506 ( 51.81%) | During        Words surviving from oldspace into                   |
                                           |          Ephe new   Ephe copy    Lev0 new   Lev0 copy      Total |
                                           | BF-AF           -       1,126           0       3,620      4,746 |
                                           | AF-ARO          -   6,927,848           0   5,741,912 12,669,760 |
                                           | Total           -   6,928,974           0   5,745,532 12,674,506 |
    Garbage           11,787,255 ( 48.19%)
  Work units         161,262,978 excluding 30,381,309 redundant scans
  Time efficiency      11051.704 words/sec        (  0.090 msec/word)
  Work efficiency          0.073 words/work-unit ( 13.681 work-units/word)

MEMORY USAGE
  Consed             17,843,417              | During         Words consed in                  Computed    Measured |
                                             |          Ephe new    Lev0 new      Static       Total       Total |
                                             | BF-AF          462           0           0         462          14 |
                                             | AF-ARO       4,940       2,508       1,280       8,728       7,448 |
                                             | ARO-BF  17,355,942      12,179     467,900  17,836,021  17,835,955 |
                                             | Total   17,361,344      14,687     469,180  17,845,211  17,843,417 |
  Net vm change       6,056,162
  Max vm size        21,467,598 at ARO(86)
```

Figure 2.3 Performance summary for SRW running on a Symbolics 3650 under Genera 7.2, 3 Mwords RAM, default generation configuration of two ephemeral levels with capacities of 200×10^3 (youngest level) and 100×10^3 words.

```
INTERVAL BF(1) TO ARO(162)
  162 gc cycles: 162 ephemeral, 0 dynamic, 0 full

SYSTEM ANALYSIS             Total           |        Mutator                   GC          |      BF-AF-ARO                ARO-BF
  Real time           43:08:20.188 (100.00%) |   42:33:15.109 ( 98.64%)   0:35:05.079 (  1.36%) | 0:46:02.094 (  1.78%)  42:22:18.094 ( 98.22%)
    Runtime           16:15:25.266 ( 37.69%) | * 15:42:47.320 ( 36.42%) * 0:32:37.963 (  1.26%) | 0:28:29.934 (  1.10%)  15:46:55.332 ( 36.58%)
    Page faults       25:12:40.930 ( 58.44%) | * 25:10:13.813 ( 58.35%) * 0:02:27.116 (  0.09%) | 0:10:40.065 (  0.41%)  25:02:00.867 ( 58.03%)
    Page creation      0:05:25.399 (  0.21%) |          -                       -               | 0:03:02.393 (  0.12%)   0:02:23.006 (  0.09%)
    Other              1:34:48.578 (  3.66%) |          -                       -               | 0:03:49.703 (  0.15%)   1:30:58.875 (  3.52%)

  Page faults            4,250,673 (100.00%) |     4,243,783 ( 99.84%)         6,890 (  0.16%) |      14,341 (  0.34%)     4,236,332 ( 99.66%)
  Page creations           190,806 (100.00%) |          -                       -               |      61,532 ( 32.25%)       129,274 ( 67.75%)

GC ANALYSIS              GC real time         GC page faults      Words scanned
  Scavenging          0:34:59.139 ( 99.72%)   6,697 ( 97.20%)   210,922,273 (100.00%)
    GCPT              0:11:06.805 ( 31.68%)   2,491 ( 36.15%)   168,269,312 ( 79.78%)
    ESRT              0:05:04.688 ( 14.47%)     742 ( 10.77%)    19,479,296 (  9.24%)
    First pass        0:08:02.235 ( 22.91%)   1,116 ( 16.20%)     7,641,622 (  3.62%)
    Final pass        0:10:45.410 ( 30.66%)   2,348 ( 34.08%)    15,532,043 (  7.36%)
  Mutator transp.     0:00:05.940 (  0.28%)     193 (  2.80%)

  Scanning            0:22:11.239 ( 63.24%)   5,838 ( 84.73%)   210,922,273 scanned
  Transporting        0:12:53.840 ( 36.76%)   1,052 ( 15.27%)    15,531,114 transported

GC EFFICIENCY
  Oldspace             41,000,762 (100.00%) | During        Words flipped into oldspace from                  |
                                            |          Ephe new    Ephe copy   Lev0 new   Lev0 copy      Total |
                                            | BF-AF    32,996,568  8,004,194          0           0  41,000,762 |

    Survived           15,531,114 ( 37.88%) | During        Words surviving from oldspace into                |
                                            |          Ephe new    Ephe copy   Lev0 new   Lev0 copy      Total |
                                            | BF-AF           -          915          0       3,850       4,765 |
                                            | AF-ARO          -    7,978,035          0   7,548,314  15,526,349 |
                                            | Total           -    7,978,950          0   7,552,164  15,531,114 |
    Garbage            25,469,648 ( 62.12%)
  Work units          185,902,233 excluding 40,551,154 redundant scans
  Time efficiency       12099.142 words/sec        ( 0.083 msec/word)
  Work efficiency           0.137 words/work-unit  ( 7.299 work-units/word)

MEMORY USAGE
  Consed               33,045,280           | During           Words consed in               Computed     Measured |
                                            |          Ephe new     Lev0 new    Static       Total        Total |
                                            | BF-AF          398          0         0          398           74 |
                                            | AF-ARO       6,981        914     1,792        9,687        7,895 |
                                            | ARO-BF  32,788,738     41,275   206,976   33,036,989   33,037,311 |
                                            | Total   32,796,117     42,189   208,768   33,047,074   33,045,280 |

  Net vm change         7,575,632
  Max vm size          19,163,730 at ARO(161)
```

Figure 2.4 Performance summary for QPE running on a Symbolics 3650 under Genera 7.2, 3 Mwords RAM, default generation configuration of two ephemeral levels with capacities of 200×10^3 (youngest level) and 100×10^3 words.

ORIGINAL PAGE IS
OF POOR QUALITY

$$\textit{total flipped} = \sum_{S} \textit{flipped}_S$$

$$\textit{total copied} = \sum_{S} (\textit{survived}_{S,\text{bf}-\text{af}} + \textit{survived}_{S,\text{af}-\text{aro}})$$

$$\textit{total consed} = \sum_{S} (\textit{consed}_{S,\text{bf}-\text{af}} + \textit{consed}_{S,\text{af}-\text{aro}})$$

where *flipped* refers to the unknown amount of the given memory space flipped into oldspace, and *survived* and *consed* refer, respectively, to the unknown number of words copied and allocated *into* the given space. Some of the terms may be zero, depending on the particular memory space and kind of garbage collection. A set of these "dynamical equations for memory" can be written for each kind of garbage collection and solved to derive information that is otherwise not directly maintained by the system.

2.2 Graphical Representations

We now present some examples of graphical representations of the memory usage data.

Figure 2.5 is a global plot of memory usage for the workload consisting of 60 iterations of the `Boyer` benchmark under the default configuration of two ephemeral levels with capacities of 200×10^3 (youngest) and 100×10^3 words. This plot is obtained by stacking up, in order from top to bottom:

- ephemeral oldspace
- ephemeral newspace
- ephemeral copyspace
- level 0 oldspace
- level 0 copyspace
- level 0 newspace
- level 0 static space
- level 0 stack space

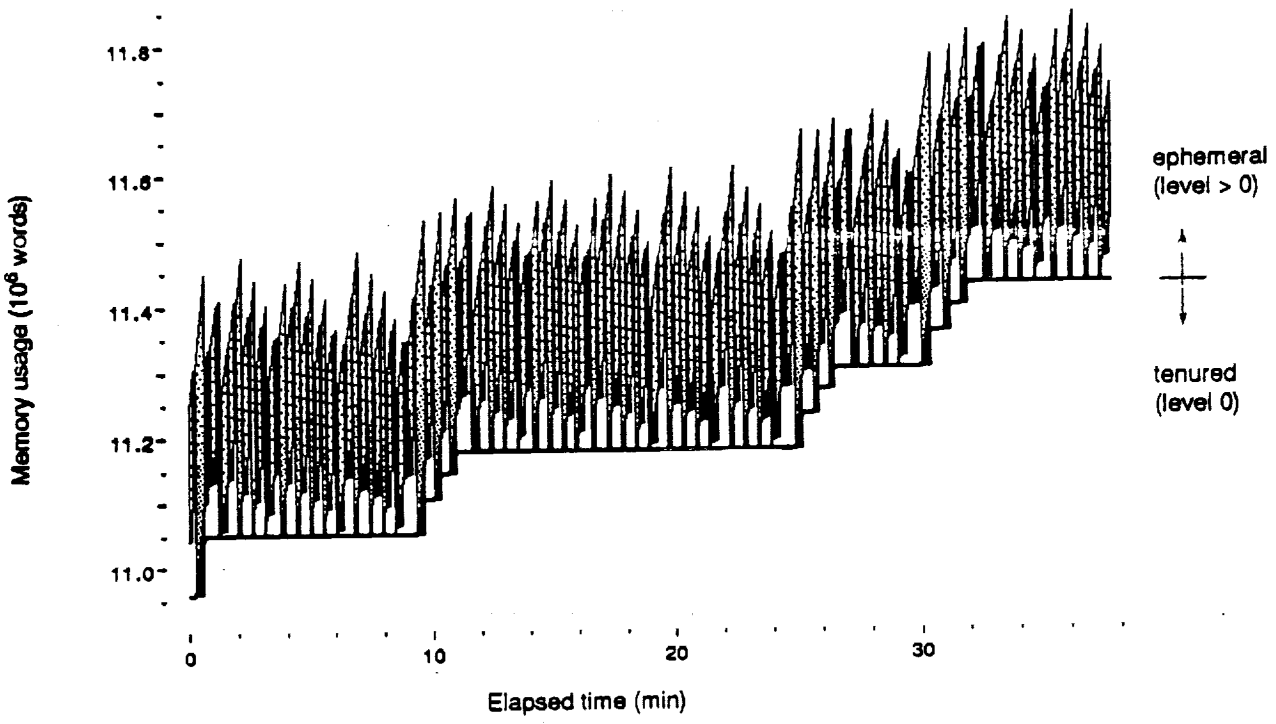

Figure 2.5 Global view of memory usage for Boyer, default generation configuration.

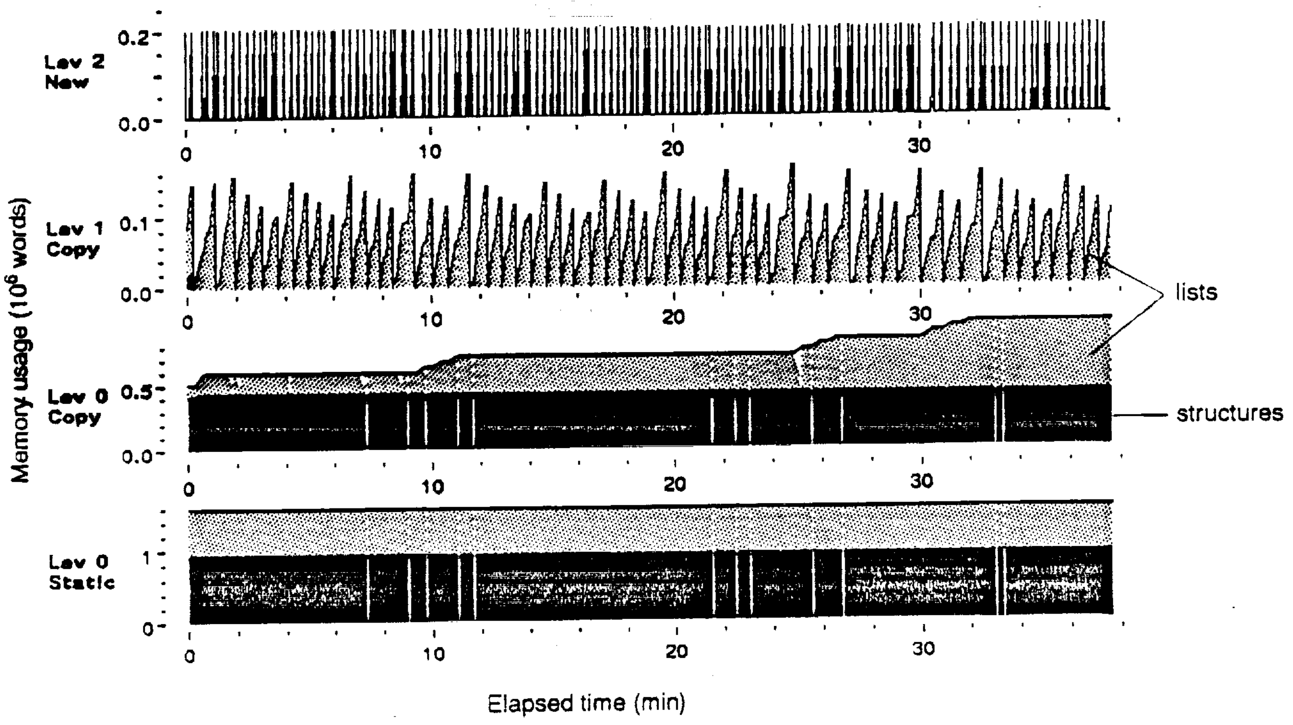

Figure 2.6 Area view of WORKING-STORAGE-AREA for Boyer.

Figure 2.7 shows a magnified view of the global plot in the vicinity of 10 elapsed minutes. Here, the horizontal axis is calibrated in cycles, to make it somewhat easier to "read." The thick line represents the boundary between the ephemeral and level 0 layers. Only the ephemeral layers are shown in their entirety. The *changes* in a given layer are highlighted by triangles—shaded medium gray for ephemeral newspace and black for both copyspace layers. Note that newspace changes represent object allocation while changes in copyspace represent survival (and garbage collector copying work). Almost all collections of level 1 result in no survivors, with the exception of a few cycles (e.g., cycles 36, 38, and 40) during which some objects are tenured.

Figures 2.6 and 2.8 are area plots for `WORKING-STORAGE-AREA`. The shading indicates that all the objects created by `Boyer` in this area are lists (conses).

Figures 2.9 and 2.10 show the global and area plots for `Boyer` under a configuration of only one level with a capacity of 2.4×10^6 words. Execution time is significantly reduced from 38 to about 15 minutes. However, a larger amount of tenuring (and tenured garbage) occurs.

By having a large number of ephemeral levels, and changing the collection policy such that every level flips when the youngest level flips,[1] we effectively configure the ephemeral levels as a shift register, with the population of objects created during a cycle being moved down by one level on each successive cycle. This configuration will explicitly show the lifetime distribution of each object population. We call the resulting memory usage plot a *chroma*.[2]

The chroma for the first 10 iterations of `Boyer` is shown in Figure 2.11. It shows that all the objects created during a single iteration become unreachable by the end of the iteration. The generation configurations in Figures 2.5–2.10 are therefore suboptimal with respect to memory utilization because they allow some objects which will soon become garbage to be tenured. The reason for this leakage is that neither configuration provides a sufficient minimum age for objects tenured. In the two-level configuration, the minimum age is 200×10^3 words (measuring time in words allocated); an object with this age that survives a collection of level 1 during cycle c will have been created just before cycle $c-1$. In the one-level configuration, the minimum age is 0 words. To avoid any tenuring, `Boyer` requires a generation configuration which guarantees a minimum age of $\frac{26,890,965}{60} \approx 450 \times 10^3$ words.

[1] This policy is accomplished easily by setting the capacity of each ephemeral non-top level to zero.

[2] By analogy with the technique of chromatography which analyzes an unknown substance by observing how far each component propagates along a medium. Here we are interested in measuring an object's lifetime by observing how many generations it survives before becoming garbage.

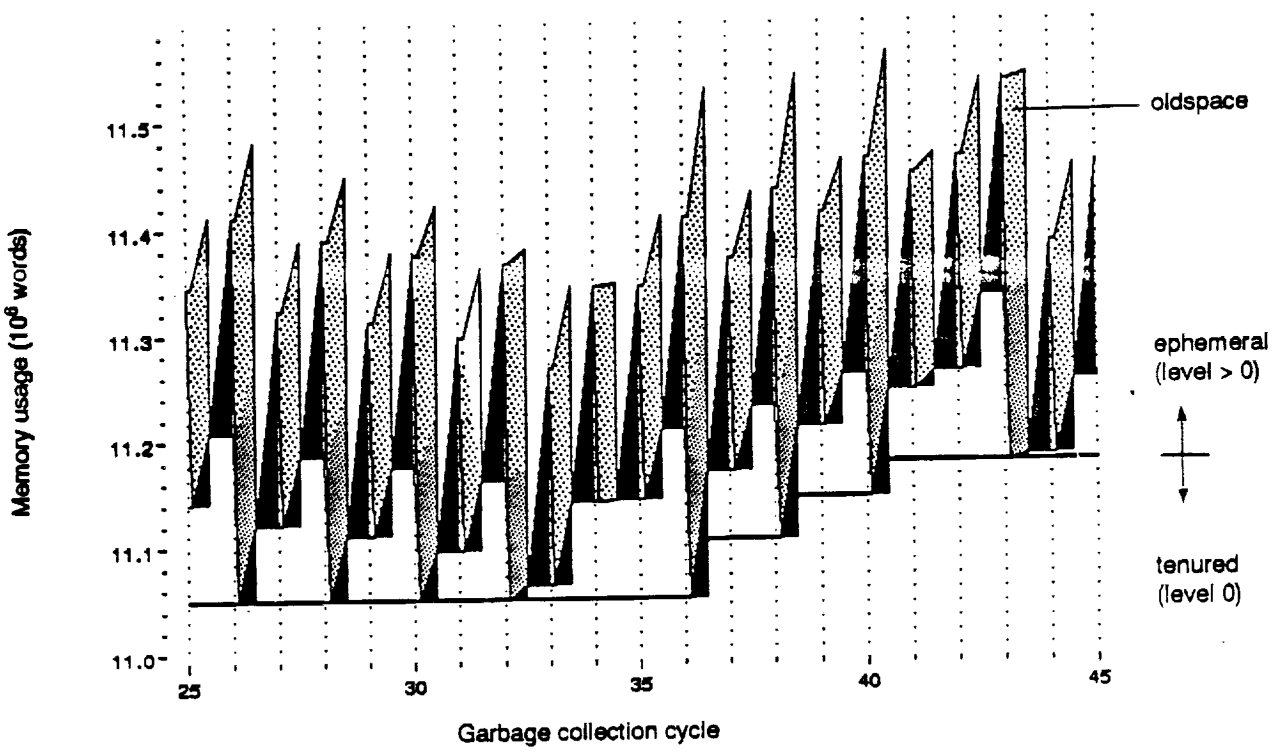

Figure 2.7 Global view of memory usage for **Boyer** (magnified).

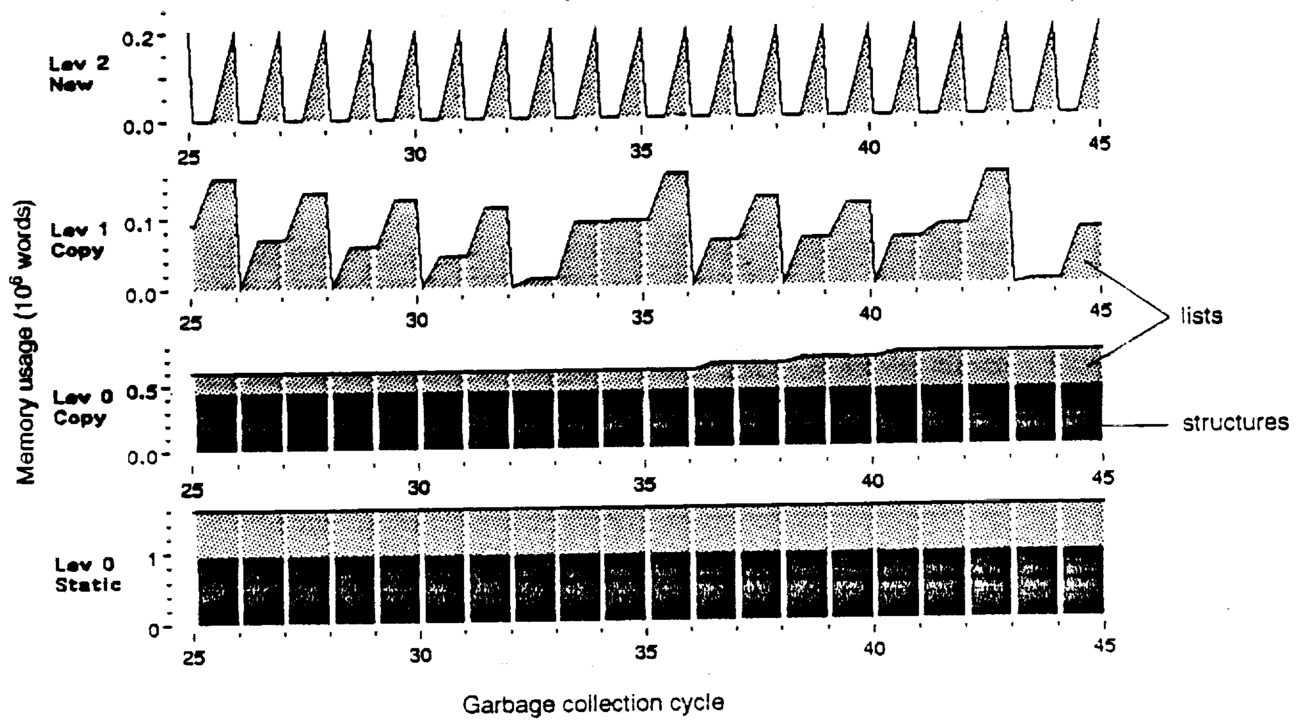

Figure 2.8 Area view of `WORKING-STORAGE-AREA` for **Boyer** (magnified).

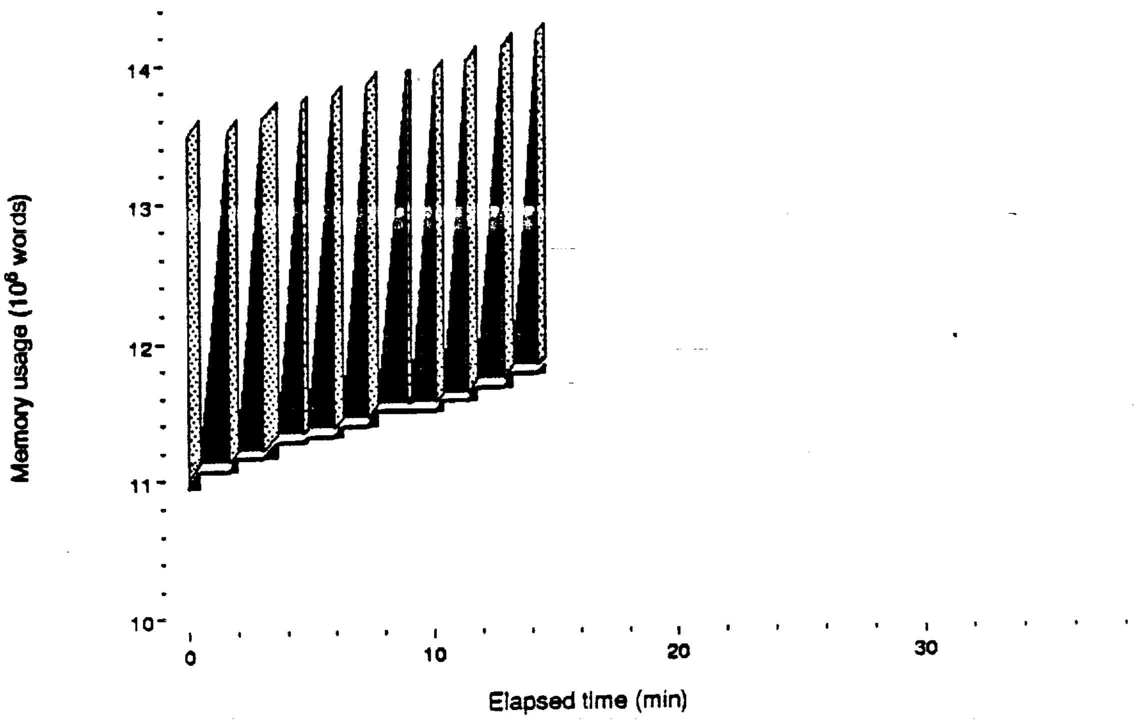

Figure 2.9 Global view of memory usage for **Boyer** under a generation configuration of one level with capacity 2.4 × 10^6 words.

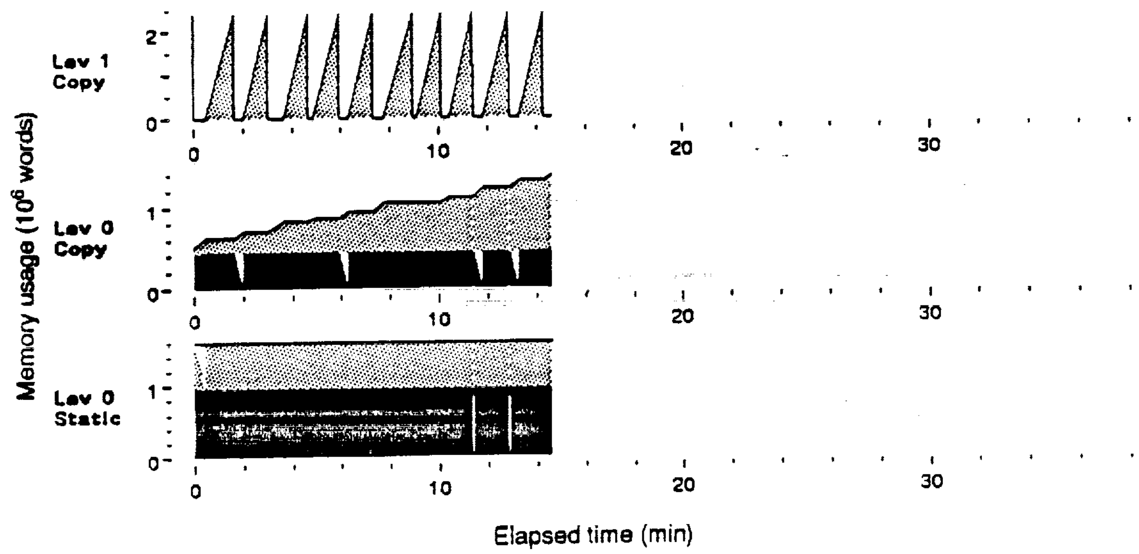

Figure 2.10 Area view of `WORKING-STORAGE-AREA` for **Boyer** under a generation configuration of one level with capacity 2.4 × 10^6 words.

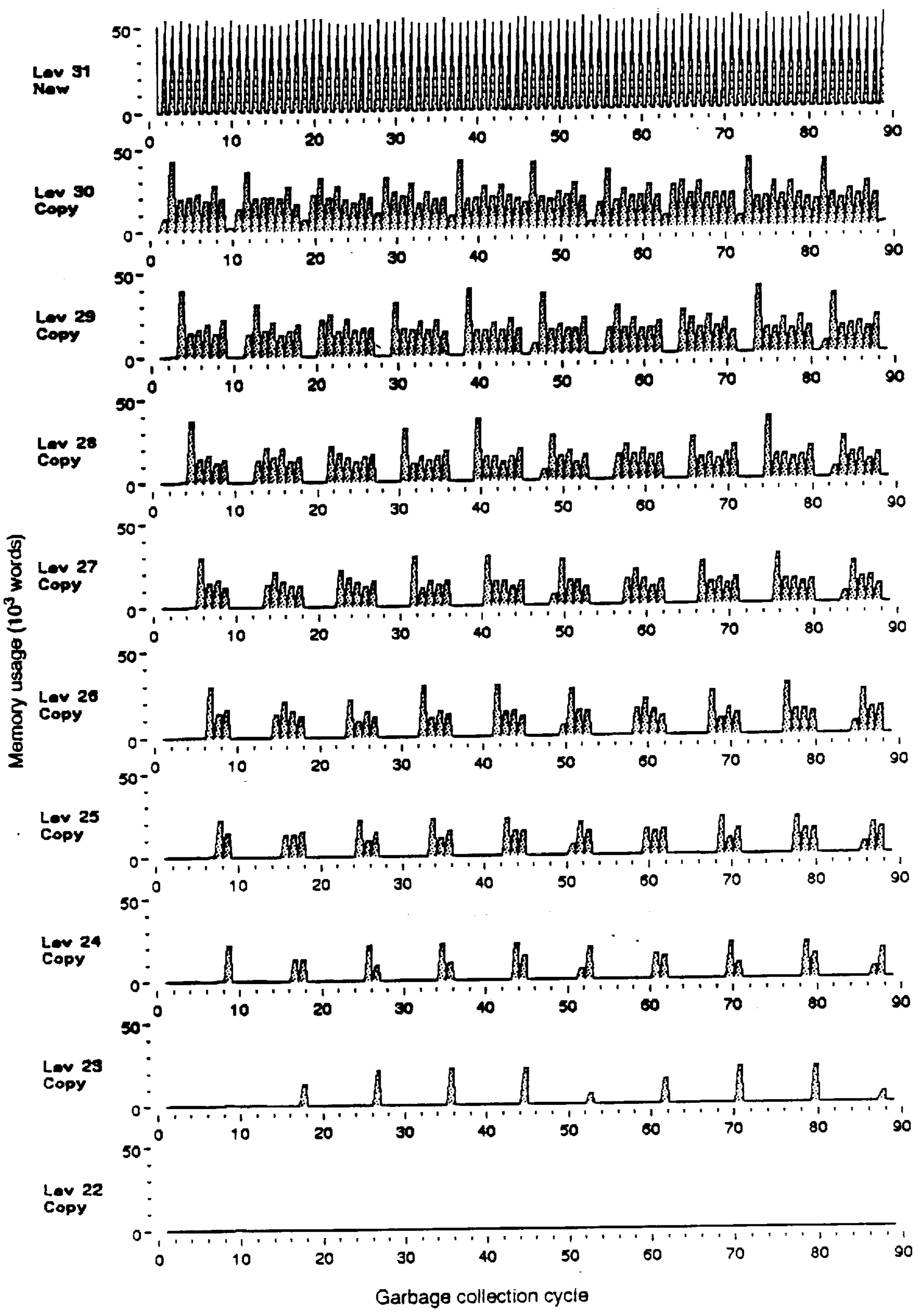

Figure 2.11 Boyer chroma at a resolution of 50×10^3 words.

The chroma for SRW is shown in Figure 2.12. The results show the presence of objects of long lifetime, medium lifetime (created during the second half), and short lifetime (reclaimed after one cycle). Furthermore, we observe a high percentage of long-lived data at the start of reading in each input file. This data consists of the text in the file and is a good candidate for creation in tenured space, so that collection work will not be wasted in copying it.

2.3 Summary

A software facility for collecting, analyzing, and visualizing memory usage and performance data on the Symbolics Lisp system has been developed. The facility records a history of memory usage and performance by drawing on existing sources of data, in particular, the various counters maintained by the system for performance metering purposes, and the memory tables maintained by the system for memory management purposes. Data collection is synchronized with garbage collection so that the abrupt transitions in the state of memory and the peaks of memory usage associated with the distinguished epochs during a collection cycle are always detected. The low time and space overhead of the instrumentation makes it suitable for nonintrusive monitoring of applications running for long periods of time.

Many current Lisp systems provide a function profiler, to help in identifying the most time-consuming pieces of code. A facility for memory usage profiling and performance evaluation, such as has been described in this chapter, is a useful addition to the set of performance measurement tools available to the user. While our instrumentation is specific to the Symbolics memory organization and garbage collector, it should be possible to add a similar data collection and analysis facility to other Lisp implementations.

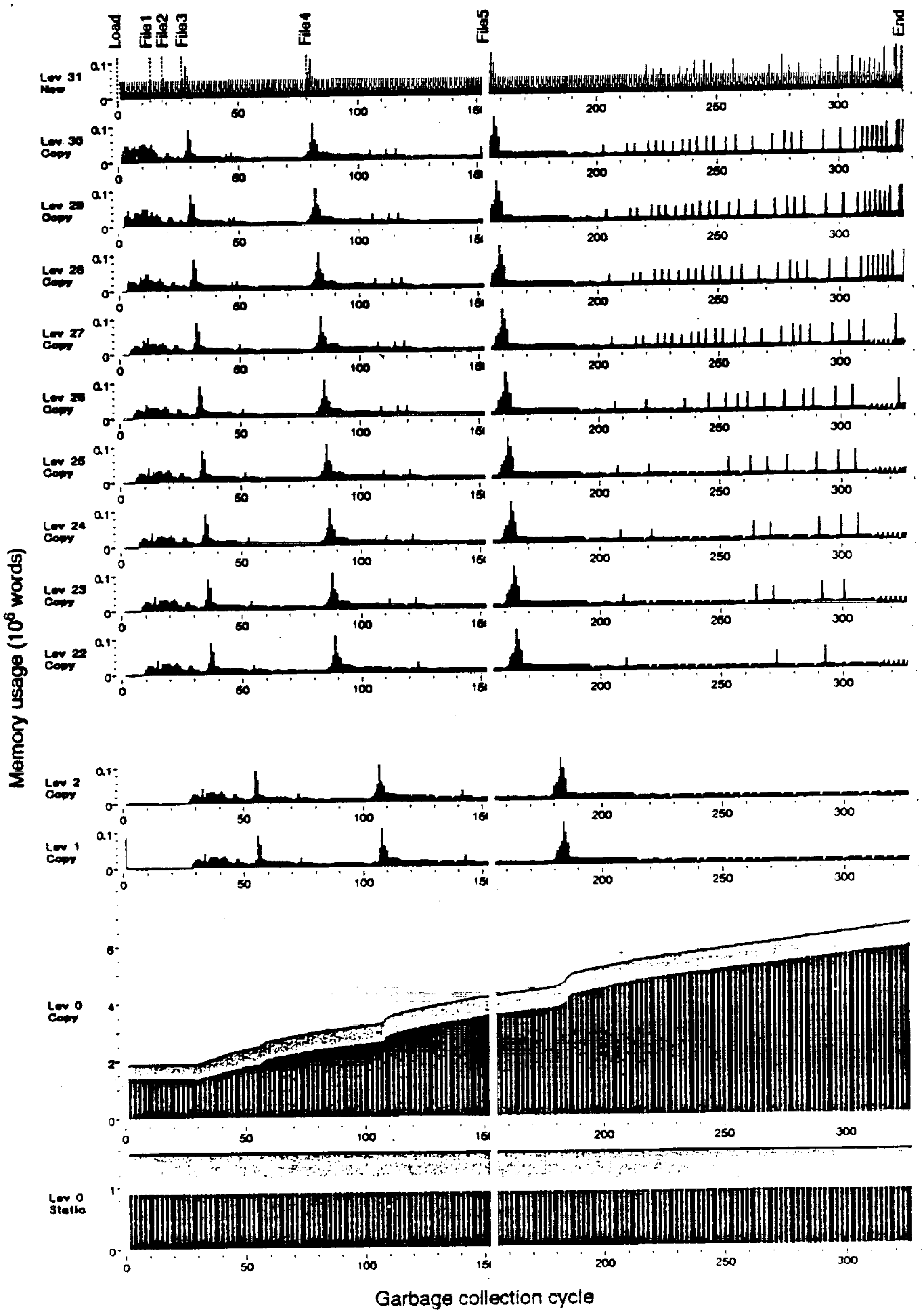

Figure 2.12 SRW chroma at a resolution of 50 × 10^3 words.

ORIGINAL PAGE IS OF POOR QUALITY

Chapter 3

Analysis of Generation-based Garbage Collection

A generation-based garbage collector in a virtual memory system, such as the ephemeral garbage collector, works because of the high mortality among newly created objects and because references from older to newer objects are created relatively infrequently [14, 24, 15, 7, 18, 25, 19, 2, 27, 28].

An important problem is that of optimizing such a collector to match the characteristics of the application to improve its efficiency and overall system performance. In the terminology of ephemeral garbage collection, the optimization problem, in its most general (and of course intractable) form, involves determining the number of ephemeral levels, and deciding which levels to collect, when to collect them, and to which level to move surviving objects. Collectively, these decisions determine the space-time configuration of the collector and represent a choice of *policy*.

An "optimal" policy is a compromise between conflicting considerations. For example, consider the ephemeral garbage collector. Unless manipulated otherwise, the normal behavior of this collector is to follow a first-level-triggered, capacity-based initiation and unconditional promotion policy. By this we mean

(1) collections are started when the occupancy of the youngest level exceeds a threshold value, called its capacity;

(2) all levels from the youngest through level l are then collected, where l is the oldest level such that all levels from the youngest through l inclusive have exceeded their respective capacities; and

(3) surviving objects are promoted to the next older level or tenured in normal dynamic space if already at the oldest ephemeral level.

The degrees of freedom we can exercise within this policy subspace are the number of levels, and the capacity of each level. Increasing the number of levels reduces the rate of creation of tenured garbage (objects that become garbage after graduating past all levels), thereby further postponing a time-consuming full garbage collection, but increases the amount of copying work for long-lived objects. Increasing the capacity of the first level allows more time for new objects to die, thereby increasing the efficiency of collection and reducing tenured garbage, but reduces locality of reference by causing memory to be compacted less frequently.

The best balance among these constraints depends on program and system characteristics and on our performance objectives. For example, we may be interested in minimizing the total execution time for a particular program. We may be interested in postponing a full garbage collection for as long as possible. We may be interested in maximizing the average execution rate for an "infinite" program spanning many full garbage collections.

In this chapter, we describe analysis and measurements which have been conducted to understand the various factors involved for the circumscribed policy subspace described above.

3.1 Timing Model

Given a benchmark program which executes over many garbage collection cycles, the total execution time for the program is the sum of the time taken by the mutator and the time taken by garbage collection. Mutator time and garbage collection time can each in turn be divided into a runtime (nonpaging) and a paging component,

$$T_{total} = T_{mutator,run} + T_{mutator,pag} + T_{gc,run} + T_{gc,pag} \qquad (3.1)$$

We assume $T_{mutator,run}$ to be invariant with respect to garbage collector parameters. Overhead due to garbage collection runtime is represented by $T_{gc,run}$. Program characteristics affecting this overhead are

- lifetime of objects,
- allocation rate, and

- connectivity of objects.

Allocation rate influences the frequency with which garbage collection will have to be invoked. Object lifetime influences the amount of copying work (and therefore part of the scanning work) which the garbage collector needs to perform. Object connectivity, in particular, the frequency (in space) of pointers from older to younger objects, influences the size of the ephemeral root set, and hence the amount of scanning work which needs to be done.

The runtime component of garbage collection $T_{gc,run}$ can be modelled in terms of the scanning and transporting work performed. Assume that the program allocates a total of M words over its execution. For simplicity, assume that we have only one ephemeral level, i.e., any objects surviving a garbage collection of this level will be tenured. Let the capacity of this level be C_0. What is the optimum value of C_0?

A model for the total garbage collector runtime is

$$T_{gc,run} = \frac{M}{C_0}[k_{scan}(W_{rootset} + W_{transp}) + k_{transp}W_{transp}] \tag{3.2}$$

where

M/C_0	number of garbage collection cycles
$W_{rootset}$	average size of the ephemeral root set (in words), which is scanned on each cycle
W_{transp}	average number of words transported on each cycle
k_{scan}, k_{transp}	machine-specific constants representing the average time per word scanned and transported, respectively.

Note that the objects which are transported also have to scavenged, since they could contain pointers to oldspace. Also, it is important to keep in mind the distinction between *total* variables such as $T_{gc,run}$ and *per cycle* variables such as $W_{rootset}$. This model for $T_{gc,run}$ does not express the complexities associated with the way the ephemeral root set is maintained and scanned nor does it account for overhead in the scanning and transporting routines, except by amortizing it over the actual number of words scanned and transported. However, the model is simple and sufficient for our purposes.

The number of words transported per cycle, W_{transp}, can be expressed in terms of program characteristics (lifetime and allocation rate) and garbage collector configuration (C_0). The next section discusses the relationship.

3.2 Survival Model

Let $S(x) = P(X > x)$ be the distribution of object lifetime X. The lifetime of an object is the time from its creation to the instant that it becomes inaccessible (garbage). Note that $S(x)$ is the *survival function* and is equivalent to other forms of specifying distribution, e.g., probability density function or cumulative distribution function. While it is possible to assume some average allocation rate, or to assume a distribution for the allocation rate, we find it convenient instead to assume that time, for purposes of expressing object lifetime X, is measured not in terms of seconds, but in words allocated. Hence $S(x)$ is a joint description of object lifetime and allocation characteristics.

Consider the set of all objects created during the time (measured in words allocated) interval (x_1, x_2). At some later time $x_{eval} \geq x_2$, we would like to know the state of this population of objects. It is easily shown that the expected number of words surviving at time x_{eval} is given by the function

$$U(a,b) = \int_a^b S(x)dx \tag{3.3}$$

where $a = x_{eval} - x_2$ and $b = x_{eval} - x_1$. Thus, $U(a,b)$ has the meaning of the expected amount of words surviving from the allocation that occurred during the interval between a words ago and b words ago.

This result can be applied to determining the expected amount of objects surviving garbage collection. Consider the general case in which we can have an arbitrary number of ephemeral levels. For mathematical convenience, we will number these levels starting from 0 for the youngest level and using successive integers to represent older levels. To avoid confusion with the actual numbering system for levels in the Symbolics system (which is the reverse, i.e., level 0 is the oldest), we will use the term generations to imply the youngest-is-0 numbering system.

If the capacity of generation i is C_i, then generation 0 will be garbage collected every C_0 words allocated, and the expected amount surviving each such collection is $L_0 = U(0, C_0)$. Hence, generation 1 receives an input of L_0 words from generation 0 every C_0 words allocated. Generation 1 will be garbage collected on average every $\lceil C_1/L_0 \rceil$ such inputs. The amount of words that generation 1 will promote to generation 2, and similar quantities for higher generations can also be derived in terms of the $U(a,b)$ function (See Table 3.1).

Table 3.1 Frequency of collection of each generation and expected survival.

Gen.	Time between inputs	Input periods between GCs	Amount promoted on each GC
0	continuous alloc.		$L_0 = U(0,\ T_1)$
1	$T_1 = C_0$	$n_1 = \left\lceil \frac{C_1}{L_0} \right\rceil$	$L_1 = U(T_1,\ (1+n_1)T_1)$
2	$T_2 = n_1 T_1$	$n_2 = \left\lceil \frac{C_2}{L_1} \right\rceil$	$L_2 = U((1+n_1)T_1,\ (1+n_1+n_1 n_2)T_1)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	$T_i = T_1 \prod_{k=1}^{i-1} n_k$	$n_i = \left\lceil \frac{C_i}{L_{i-1}} \right\rceil$	$L_i = U(T_1(\sum_{k=0}^{i-1} \prod_{m=1}^{k} n_m),\ T_1 \sum_{k=0}^{i} \prod_{m=1}^{k} n_m)$

While the development up to this point has not assumed a particular form for $S(x)$, e.g., exponential, expressing the expected amount of survival from a particular generation in terms of the $U(a, b)$ function already provides useful information. The arguments a and b define bounds on the *age* of the objects surviving a particular generation. For example, the amount surviving a garbage collection of generation 0 is $U(0, C_0)$. This means that the minimum age for surviving objects is 0 and the maximum age is C_0, as can be verified by a little thought. This suggests that a one-ephemeral-level configuration may tenure short-lived objects prematurely (in particular, the ones created just before the flip) since it does not provide a (nonzero) *minimum guarantee of age*. A two-ephemeral-level configuration guarantees that tenured objects will have an age of at least C_0.

Returning to the timing model for $T_{gc,run}$ for the case of a one ephemeral level configuration (Equation (3.2)), the number of words transported per cycle can therefore be expressed in terms of program characteristics (lifetime and allocation rate) and garbage collector configuration (C_0) simply as $W_{transp} = U(0, C_0)$.

The behavior of $T_{gc,run}$ is now clear.

- Since $U(0, C_0) = \int_0^{C_0} S(x)dx$ and since $S(x)$ is a survival function (i.e., $0 \leq S(x) \leq 1$ and $S(x)$ is monotonic nonincreasing with x), W_{transp} can grow no faster than C_0. More specifically, if α is the fraction of allocation which is long-lived (i.e., $\lim_{x\to\infty} S(x) = \alpha$), then W_{transp} will grow no faster than αC_0.

- A first-level capacity of C_0 effectively defines a cut across the object graph, with objects in the first level on one side of the cut, and tenured objects on the other side. The ephemeral root set consists of the pages in tenured space containing pointers into the first level. As C_0 increases,

- the amount of newly allocated objects in the first level at garbage collection time increases,
- the amount of tenured garbage decreases,
- the contribution to $W_{rootset}$ due to pointers from tenured garbage objects decreases, and
- the contribution to $W_{rootset}$ due to pointers from tenured nongarbage objects *could* increase. This increase could be proportional to C_0 in the worst case.

In our experiments, $W_{rootset}$ in fact decreases with C_0 (Section 3.3, Figure 3.5).

Hence, $T_{gc,run}$ is a nonincreasing (and, practically speaking, a decreasing) function of C_0. That is, according to the timing and survival model above, total garbage collector runtime decreases with increasing first level capacity. The lowest $T_{gc,run}$ should be realized at infinite C_0, i.e., when garbage collection is turned off.

This analytical conclusion is indeed verified experimentally. However, the total execution time also includes paging, which we have heretofore ignored. As the following section will show, paging can increase with C_0 and if so, the resulting tradeoff between decreasing $T_{gc,run}$ and increasing paging time defines an optimum range of values for C_0.

3.3 Experimental Results

To determine the effects of number of ephemeral levels and level capacities, we ran several programs under controlled conditions using various generation configurations. The configurations used were: one level with various first-level capacities; and two levels with various first- and second-level capacities. A Symbolics 3650 with 3 Mwords of main memory was used. Some experiments were also run under reduced main memory sizes of 2 and 1 Mword, but the essential observations remained unchanged. The test programs included the Boyer benchmark [12], short- and long-running versions of QPE, and SRW (described in Section 2.1.2).

The following observations can be made from the results.

Execution time is most sensitive to the capacity of the first level. Thus, the decision of how many ephemeral levels to have beyond the first, and what capacities to specify for these levels, is more important from the point of view of guaranteeing minimum tenuring age (thereby avoiding tenuring of intermediate lifetime objects), rather than of execution time performance. For a given generation configuration, bounds on tenuring age can be computed easily from the arguments to the U function as shown in Table 3.1.

The various programs exhibited a variety of behaviors. At one extreme, the small programs, `Boyer` and `QPE-short`, were similar in that both had negligible paging time, and hence total execution time was primarily the sum of mutator runtime and garbage collector runtime, i.e., $T_{total} \approx T_{mutator,run} + T_{gc,run}$. Furthermore, increasing the capacity of the first level resulted in decreasing $T_{gc,run}$ (and thus T_{total}), and this trend continued indefinitely, such that the best performance (lowest execution time) was achieved at infinite first-level capacity—effectively, when garbage collection was turned off. It is perhaps no accident that garbage collection is often inhibited when running small programs for benchmarking purposes.

At another extreme, `QPE-long` was seen to have a very high amount of $T_{mutator,pag}$, and tuning generation configuration did little to improve performance. For paging-bound workload such as this, in which mutator locality is the primary problem, the appropriate course of action is to consider other techniques such as statically or dynamically reorganizing objects within pages (Chapter 4), modifying the algorithms or data structures employed, or increasing main memory size.

The `SRW` program was observed to lie between these extreme categories and to exhibit interesting tradeoff characteristics. In the remainder of this chapter, we examine the results from this program in greater detail to understand the reasons for the observed behavior.

3.3.1 One ephemeral level configuration

First, consider the one ephemeral level case. Figures 3.1–3.6 all pertain to this configuration.

The plot of total execution time vs. capacity (Figure 3.1) shows the existence of an optimum capacity which is some fraction of main memory size. The range of near-optimum values is seen to depend on main memory size, with a larger size affording a broader range.

At small capacities, when collections are very frequent, the rise in T_{total} is caused by a rise in $T_{gc,run}$ and in $T_{mutator,pag}$ (Figure 3.2).

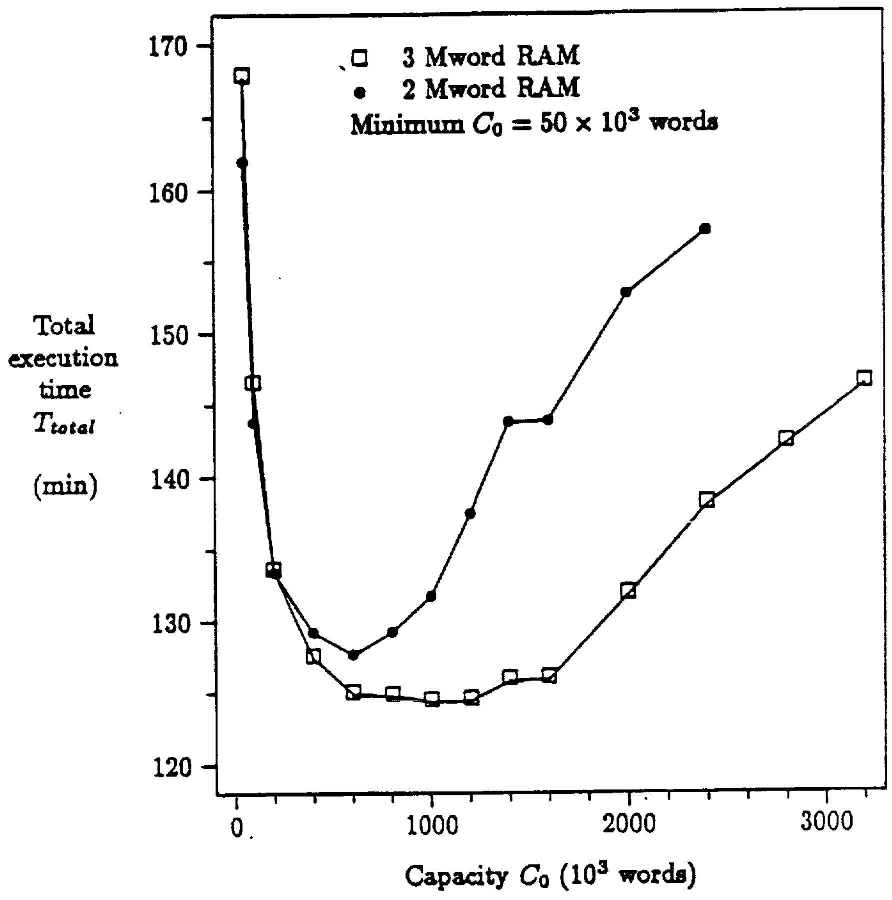

Figure 3.1 SRW total execution time vs. first-level capacity for a generation configuration of one ephemeral level.

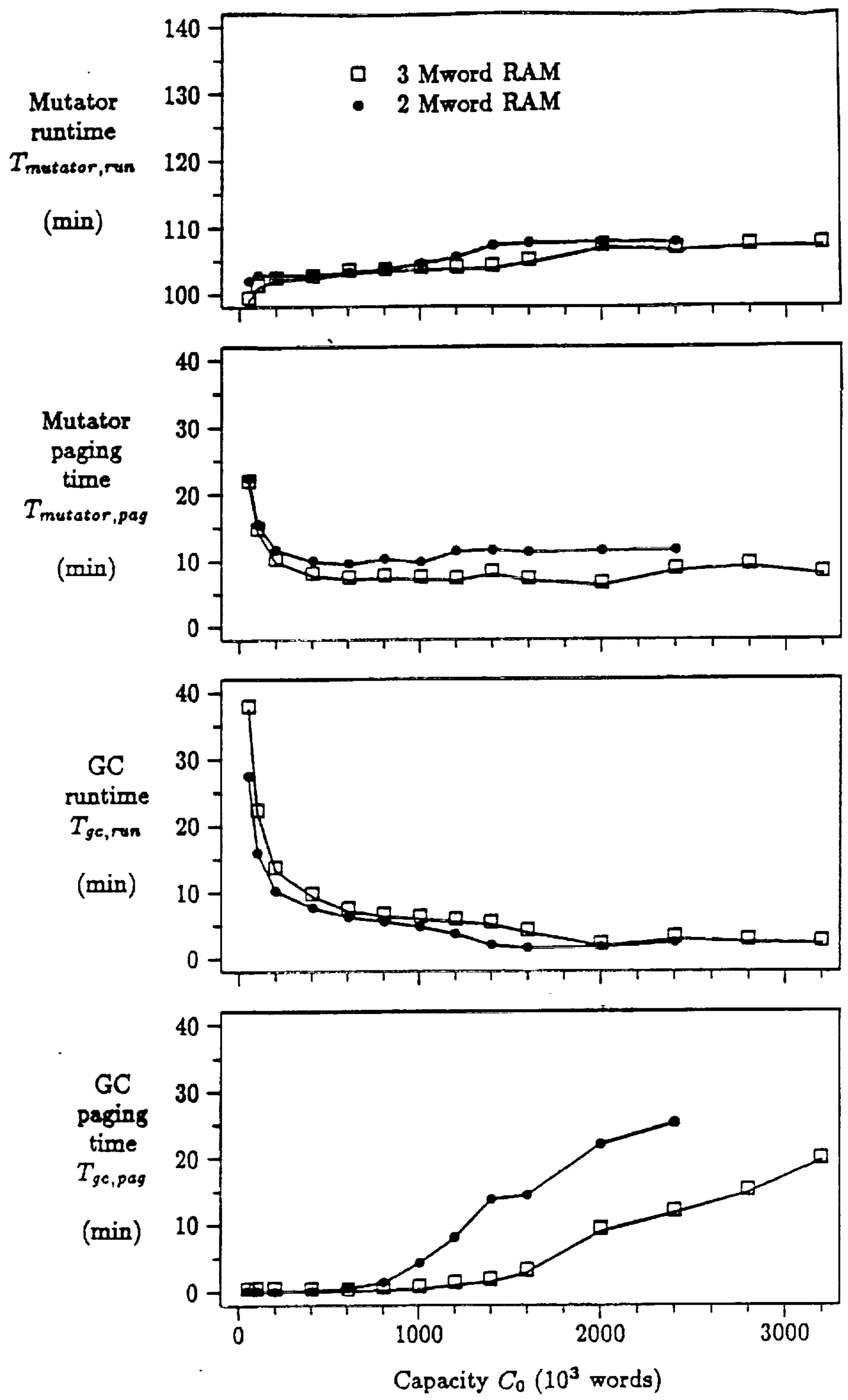

Figure 3.2 Breakdown of SRW total execution time.

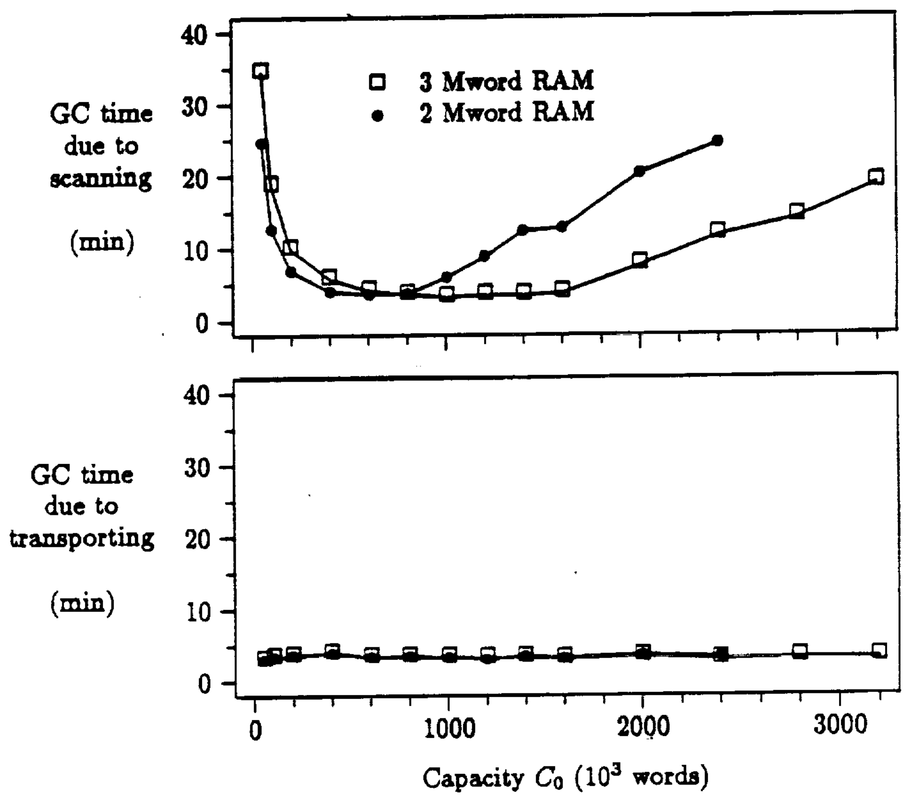

Figure 3.3 Breakdown of SRW total garbage collection time ($T_{gc,run} + T_{gc,pag}$) into scanning and transporting components.

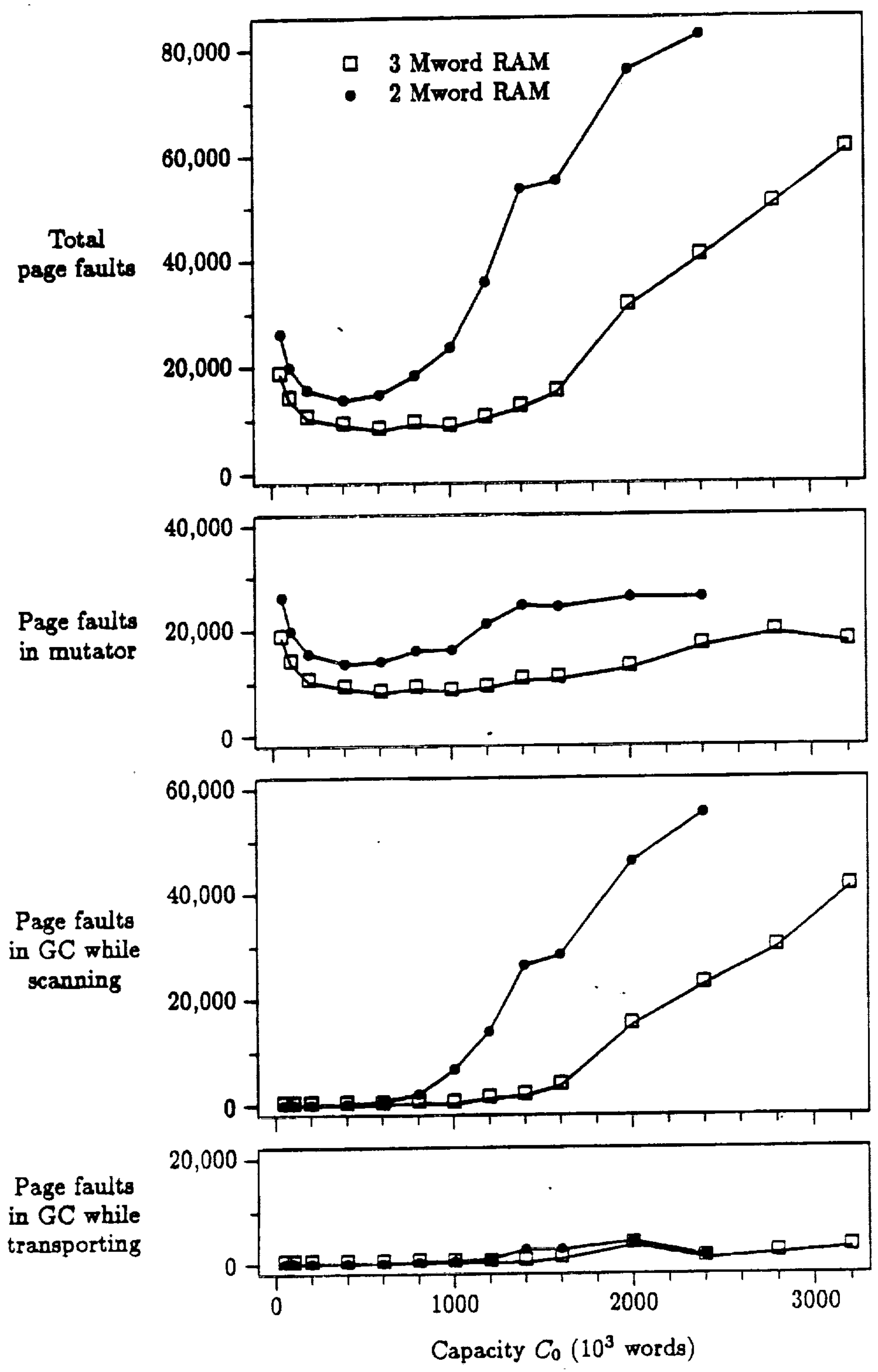

Figure 3.4 SRW page faults.

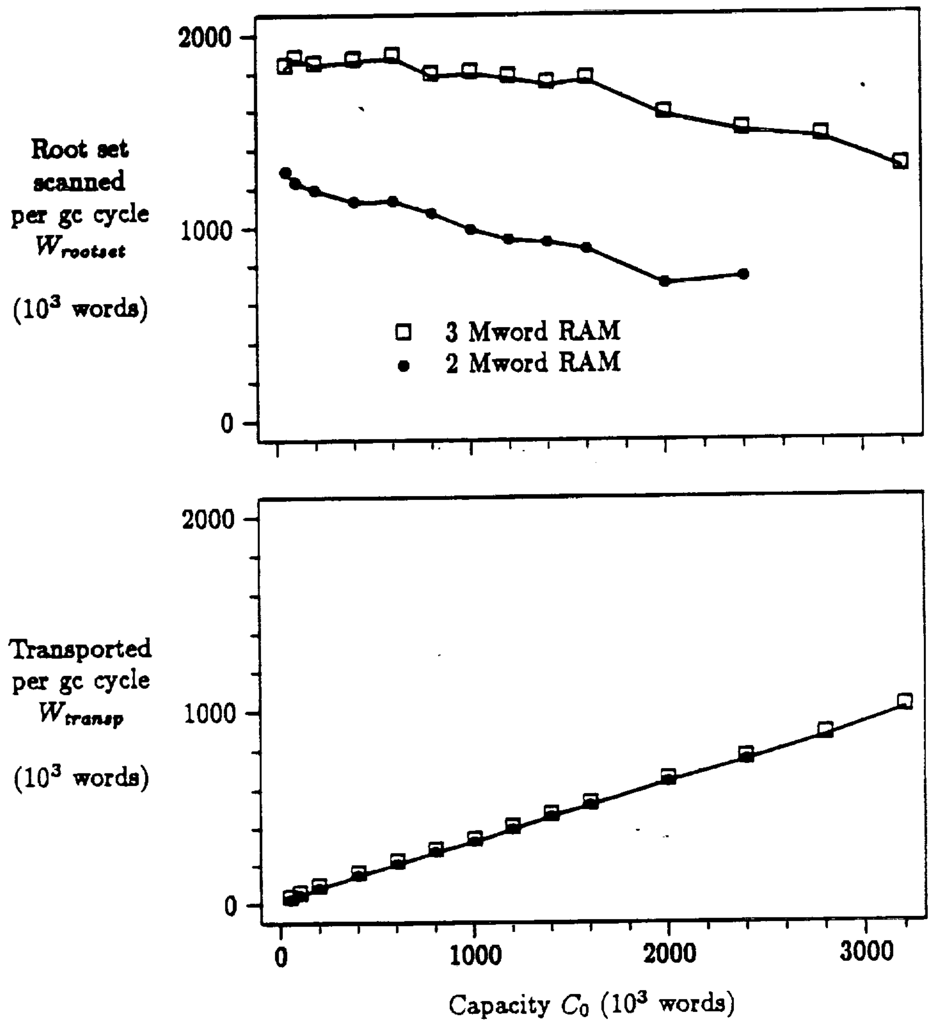

Figure 3.5 SRW root set words scanned and words transported per garbage collection cycle.

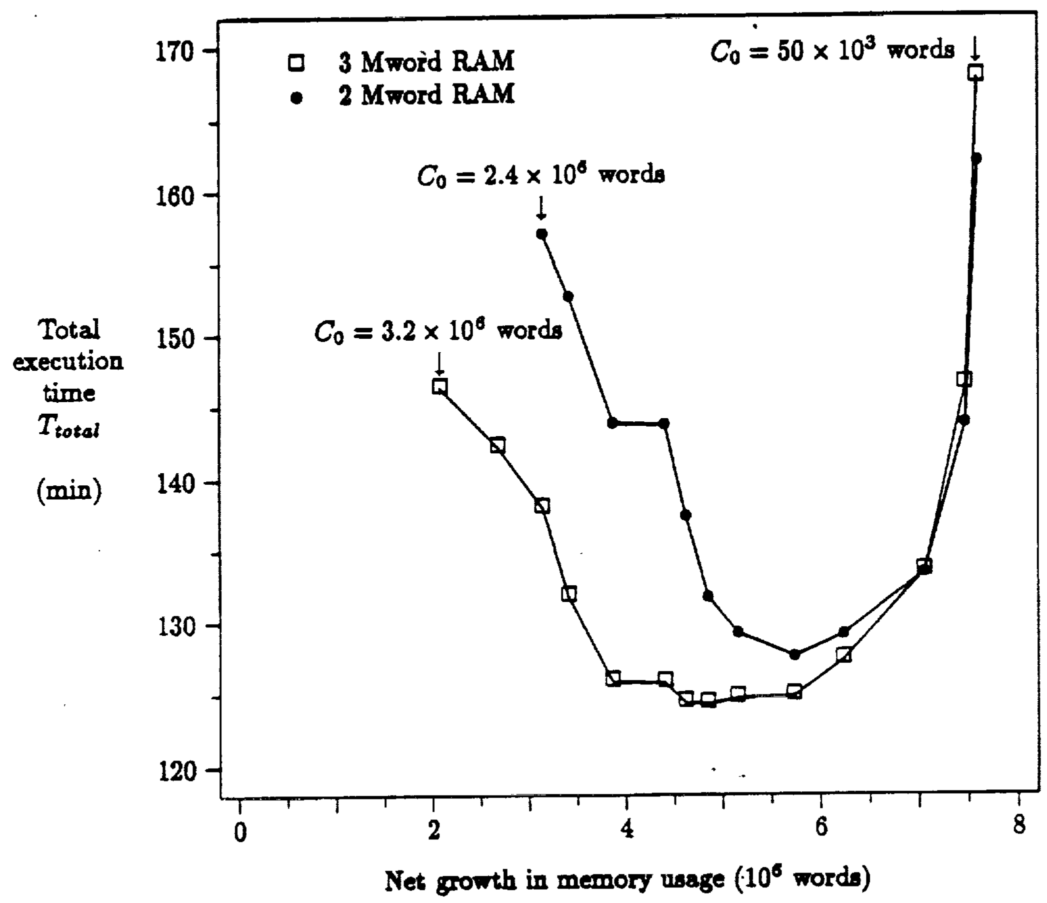

Figure 3.6 SRW total execution time vs. memory growth tradeoff.

The behavior of $T_{gc,run}$ is explained by the discussion in Section 3.2. To restate the explanation using the empirical results, consider the timing model (Equation (3.2)) and the observed relationships of $W_{rootset}$ and W_{transp} to C_0 as shown in Figure 3.5.

- Since $W_{transp} \propto C_0$, the total transporting time component of $T_{gc,run}$, $(M/C_0)k_{transp}W_{transp}$, is constant with respect to C_0.
- Since $W_{rootset}$ decreases (slightly) with C_0, the total scanning time component of $T_{gc,run}$, in particular, the total time scanning the root set, $(M/C_0)k_{scan}W_{rootset}$, decreases at least as fast as $1/C_0$.

Therefore, $T_{gc,run}$ decreases at least as fast as $1/C_0$. Note that Figure 3.3 verifies the above assertions regarding the scanning and transporting terms of $T_{gc,run}$. In reading this figure, note that, at small capacities, the total scanning and transporting times plotted therein consist primarily of runtime rather than paging time—since there is practically no paging during garbage collection (from Figure 3.2 or 3.4).

Essentially, at small capacities, $T_{gc,run}$ is high because collections are frequent and each collection has to scan the root set, whose size is relatively invariant with respect to C_0.

Possible causes for the rise in $T_{mutator,pag}$ at small capacities are

(1) displacement of the mutator's pages due to more frequent scanning of the root set, and

(2) reduced locality of the program's objects in tenured space as a result of an increased proportion of tenured garbage therein.

At large capacities, when collections are infrequent, the rise in T_{total} is caused by a rise in paging activity (Figures 3.2 and 3.4). The causes for the larger working set are as follows:

(1) Since C_0 is the amount allocated between garbage collections, as C_0 increases, the mutator, in the process of initializing created objects, references a proportionately larger amount of virtual memory.

(2) The number of words transported (and therefore having to be scanned) per cycle, W_{transp}, increases as αC_0 (Figure 3.5) where $\lim_{x\to\infty} S(x) = \alpha \approx 0.31$ is the fraction of allocation which is long-lived. The number of root set words scanned per cycle, $W_{rootset}$, decreases only about half as fast, thus providing only a partial cancelling effect. Further, since

each object transported requires touching two pages—one in oldspace and the other in copyspace—the rate of increase in page working set size with C_0 due to transported objects is effectively doubled.

(3) The locality of the objects in the first level could be reduced as a result of an increased proportion of garbage therein.

Figure 3.4 shows that the increased page fault rate is experienced primarily by the garbage collector during scanning, and to a lesser extent by the mutator.

Figure 3.6 shows the space-time tradeoff. The minimum increase in virtual memory required over the execution of the program is achieved at the highest capacity (lowest collection frequency) since that configuration minimizes tenured garbage, but the cost is paid in execution time as a result of paging. Minimum execution time incurs a moderate growth in memory usage.

Finally, it should be mentioned that the difference in the curves for root set scanned per cycle, $W_{rootset}$, for different main memory sizes (Figure 3.5) is a consequence of the way the ephemeral root set is maintained.[1]

3.3.2 Two ephemeral level configuration

The results for the two ephemeral level case are shown in Figure 3.7. Each curve in this graph represents execution time vs. second-level capacity C_1 for a fixed first-level capacity C_0. For comparison, the dotted lines indicate the execution time for a one-level configuration with the same capacity C_0.

The two-level curves are observed to be "flatter" than the one-level curves (Figure 3.1), which implies the relative insensitivity of T_{total} to second-level capacity as stated earlier.

The curves can be regarded as offsets from the corresponding one-level execution time, where the offset indicates the cost of having the second level. The rising characteristic of this offset as

[1] In the Symbolics system, separate tables are maintained to keep track of ephemeral root set pages which are in main memory (GCPT) and which are on disk (ESRT). When a virtual page which is tagged as part of the root set by the GCPT is ejected from main memory, checks are performed to determine whether the page is really a root set page before creating an entry for it in the ESRT. Hence, as root set pages in main memory are more frequently removed from main memory, such as would be more likely to happen at smaller main memory sizes, the total number of root set pages could decrease, as more GCPT-tagged pages are found not to be part of the root set.

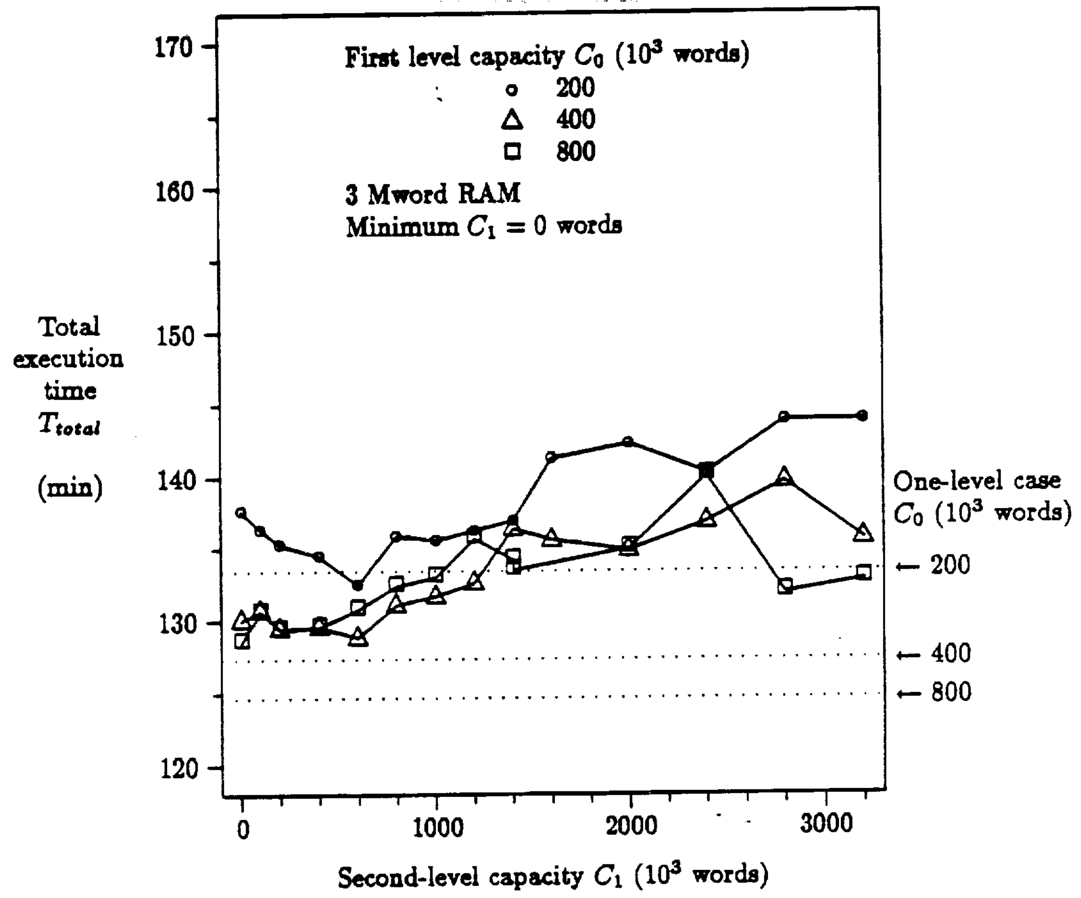

Figure 3.7 **SRW total execution time vs. second-level capacity for a generation configuration of** two ephemeral levels. Dashed lines indicate T_{total} for a one-level configuration with the specified capacity.

C_1 increases is similar to that in the one-level case and is also caused by increased paging, but the rise is less rapid because of the lower frequency (in real time) of flips of the second level. ("Allocation" into the second level occurs at a lower rate—and at discrete instants—than into the first level.)

However, unlike in the one-level case, the curves do *not* exhibit the rapid rise as C_1 decreases to zero. This is explained by noting that the frequency of first-level flips imposes an upper bound on the frequency of second-level flips. At $C_1 = 0$, the second level flips every time the first level flips, rather than infinitely frequently.

The jaggedness in the curves at large values of C_1 is due to boundary effects, i.e., over the program execution, the exact *number* of times that the second-level flips becomes significant.

3.4 Summary

An analysis of the effect of generation configuration in a generation-based garbage collector was conducted. In particular, the effect of the number of ephemeral levels and level capacity was studied. Analytic timing and survival models were used to represent the runtime component of garbage collection time and to derive structural results on the behavior of garbage collection runtime in the case of a one-level configuration. The survival model provides bounds on the age of objects surviving a garbage collection at a particular level.

Through controlled experiments with different generation configurations, it was found that execution time is most sensitive to the capacity of the first level. For SRW, the existence of a range of optimum values for first-level capacity demonstrates the potential for the tuning of garbage collection. The data suggests that, as main memory size increases, the optimum range broadens, i.e., the choice of capacity becomes less critical.

The factors contributing to suboptimal execution time performance were discussed. For the one-level case, at small capacities, more frequent scanning of the ephemeral root set and possible degradation in locality due to tenured garbage led to increased mutator page fault rate and garbage collector runtime. At large capacities, the increased amount of allocation per cycle, the increased amount of objects transported and scanned per cycle, and possibly the degradation in locality due to garbage in the ephemeral level led to an increased page fault rate.

Chapter 4

Dynamic Reordering

In this chapter, we discuss a memory management system which integrates garbage collection and dynamic reordering, and we present a method for measuring the intrinsic effectiveness of reordering. The method is used in two experiments, one involving system workload and the other a large application, and the results are discussed.

Dynamic reordering is an attempt to improve locality of reference by reorganizing objects within pages so as to group together objects which are being actively used. The motivation for reordering is the following set of empirical observations: (1) object sizes are usually much smaller than a page, and (2) usually only a small fraction of all accessible objects is accessed during a given interval of time. Together these characteristics create the potential for a kind of fragmentation in which the accessed objects are scattered about many pages. The evaluation technique we propose quantifies the degree of fragmentation.

The basic idea in reordering, as developed by Courts [7] and originally proposed by White [26], is to exploit existing garbage collector functionality to correct the fragmentation problem, assuming that it exists. Specifically, the existing capabilities which are relevant are

- detection of accesses to objects, and
- copying of accessed objects.

Recall that, to garbage collect some specified part of memory, that part is flipped into oldspace and the read barrier is raised for (or sensitized to) that portion of address space. When the barrier detects an attempt to reference an object in oldspace, the object is copied. All possible pointers to objects in oldspace are then found by scavenging other appropriate parts of memory. (Basically, scavenging involves reading memory locations sequentially with the intention of causing barrier faults.) At the end of scavenging, all reachable objects in oldspace have been evacuated, and oldspace can be made free for new allocation.

A similar procedure can be used for reordering purposes. We flip the part of memory to be reordered into oldspace and raise the read barrier as for garbage collection. However, scavenging is not performed; we simply allow the mutator to execute normally. In the process, the objects in oldspace which are accessed will be copied.

In other words, the same mechanisms used during garbage collection to determine reachability of objects (flipping and read barrier sensitization) and to copy them (transporting) can be applied during reordering to determine "activeness" of objects and to copy them. Garbage collection is interested in copying all accessible objects, in the order that scavenging discovers them. Reordering is interested in copying only the subset of all accessible objects which are in the mutator's working set, in the order that they are first accessed.

The existence of common mechanisms used by garbage collection and reordering suggests the possibility and desirability of an integrated garbage-collecting/reordering memory management system. We have developed such a system for the Symbolics computer, which is described in Section 4.3. A similar system is the temporal garbage collector (TGC) developed by Courts [7] for the Texas Instruments Explorer Lisp computer.

4.1 Previous Work

Strategies for static reorganization to improve locality have been the subject of several previous investigations. Ferrari [9, 10, 11] and Hatfield [13] developed program restructuring techniques in static language systems (FORTRAN) based on reference traces. Stamos [20] studied graph-based algorithms for reordering system objects in a Smalltalk system. Andre [1] developed many techniques for ordering system objects in the Symbolics Lisp system, based on metering dynamic references and on detailed knowledge of the referencing characteristics of certain critical system operations.

The only work on dynamic reorganization which we are aware of prior to Courts' garbage collector [7] and White's proposal [26] is that of Baer [3] who simulated a memory system in which pages were dynamically grouped within larger units of disk transfer.

4.2 Issues in an Integrated Memory Management System

Before describing our memory management system in detail, we first discuss the issues facing such a system and outline the approaches taken.

4.2.1 Simultaneous garbage collection and reordering

In an integrated system, it should be possible to garbage collect some parts of memory while simultaneously reordering other parts. Therefore, it is necessary to distinguish between oldspace which is being collected and oldspace which is being reordered. Our system, as well as the TGC, defines two types of oldspace corresponding to the two possible usages, called *true oldspace* and *reordering oldspace*.[1] The memory management system needs to recognize these two types of oldspace and treat them accordingly.

While garbage collection and reordering share the same mechanisms, the timing of their associated events is different. Garbage collection involves a definite control sequence—flip, scavenge, reclaim oldspace, wait, flip, ... Reordering begins with a flip, but the ensuing mutator-induced transporting action should be able to continue indefinitely until the next flip which either begins another round of reordering or begins a normal garbage collection sequence. The memory management system needs to be able to handle these disparate scenarios.

4.2.2 Preserving object order under garbage collection

Although reordering can continue indefinitely, eventually, we will want to garbage collect memory which is being reordered, to reclaim space taken up by

- any objects in reordering oldspace or copyspace which have become unreachable, i.e., garbage, and
- forwarding pointers in reordering oldspace, which may or may not be garbage depending on whether there exist pointers to them which have just not yet been "snapped."

We will also want to garbage collect before saving an image of virtual memory on disk for future use, or for release to other users. The problem with performing a normal copying garbage collection is that it will destroy the order of objects in memory, negating any benefit

[1]The corresponding types are *from-space* and *train-space* in the TGC.

from reordering, since the reachable objects are copied in an order determined only by graph connectivity (e.g., depth-first order), without regard to their current placement.

The approach taken by the TGC is to preserve order *approximately* by defining a new (horizontal) dimension in the virtual memory organization. The vertical dimension represents generations (levels) as before. Within each generation, objects are divided into activity categories. Essentially, during garbage collection, surviving objects in a given activity category are copied together to the next (less active) category. Hence, order is maintained approximately in the sense that the survivors remain together, although their relative ordering could change. Since the survivors are added to any existing objects in destination category, a subsequent garbage collection will flip the union of the added and added-to objects. Because of this coalescing, and the few number (four) of activity categories—such that inactive objects eventually and quickly migrate to the least active category under repeated garbage collection—any object "togetherness" established by reordering can deteriorate over time.

Of course, object togetherness can be re-established by reordering, but it would be desirable to have a scheme in which togetherness does not deteriorate under repeated garbage collection. We propose two non-mutually-exclusive approaches to the problem of preserving order.

4.2.2.1 Preserving togetherness through lineages

The first approach is similar in spirit to the TGC. The objective is to keep together the objects which have been copied as a result of reordering. To this end, a *keep* bit in the region table is set for each copyspace region R which is to receive objects to be transported out of reordering oldspace. This bit indicates that, when region R is later flipped into oldspace for garbage collection, the surviving objects must be copied to their own unique copyspace region R' (rather than into a common region with other survivors), and the set state of the keep bit must be passed on to R'. (The oldspace region R will be reclaimed and its keep bit reset.)

Thus, by setting the keep bit for a copyspace region during reordering, a *lineage* is established for the objects in that region. An object can leave its lineage only by becoming garbage and passing from existence, or if it is copied during reordering, in which case, it joins a new lineage. Objects in a lineage will remain together when garbage collected (although their relative order could change). A lineage implicitly defines an activity category unique to a set of objects placed together by reordering. Unlike activity categories in the TGC, there can be any number of lineages existing at a time, and there is no coalescing among different lineages except when explicitly requested (e.g., during compaction, explained below, or when the region table is almost full due to the existence of many lineages).

4.2.2.2 Preserving exact order through compaction

The second approach, which has no counterpart in the TGC, is an operation called *compaction*. Essentially, compaction is a garbage collection of reordering oldspace which preserves object order *exactly*, and reclaims space taken up by forwarding pointers but not by other garbage. To do this, reordering oldspace is flipped into true oldspace as if to start a normal garbage collection. However, before scavenging, *all nonforwarded objects* in (the now true) oldspace are transported in their current memory order. We call this operation *bulk transportation*. Scavenging is then performed as usual, but its purpose now is simply to redirect outstanding pointers to oldspace; no objects can possibly be transported out of oldspace.

Compaction can optionally honor or ignore lineage. When bulk transporting nonforwarded objects out of an oldspace region whose keep bit is set, the copies of the objects can be placed in their own descendant copyspace region, or in a common copyspace region (effectively coalescing lineages).

Since compaction reclaims space taken up by forwarding pointers but not other garbage, it is ideal when reordering oldspace is known to have little or no garbage, e.g., an area of memory containing permanent objects which has been flipped for reordering, subjected to some representative workload, and now is ready to be "set."

Is there any advantage to preserving exact order rather than just togetherness? Although we have not had any actual experience with this idea, one situation in which it would be useful to preserve order exactly through compaction, is when using dynamic reordering to *induce sequentiality* in the mutator's page reference string. One scenario is if it is known that the mutator references objects in a cyclic manner. Another scenario is if reordering is intended to optimize memory in a "production environment," in which it is known that the workload used to effect the reordering is very similar if not identical to the production workload. In both cases, reordering objects may result in a pattern of page references which may be amenable to prefetching.

We have implemented compaction in our system. However, implementation of togetherness-preserving garbage collection using the lineage concept as outlined above requires some modifications to the microcode (which determines the region to which an object will be transported) to which we did not have access.

4.2.3 Multiple reorderings

Suppose we begin a reordering of some part of memory. After some time, due to the buildup of garbage among the objects in copyspace, or due to shifts in the mutator working set, the set of objects in use in copyspace may become fragmented. It would be desirable to begin a new round of reordering without terminating the current one by a garbage collection (or compaction). For example, a garbage collection could take a very long time, and we may prefer to do it only overnight or on weekends.

The memory management system should make it possible to terminate a reordering and begin a new one, without requiring an intervening garbage collection, by simply flipping copyspace into reordering oldspace and allocating a fresh copyspace to receive the currently used objects. After this is done a number of times, reordering oldspace will consist of regions belonging to different lineages, which can be visualized as being ranked in order of time of formation, i.e., the time at which the constituent objects were placed together. At any one time, there can be any number of existing lineages. The granularity with which objects are assigned to lineages is controlled simply by the times at which new reorderings are begun.

4.3 System Description

We discuss the design of a new memory management system for the Symbolics Lisp computer, which integrates garbage collection with dynamic reordering, and compaction, an operation supplementary to reordering (Section 4.2.2.2).

It is useful to view the garbage collector not as a monolithic system procedure, but as a collection of components, which, when invoked according to different rules, can perform a variety of object management tasks, namely, garbage collection, reordering, and compaction. These components are now described. The following presentation is oriented towards emphasizing the modifications or additions made to functions in the existing memory management system, as developed by Moon [15].

4.3.1 Flipper

The changes to the functions for flipping enable them to handle requests to begin a normal garbage collection, a compaction of reordering oldspace, or a reordering. Also, if a compaction is requested, the flipping operation performs a bulk transportation of objects in the appropriate regions.

Flipping of ephemeral space (level > 0) is performed by the function `GC-FLIP-EPHEMERAL-SPACES-NOW` which can be given a specification of which ephemeral levels to flip for ephemeral garbage collection, which levels to flip for compaction, and which levels to flip for reordering.

Flipping of dynamic space (level 0) is performed by `GC-FLIP-NOW` which can be given a specification of which areas to flip for dynamic garbage collection, which areas to flip for compaction of reordering oldspace, and which areas to flip for reordering.

The task of either flipper is to

(1) Determine whether it is permissible to flip. The previous garbage collection or compaction should have been completed.

(2) Relabel the regions constituting newspace, copyspace, or reordering oldspace in the requested levels (or areas) as oldspace, setting their reorder bit in the region table accordingly. The reorder bit distinguishes a true oldspace region from a reordering oldspace region.

(3) Raise the read barrier in the processor for all oldspace regions, whether true or reordering oldspace.

(4) If one or more levels (or areas) were flipped for garbage collection, or compaction, prepare to scavenge the regions which should be scavenged. This involves primarily setting the regions' respective scan pointers to either zero or the current free pointer.

(5) If one or more levels (or areas) were flipped for compaction, transport all nonforwarded objects in the oldspace regions constituting those levels (areas).

Table 4.1 Read barrier states.

Scavenging	Scavenger process	Read barrier should be raised for	
		True oldspace	Reordering oldspace
In progress	Running	Yes	No
In progress	Not running	Yes	Yes
Not in progress	—	Nonexistent	Yes

4.3.2 Scavenger

The changes to the functions for scavenging are

- To lower temporarily the read barrier for reordering oldspace while the scavenger is running.
- To make it possible to scavenge a page or region in reordering oldspace.

Scavenging is performed by the function %GC-SCAVENGE and involves scanning all appropriate regions as prepared by the flipper. The objective is to find all pointers to true oldspace by reading memory locations sequentially so as to induce read barrier faults. When a barrier fault occurs, the referenced object is copied if necessary (by the transporter), and the faulting pointer is redirected to the copy. The scavenger's linear scan through memory is occasionally suspended in order to scavenge the "last page" in a copyspace region which has just grown due to transportation. This technique is to achieve an approximately depth-first copying order [15].

The first change in our system is to "hide" the existence of reordering oldspace from the read barrier during scavenging. Note that scavenging can be performed incrementally, i.e., interleaved with other processes (the mutator). While the scavenger is running, the read barrier should be raised for true oldspace but not reordering oldspace, i.e., the barrier should be sensitive to pointers to true oldspace only (Table 4.1). Any pointers to reordering oldspace encountered during the scan should be ignored—reordering oldspace is not being garbage collected. When other processes are running, the read barrier for reordering oldspace should be raised so that the usual faulting of objects out of reordering oldspace can occur.

In the new system, every time control passes to the scavenger process, we temporarily lower the read barrier for reordering oldspace. (The barrier remains raised for true oldspace.) The overhead for this manipulation is very low since the scavenger process runs for a relatively long time before allowing itself to be pre-empted.

The second change makes it possible to scavenge reordering oldspace by "hiding" the presence of forwarding pointers. This change is essentially a solution to a problem caused by the existing behavior of one machine instruction. Before explaining this change, we first note that pages or regions in reordering oldspace can be part of the memory to be scavenged.

- If an ephemeral garbage collection is in progress, the memory to be scanned consists of the ephemeral root set—the pages remembered by the GCPT (in-main-memory) and ESRT (on-disk) tables—followed by the portions of copyspace which appear after the flip. It is possible for a page in reordering oldspace to be part of the root set, e.g., reordering oldspace in an ephemeral level not being garbage collected or in level 0.

- If a dynamic garbage collection is in progress, the memory to be scanned consists of all regions which are not in true oldspace. If a dynamic garbage collection of some areas is in progress, it is possible for reordering oldspace regions in other areas to be present. These regions must be scavenged; they could have pointers to true oldspace.

In principle, scanning a page in reordering oldspace is no different from scanning a page in any other space, except that we can encounter forwarding pointers. These pointers are irrelevant and should have no effect since they cannot point to true oldspace. Scanning a range of addresses for pointers to oldspace is efficiently implemented via the function %BLOCK-GC-TRANSPORT which compiles into a single machine instruction. Unfortunately, this instruction signals an error when it detects a forwarding pointer. To avoid the occurrence of this error, during scavenging, we search for forwarding pointers and call %BLOCK-GC-TRANSPORT only on address ranges not containing forwarding pointers. This effectively "hides" forwarding pointers from %BLOCK-GC-TRANSPORT.

4.3.3 Single-object transporter

Transporting a single object from oldspace to copyspace and installing forwarding pointers is performed by the function TRANSPORT-TRAP. The change introduced here concerns the level to which an object is copied. The destination level is determined differently depending on whether the object is in true or reordering oldspace, i.e., whether the object was discovered as a result of garbage collection or as a result of reordering.

- If the object is in true oldspace, the destination level is the next older level (or zero if the object is already at level 0).[2]
- If the object is in reordering oldspace, the destination level is the same level.

4.3.4 Bulk transporter

As explained earlier, compaction of reordering oldspace is done by performing a normal garbage collection of the reordering oldspace, except that immediately after flipping the constituent regions into true oldspace, we transport all nonforwarded objects therein. The destination level for copying objects is the same level. Bulk transportation of objects is performed by %GC-TRANSPORT-EPHEMERAL-SPACE or %GC-TRANSPORT-AREA, depending on whether the reordering oldspace is in an ephemeral level or in (level 0 of) an area being compacted. These are new functions, with no counterparts in the existing system.

4.3.5 Reclaiming oldspace

Oldspace reclamation is performed by the function GC-RECLAIM-OLDSPACE. It is run after scavenging is completed, at which time no pointers to true oldspace exist. Its responsibility is to reclaim true oldspace regions by relabeling them as free space, thereby making them available for future allocation. The change we introduce is simple: Only true oldspace regions are relabelled as free space. Reordering oldspace regions are not reclaimed.

4.3.6 Ephemeral root set table maintenance

The system keeps track of the ephemeral root set, i.e., the pages into which the processor has written pointers to ephemeral space, by means of two tables, one for in-main-memory pages (GCPT), the other for on-disk pages (ESRT). The GCPT consists of a single bit for each page frame in main memory; the processor sets the bit associated with a main memory page when a write of an ephemeral pointer occurs to the that page. The ESRT is a sparse table which is maintained in cooperation with the virtual memory management system as follows.

[2]Promotion to the next level is the normal case. The mechanism for implementing the promotion policy for true oldspace objects is a look-up table, specifying, for each ephemeral level, the destination level. Hence, other policies are easily effected. For example, an ephemeral level L could be made "sticky" by setting the destination level for L to be L.

When the virtual memory management software ejects a page from main memory, it calls the function GC-PAGE-OUT for the purpose of maintaining the ESRT. In the existing system, this function checks if the page is in oldspace.

- If the page is in oldspace, any existing ESRT entry for the page is unconditionally deleted, since oldspace is not part of the root set and should not be scavenged.
- If the page is not in oldspace, an entry for the page in the ESRT is created if necessary, or the existing entry is updated or deleted as appropriate. The appropriate action to take is determined by scanning the page to see whether it contains any pointers to ephemeral space. If there are any, the ESRT entry consists of a bit mask indicating the ephemeral levels referenced by the pointers. If there are no such pointers, any existing ESRT entry is deleted.

The change required in the new system is to make the check described above more specific. We check if the page is in *true* oldspace. A page in reordering oldspace can be part of the root set for ephemeral garbage collection. Its ESRT entry must not be unconditionally deleted, but must be maintained like that for a non-oldspace page.

4.4 Evaluation

One way to evaluate the effectiveness of reordering is to compare the time (or page faults) it takes to run a workload with and without reordering. Courts [7] used a "system benchmark," consisting of a script of typical user interactions (e.g., editing, compiling), to show reduction in execution time by a factor of about four under constant main memory size due to reordering. By experimenting with different main memory sizes, he also showed a reduction by a factor of about two in main memory size required for constant execution time.

Clearly, the amount by which reordering can reduce execution time (and paging time in particular) is dependent on main memory size. For example, if main memory is much larger than the threshold at which thrashing begins to occur for a particular workload, reordering will be of little benefit in reducing paging time.

It is not our main intention in this thesis to provide similar measurements of execution time or page faults, but rather to propose a method to evaluate a reordering in a way that is independent of main memory size. The method measures the reduction in page working set size. This measurement is more reflective than execution time of the intrinsic potential benefit from reordering. It can be used, as we show later, for determining which areas in memory are good candidates for reordering, and which are not. Since the evaluation is not affected by main memory size, it can also provide useful information in a situation in which one is developing an application or system to be run on other machines. The development machine may have a large main memory such that reordering has little effect on execution time, but the delivery machine may have less memory.

The measurement technique, called ***scanning for transport statistics***, is now described. The evaluation is analogous to that for file compression, in which the performance of the compression algorithm is measured by the absolute and relative reduction in file size.

4.4.1 Scanning reordering oldspace for transport statistics

Suppose a reordering of an area begins at time t_{flip}. For simplicity, assume that this is the first time that the area is being reordered. (This assumption is unnecessary and will be removed shortly.) At some arbitrary time $t_{eval} > t_{flip}$, we would like to evaluate the reordering that has occurred. Let Ψ be the set of objects in reordering oldspace which have been accessed during the interval (t_{flip}, t_{eval}). These objects will have been transported to copyspace, and forwading pointers will have been installed in their former locations in reordering oldspace (Figure 4.1).

Define the following sets:

ω_{old}	Set of pages in reordering oldspace containing Ψ
ω_{copy}	Set of pages in copyspace containing Ψ

Thus, over the interval (t_{flip}, t_{eval}), Ψ is the ***partial object working set.*** "Partial" refers to the fact that Ψ is a subset of the full object working set, i.e., all objects accessed during (t_{flip}, t_{eval}). It is the subset that we happen to know about because of the object faulting action. Similarly, ω_{old} and ω_{copy} are ***partial page working sets.*** They are disjoint, nonexhaustive subsets of the full page working set. Henceforth, by "working set," we will mean partial working set.

By scanning reordering oldspace at t_{eval}, looking for objects which have been forwarded to copyspace, we can obtain in one pass the following *transport statistics*:

$PAG_{old} = \|\omega_{old}\|$	Number of pages in reordering oldspace containing Ψ
WOR	Total size of Ψ in words

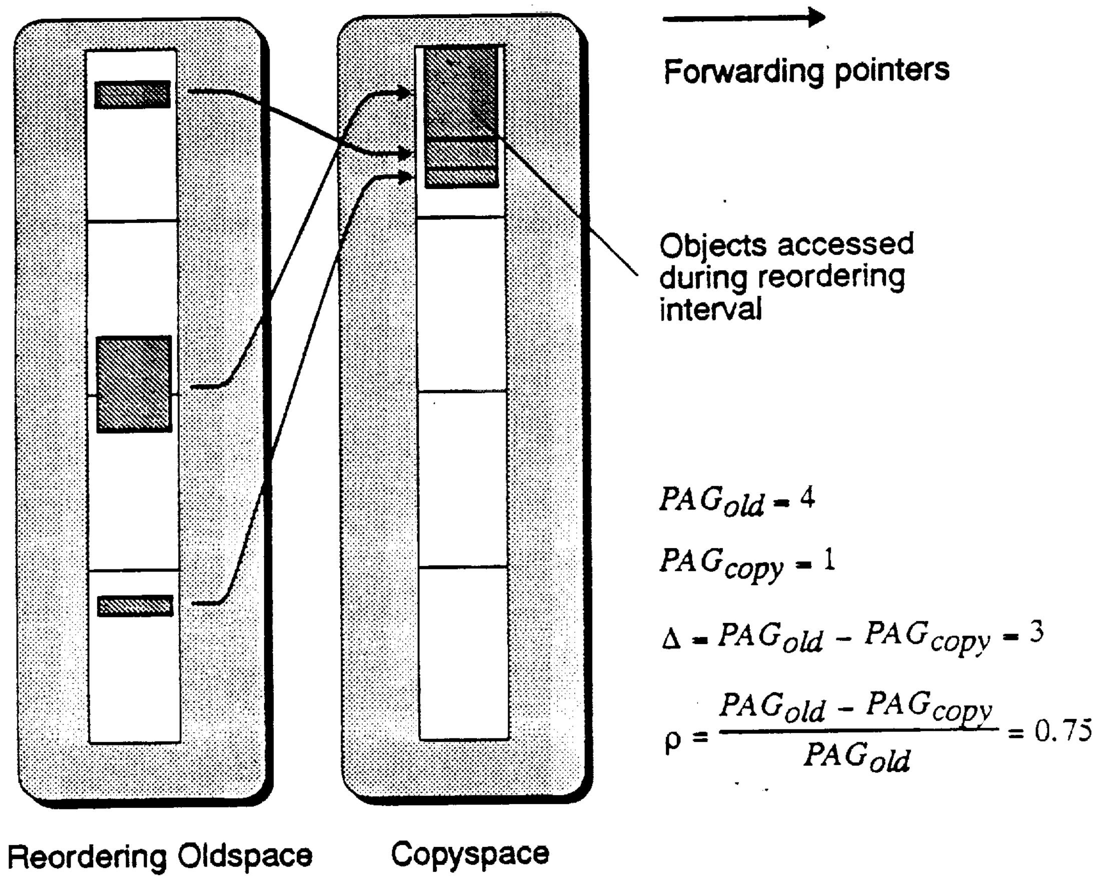
Forwarding pointers
Objects accessed during reordering interval
$PAG_{old} = 4$
$PAG_{copy} = 1$
$\Delta = PAG_{old} - PAG_{copy} = 3$
$\rho = \frac{PAG_{old} - PAG_{copy}}{PAG_{old}} = 0.75$
Reordering Oldspace
Copyspace

Figure 4.1 Example evaluation of reordering performance.

Other data can also be collected during the pass, such as would enable us to characterize the distribution of object sizes. However, only the above is essential. We then compute

PAG_{copy}	$= \lvert\omega_{copy}\rvert = \left\lceil \frac{WOR}{PAGESIZE} \right\rceil$	Number of pages in copyspace containing Ψ
Δ	$= PAG_{old} - PAG_{copy}$	Reduction in page working set size
ρ	$= \frac{\Delta}{PAG_{old}}$	Page working set compression ratio

where *PAGESIZE* is the size of a page in words and $|x|$ denotes the number of elements in a set x. For the Symbolics Lisp computer, $PAGESIZE = 256$.

4.4.2 The density and working set reduction measures

The compression ratio ρ is the relative reduction in working set size due to reordering. It also measures the fragmentation of the objects in Ψ. To see this, note that, ignoring the ceiling operator,

$$1 - \rho = \frac{PAG_{copy}}{PAG_{old}} = \frac{WOR}{PAG_{old} \cdot PAGESIZE} \tag{4.1}$$

which is the density with which the objects are packed into pages. A value of $\rho \ll 1$ implies that the objects are densely packed, while $\rho \approx 1$ implies that the they are scattered over many pages, intermixed with much garbage or with accessible but unused objects.

To be precise, the "pages saved" measure Δ should be defined as the reduction in working set size over the interval (t_{flip}, t_{eval}) which would have been realized had the objects in Ψ been compactly laid out in virtual memory at time t_{flip}.

This hypothetical savings Δ will indeed be realized if reordering is intended to optimize memory in a "production environment." This usage is analogous to the use of program restructuring techniques in static systems (e.g., Hatfield [13] and Ferrari [11]). In this usage, a terminating compaction operation is performed, reordering oldspace is reclaimed, and the new layout is subjected to similar workload. Similarity between the workload used to effect the reordering and the "production" workload requires only similarity in the *set* of objects referenced, not in the sequence of references.

When reordering is used in a more dynamic sense, i.e., to continually tailor object layout to usage characteristics, Δ provides an approximate measure of the savings. In this usage, we are interested in performance as reordering occurs, rather than during a future "production run." By the strict definition of a working set [8], namely, the set of unique pages accessed during a specified interval, the working set size over (t_{flip}, t_{eval}) is actually *larger* than it would have been had reordering not been initiated at t_{flip}. Without reordering, only PAG_{old} pages would have been accessed; with reordering, an additional PAG_{copy} pages are accessed since the active objects have to be copied. It would appear that reordering has actually degraded locality!

To understand this paradox, note that the period during which a round of reordering takes place will usually be very long. During this interval, an attempt to reference an object O in reordering oldspace will create a copy O' of the object in copyspace. After transportation, the oldspace version O, and therefore the page on which it resides, will again be touched if there exist other pointers to it and if those pointers are exercised (and, as a result, permanently redirected). Clark [6] has shown that, most of the time, there is only one pointer to an object. Thus, due to this connectivity property of the object graph, the probability that O will be rereferenced is low. Due to the long duration of a reordering interval, the probability that O' will be rereferenced is high. It is therefore reasonable to expect that the *average* working set size over an arbitrary *subinterval* of (t_{flip}, t_{eval}) will be smaller as a result of reordering, with the decrease from the unreordered case being approximately given by, or at least proportional to, Δ. We adopt this interpretation of Δ and present some experimental results consistent with it (Section 4.5.2), but suggest for future work that actual traces of object references be analyzed to verify it. An object-level simulation system such as developed by Zorn [31] would be useful in this regard.

4.4.3 Computation vs. measurement of number of copyspace pages

As an alternative to computing $PAG_{copy} = \lceil WOR/PAGESIZE \rceil$ as shown above, it may also be possible to measure it easily. During the interval (t_{flip}, t_{eval}), if there is no reason for growth in copyspace other than transportation of objects from reordering oldspace, we can simply note the increase in the number of pages of copyspace. The measured value should equal the computed value exactly.

However, in general, there could be other reasons for expansion of a copyspace region, most plausibly, objects surviving a garbage collection of a higher (younger) ephemeral level and being copied to the level being reordered.[3] In this case, the computed PAG_{copy} should be interpreted

[3]Another possibility is a Lisp system which does not have separate newspace and copyspace, but simply allocates new objects in the same space as objects copied from oldspace.

as the theoretical minimum number of copyspace pages needed to contain Ψ. The actual number of pages could be greater due to objects copied during garbage collection being interspersed with the objects in Ψ.

In using the computed (theoretical minimum) PAG_{copy} to derive the Δ and ρ metrics, we are evaluating the "intrinsic" effectiveness of reordering, independent of whether there are any sources of input to copyspace other than reordering oldspace.

If it is desired to evaluate the *actual* rather than intrinsic benefit from reordering when copyspace is being simultaneously "polluted" from another source, an inexpensive way to measure the actual number of copyspace pages containing Ψ is as follows. Modify the single-object transporter to maintain a count of copyspace pages receiving objects from reordering oldspace. This will require storage, on a per-region basis, for a counter and for the virtual page number of the "last page marked."

4.4.4 Generalization to multiple reorderings

As defined above, the method of scanning for transport statistics does not require the assumption that we are reordering an area (or ephemeral level) for the first time. In general, a reordering of an area can be initiated more than once before garbage collecting or compacting it.

Suppose reordering has been initiated at times $t_{flip1}, t_{flip2}, \ldots, t_{flipN}$ without any intervening garbage collection or compaction, where t_{flipN} is the time at which the most recent reordering began. When scanning reordering oldspace for transport statistics at $t_{eval} > t_{flipN}$, we can encounter objects which are forwarded several times within reordering oldspace before possibly being forwarded to copyspace.

During the scan, we make an object contribute towards PAG_{old} and WOR only if the object is forwarded *directly* to copyspace. The set of such objects is precisely the set Ψ of objects which have been accessed during the most recently initiated reordering interval, (t_{flipN}, t_{eval}). In other words, at evaluation time, we are interested only in the last link in a chain of forwarded objects terminating in copyspace. The statistics so obtained will be an evaluation of the most recently initiated reordering interval.

4.4.5 Parsing oldspace

Scanning oldspace (reordering as well as true oldspace—see Section 4.4.6) to collect transport statistics involves "parsing" memory to determine object boundaries, i.e., starting addresses and sizes. This parsing is complicated by the presence of forwarding pointers. In particular, the problem of *noninvariance of object representation with respect to forwarding* arises. We describe this problem and the solution adopted.

Objects in memory are self-identifying by virtue of the tagged architecture and the Lisp system's conventions for representing objects [16]. Hence, it is easy to determine the virtual memory extent of a nonforwarded object given any address in its representation. This simple determination is done by the single-object transporter when called to evacuate a previously untransported object in oldspace, and by the bulk transporter (Section 4.3.4) while evacuating all untransported objects in oldspace.

However, when scanning oldspace for transport statistics, we are interested only in the objects forwarded to copyspace. In particular, we would like to know about a forwarded object as it existed just *before* forwarding. A problem arises because, when an object is forwarded, its existing representation is overwritten with forwarding pointers. At scanning time, only the new representation can be examined for size and other information, and the new representation can be different from the pretransport one.

The nature of the possible difference depends on whether the object is a list object or a structure object. Lists are built from list cells, also called *cons* cells or simply *conses*. Structure objects refer to all other types of data (e.g., arrays, symbols, compiled functions) which are represented by a header word followed by one or more words of information. Structures and lists are stored in separate regions. Given a particular region of oldspace to be scanned, the parsing algorithm appropriate for the region's type is applied.

4.4.5.1 Noninvariance of structure size across forwarding

Immediately after a structure object is transported, the sizes of the oldspace and copyspace representations are equal, of course. However, in order to support such Common Lisp language features as adjustable arrays [21], an object may grow in place or, if this is not possible, the system may forward an object to a new and larger representation. Because of the possibility of in-place expansion of an object, and of forwarding when an object cannot expand in place, the size of an object before forwarding is not necessarily the size of the object to which it is forwarded. That is, the size of a structure object is usually, but not always, invariant with respect to forwarding.

To correctly determine object boundaries in structure oldspace, we use an algorithm that "looks ahead" by one object and resorts to a less efficient word-by-word scan in the rare event that a size discrepancy is detected.

This algorithm maintains a scan pointer P which always points to the first word of an object whose size is as yet unknown. Call this object O. If O is not forwarded, its size S is easily determined from system conventions, and the scan pointer is incremented by the size. If O has been forwarded, the chain of forwarding pointers is followed to the real (unforwarded) object O', whose size S' is then determined. In the vast majority of cases, S' is the correct amount by which to increment the scan pointer. However, if a size change has occurred over any link in the forwarding chain, it would be erroneous to use S' as the size of O. We detect the occurrence of a size change by checking whether $P + S'$ contains the start of an object (after following any forwarding pointers). If so, S' is accepted as the size of O. Otherwise, a word-by-word search beginning with location $P + 1$ is made for the first location P_{next} which contains (or is forwarded to) the start of an object. The size of O is then $P_{next} - P$.

4.4.5.2 Non-invariance of list representation across forwarding

For lists, the difference between oldspace and copyspace representations arises from the use of cdr-coding to make more efficient use of memory. In a Lisp system which does not use cdr-coding, the size of a *cons* is always two words—one each for the *car* and *cdr*—and there is no possibility of this size ever changing. In a cdr-coding system, some *conses* may be represented normally (two words) while others may be cdr-coded (one word for the *car*, with the *cdr* being implicit).

Transformations on the representation of lists can occur at any time due to the use of the RPLACD function and during transportation. When RPLACD is performed on a cdr-coded *cons*, the *cons* must be forwarded to a normal *cons*.

The transporter (both single-object and bulk) can also change list representation. Transporting a normal *cons* out of oldspace is straightforward; the two words are simply copied and there is no change. However, when transporting a cdr-coded *cons* C_1, there are two approaches. One is to create a normal *cons* in copyspace. Doing so eventually converts all accessible lists to unencoded form. The other approach, which is taken in the Symbolics system, is to preserve the cdr-coding by transporting the cdr-coded *segment* surrounding C_1, where the segment is terminated either by the end of the list or by an RPLACD-forwarding pointer. If the segment is terminated by an RPLACD-forwarding pointer (to a normal *cons* C_2), then the copy of the segment will be different in one of two ways:

(1) The forwarding pointer to C_2 will be replaced by C_2 itself, i.e., C_2 is also transported and *reattached* to the segment. In this case, the copy of the segment will be one word larger.

(2) The forwarding pointer to C_2 and the cdr-coded *cons* preceding it are converted into a normal *cons*.

Similar to the structure parsing algorithm, the algorithm for parsing list oldspace uses knowledge of the possible changes in representation that can occur as described above and performs any necessary "look ahead" tests to infer the changes that have occurred and correctly maintain transport statistics.

4.4.6 Application to true oldspace

Finally, it is should be noted that the technique of scanning reordering oldspace for transport statistics, which we use to quantify the locality-improving effect of reordering—increasing the density of active objects—can also be used without modification for another purpose. We can and have applied the same scanning procedure on true oldspace, after scavenging is completed but before oldspace is reclaimed, to quantify the locality-improving effect of garbage collection—increasing the density of reachable objects.

4.5 Experimental Results

The method discussed above to measure the effect of reordering was applied to system workload and to the SRW program.

4.5.1 System workload

The first workload considered was an interactive session. The session lasted several hours and involved editing files, compiling, reading mail, issuing many *Command Processor* commands, and exercising many interactive utilities, such as the *File System Editor*, *Document Examiner*, *Flavor Examiner*, *Inspector*, *Terminal*, *Peek*, *Namespace Editor*, *Notifications*, and so on. Basically, this workload consists of "system" programs rather than user applications.

The parts of memory considered for reordering consisted of level 0 of `WORKING-STORAGE-AREA` and 20 other selected areas, totalling about 65% of the total initial virtual memory usage. The areas not considered included small areas (less than about 10,000 words) and unreorderable areas, e.g., areas containing stacks, non-Lisp-objects, and areas specifically prevented from being flipped. An important area *not* considered for reordering was `COMPILED-FUNCTION-AREA`, since Andre found that the best strategy was to preserve (or, after many redefinitions, restore) source-file ordering of compiled function objects [1]. This conclusion is similar to that of Ferrari [9] for a static language system.

Since the selected areas were flipped for reordering at the start of the session, we refer to the objects contained therein as *pre-existing*, to distinguish them from objects created during the session. In most of these areas, among the pre-existing objects, there is very little or no garbage since they consist primarily of system objects present in virtual memory when the system is booted. Because of the low percentage of garbage, it is mainly through reordering rather than garbage collection that we can hope to improve locality among these objects.

Tables 4.2–4.5 present transport statistics taken at the end of the interactive session. Table 4.2 shows, for each area,

- the size of reordering oldspace;
- the total size of the objects accessed during the session, expressed in number of words (WOR) and as a percentage of reordering oldspace;
- the number of oldspace pages (PAG_{old}) and copyspace pages (PAG_{copy}) occupied by the objects accessed during the session; and
- the reduction in working set size (Δ), and the compression ratio (ρ).

The areas are ranked in order of decreasing Δ.

Table 4.2 Transport statistics for pre-existing objects under system workload.

Pre-existing objects in area	All objects							
	Flipped (words)	Accessed (words)	Percent accessed	PAG_{old} (pages)	PAG_{copy} (pages)	Δ (pages)	ρ (percent)	
FLAVOR-AREA	1846875	315199	17.1	2131	1233	898	42.1	█
WORKING-STORAGE-AREA	1544846	255215	16.5	1544	998	546	35.4	█
PNAME-AREA	487163	9481	1.9	561	38	523	93.2	█
DEBUG-INFO-AREA	1113525	61775	5.5	607	242	365	60.1	█
PROPERTY-LIST-AREA	174742	13568	7.8	324	53	271	83.6	█
WHO-CALLS-DATABASE-AREA	100287	23609	23.5	282	93	189	67.0	█
SAGE-COMPLETION-AREA	171894	98765	57.5	575	387	188	32.7	█
PATHNAME-AREA	438160	86206	19.7	508	338	170	33.5	█
PERMANENT-STORAGE-AREA	32244	12383	38.4	101	49	52	51.5	█
EDITOR-LINE-AREA	521039	110430	21.2	481	432	49	10.2	▏
PRESENTATION-AREA	222889	3553	1.6	47	15	32	68.1	█
PRESENTATION-TYPE-AREA	15416	7093	46.0	57	28	29	50.9	█
NAMESPACE-OBJECT-AREA	12883	5395	41.9	50	22	28	56.0	█
HANDLER-DYNAMIC-AREA	118249	89763	75.9	373	352	21	5.6	▏
SHEET-AREA	41354	12514	30.3	70	50	20	28.6	█
HANDLER-TABLE-AREA	11198	5780	51.6	41	23	18	43.9	█
EDITOR-NODE-AREA	20590	12948	62.9	66	52	14	21.2	█
FONT-AREA	56919	17955	31.5	84	71	13	15.5	█
PKG-AREA	375300	347032	92.5	1366	1356	10	0.7	
DISK-ARRAY-AREA	74786	42412	56.7	171	166	5	2.9	
BIT-ARRAY-AREA	498651	402052	80.6	1575	1571	4	0.3	
Total	7879010	1933128	24.5	11014	7569	3445	31.3	█

ORIGINAL PAGE IS
OF POOR QUALITY

Table 4.3 Transport statistics for pre-existing structure objects under system workload.

Pre-existing objects in area	Structures						
	Flipped (words)	Accessed (words)	Percent accessed	PAG_{old} (pages)	PAG_{copy} (pages)	Δ (pages)	ρ (percent)
FLAVOR-AREA	1266200	283690	22.4	1551	1109	442	28.5
WORKING-STORAGE-AREA	1411164	250756	17.8	1457	980	477	32.7
PNAME-AREA	487163	9481	1.9	561	38	523	93.2
DEBUG-INFO-AREA	115303	42439	36.8	205	166	39	19.0
PROPERTY-LIST-AREA							
WHO-CALLS-DATABASE-AREA	49098	19515	39.7	88	77	11	12.5
SAGE-COMPLETION-AREA	108474	65283	60.2	401	256	145	36.2
PATHNAME-AREA	372659	85304	22.9	476	334	142	29.8
PERMANENT-STORAGE-AREA	10447	8121	77.7	37	32	5	13.5
EDITOR-LINE-AREA	482130	104402	21.7	450	408	42	9.3
PRESENTATION-AREA	215815	3517	1.6	46	14	32	69.6
PRESENTATION-TYPE-AREA	15416	7093	46.0	57	28	29	50.9
NAMESPACE-OBJECT-AREA	4531	2229	49.2	18	9	9	50.0
HANDLER-DYNAMIC-AREA	107318	89147	83.1	359	349	10	2.8
SHEET-AREA	39604	12326	31.1	63	49	14	22.2
HANDLER-TABLE-AREA							
EDITOR-NODE-AREA	12070	12070	100.0	48	48	0	0.0
FONT-AREA	56919	17955	31.5	84	71	13	15.5
PKG-AREA	374598	346869	92.6	1362	1355	7	0.5
DISK-ARRAY-AREA	74786	42412	56.7	171	166	5	2.9
BIT-ARRAY-AREA	498651	402052	80.6	1575	1571	4	0.3
Total	5702346	1804661	31.6	9009	7060	1949	21.6

Table 4.4 Transport statistics for pre-existing list objects under system workload.

Pre-existing objects in area	Lists							
	Flipped (words)	Accessed (words)	Percent accessed	PAG_{old} (pages)	PAG_{copy} (pages)	Δ (pages)	ρ (percent)	
FLAVOR-AREA	580675	31509	5.4	580	124	456	78.6	█
WORKING-STORAGE-AREA	133682	4459	3.3	87	18	69	79.3	█
PNAME-AREA								
DEBUG-INFO-AREA	998222	19336	1.9	402	76	326	81.1	█
PROPERTY-LIST-AREA	174742	13568	7.8	324	53	271	83.6	█
WHO-CALLS-DATABASE-AREA	51189	4094	8.0	194	16	178	91.8	█
SAGE-COMPLETION-AREA	63420	33482	52.8	174	131	43	24.7	█
PATHNAME-AREA	65501	902	1.4	32	4	28	87.5	█
PERMANENT-STORAGE-AREA	21797	4262	19.6	64	17	47	73.4	█
EDITOR-LINE-AREA	38909	6028	15.5	31	24	7	22.6	█
PRESENTATION-AREA	7074	36	0.5	1	1	0	0.0	
PRESENTATION-TYPE-AREA								
NAMESPACE-OBJECT-AREA	8352	3166	37.9	32	13	19	59.4	█
HANDLER-DYNAMIC-AREA	10931	616	5.6	14	3	11	78.6	█
SHEET-AREA	1750	188	10.7	7	1	6	85.7	█
HANDLER-TABLE-AREA	11198	5780	51.6	41	23	18	43.9	█
EDITOR-NODE-AREA	8520	878	10.3	18	4	14	77.8	█
FONT-AREA								
PKG-AREA	702	163	23.2	4	1	3	75.0	█
DISK-ARRAY-AREA								
BIT-ARRAY-AREA								
Total	2176664	128467	5.9	2005	509	1496	74.6	█

ORIGINAL PAGE IS
OF POOR QUALITY

Table 4.5 Other transport statistics for pre-existing objects under system workload.

Pre-existing objects in area	Structures					Lists
	Count	Min size (words)	Max size (words)	Mean size (words)	Std. dev. (words)	Normal conses
FLAVOR-AREA	3716	1	3095	76.3	309.9	90
WORKING-STORAGE-AREA	6594	1	10404	38.0	327.9	777
PNAME-AREA	2085	2	16	4.5	2.2	
DEBUG-INFO-AREA	109	2	41568	389.3	3980.7	73
PROPERTY-LIST-AREA						226
WHO-CALLS-DATABASE-AREA	24	11	8437	813.1	1845.2	2013
SAGE-COMPLETION-AREA	6784	2	10993	9.6	182.0	70
PATHNAME-AREA	1128	2	42887	75.6	1348.2	11
PERMANENT-STORAGE-AREA	602	1	4005	13.5	163.0	92
EDITOR-LINE-AREA	4224	11	123	24.7	15.4	0
PRESENTATION-AREA	428	3	130	8.2	10.4	18
PRESENTATION-TYPE-AREA	173	41	41	41.0	0.0	
NAMESPACE-OBJECT-AREA	332	2	17	6.7	4.1	12
HANDLER-DYNAMIC-AREA	166	7	20437	537.0	2413.7	5
SHEET-AREA	168	5	186	73.4	57.3	57
HANDLER-TABLE-AREA						0
EDITOR-NODE-AREA	710	17	17	17.0	0.0	67
FONT-AREA	106	5	1389	169.4	177.2	
PKG-AREA	143	1	33058	2425.7	5112.6	17
DISK-ARRAY-AREA	68	15	1165	623.7	547.4	
BIT-ARRAY-AREA	21	749	25439	19145.3	10050.1	
Total	27581	1	42887	65.4	860.8	3528

The results show that, for many areas, only a small fraction of all reachable objects was referenced: of the total amount flipped into oldspace, only 24% was transported. However, the fraction transported varied considerably among the different areas. For example, in PNAME-AREA, which contains the print names (strings) of symbols, only 2% was transported, while in PKG-AREA, which primarily contains package objects, 92% was transported.

Reordering resulted in a reduction of working set size by 31%. However, this percentage is biased upward by a few "bad" areas. Significant reductions in working set size were achieved in almost all areas. In PNAME-AREA, the 93% reduction is particularly dramatic, and indicates high fragmentation among symbol print names. However, there are five areas with $\rho \leq 10\%$ and which are therefore unsuitable for reordering.[4] (The specific reasons will be discussed shortly.) If we remove these five areas from the analysis, the overall fraction of oldspace transported drops from 24% to 15%, and the compression ratio rises from 31% to 48%. That is, over the areas which are reasonable candidates for reordering, a decrease in the working set size by one-half was realized.

The next two tables, Tables 4.3 and 4.4, present the same information as the first, but broken down into structure and list objects.

For most areas, as well as overall, list objects are more fragmented than structure objects. Over all the areas, the compression ratio ρ for lists was 75% while for structures it was 22%. However, the absolute reduction Δ in working set size due to reordering of structures was greater (1949 pages) than that due to reordering of lists (1496 pages) because of the much larger number of structure pages referenced.

The difference between structure and list objects in the effectiveness of reordering suggests that the memory management system should be capable of reordering *only* the regions of an area containing lists. Currently, our system does not distinguish between list and structure regions and flips both types when requested to reorder an area (or ephemeral level). Good examples for which such a flexibility would be useful are DEBUG-INFO-AREA and *WHO-CALLS-DATABASE-AREA*. In these areas, reordering lists yields significant benefit, while reordering structures yields little benefit but has significant cost (as measured by the number of words transported) and is therefore undesirable. Note that selectively flipping list or struc-

[4]The areas are EDITOR-LINE-AREA, *HANDLER-DYNAMIC-AREA*, PKG-AREA, DISK-ARRAY-AREA, and BIT-ARRAY-AREA.

ture regions of an area applies not only to reordering but to garbage collection as well. As in generation-based garbage collection, the underlying theme is that of improving efficiency by expending memory management effort only on selected portions of virtual memory.

Table 4.5 shows other statistics on the objects accessed during the interactive session. For structure objects, the distribution of object size is indicated. For list objects, the number of normally coded *conses* is shown.

The size distribution for structures is helpful in understanding the reordering results of the various areas. Four of the five "bad" areas mentioned earlier ($\rho \leq 10\%$) are seen to have mean structure sizes much larger than the size of a page (256 words). This is also true for the two areas DEBUG-INFO-AREA and *WHO-CALLS-DATABASE-AREA* mentioned as good candidates for list reordering but not structure reordering. For these areas, the potential for fragmentation, which is predicated on objects being smaller than a page, is therefore greatly diminished. The fifth "bad" area, EDITOR-LINE-AREA, has small objects (a mean of 25 words), but is apparently not fragmented. This behavior is explained by noting that this area contains the text of editor buffers, and the objects (strings) are already well-ordered (sequentially) in memory.

4.5.2 SRW program

The effectiveness of reordering was also measured while running the SRW program. The same 21 areas as in the system workload experiment were flipped for reordering at the start of execution; these areas contain the pre-existing objects.

However, we also considered the objects created in WORKING-STORAGE-AREA *during* execution. These objects are normally created in the ephemeral part of WORKING-STORAGE-AREA. Assuming that WORKING-STORAGE-AREA is configured to have two ephemeral levels, one way to evaluate reordering performance for (some of) these objects is to initiate a reordering of the second level at some arbitrary time into the execution.

Instead, a more systematic approach was taken. We group the objects created during each well-defined phase of execution and reorder the groups separately. In general, this method makes it possible to relate any interesting results to the phase of execution. The grouping was done as follows.

The SRW program goes through six phases, called *Load*, *File1*, ..., *File5*. The first phase consists of loading the program into virtual memory and performing any initializations; the remaining five phases correspond to processing each of five input data files. WORKING-STORAGE-AREA was configured to have $6+2$ ephemeral levels, with the lower six levels reserved for the objects from each phase, and with the highest two levels serving as the normal "screening levels." The lower of these screening levels was made to be "sticky" so that any objects surviving a garbage collection of this level would remain at the level. All levels were initially empty, having been garbage collected before starting the application, with surviving objects being transported to level 0 (and therefore contributing to the population of pre-existing objects).

During execution, the screening levels were garbage collected as usual at some reasonable frequency, but at the end of phase i, both screening levels were garbage collected, with surviving objects being transported to the level reserved for phase i. That reserved level was then immediately flipped for reordering.[5]

The measurements taken at the end of the last phase (and of the program) are presented in Tables 4.6–4.9 for the pre-existing objects and in Tables 4.10–4.13 for the objects created during execution.

In general, the results for SRW are similar to those for the system workload, and even more striking. The application references a far smaller percentage of the accessible objects, and the objects referenced are more widely scattered among pages. For the pre-existing objects, over all areas, only 7% of all memory flipped into reordering oldspace was accessed (Table 4.6). The compression ratio achieved was 43%. Several areas were completely untouched, since the application did not involve the use of the editor, windowing system, or *Document Examiner*. As expected, because they contain large objects, the areas which were unsuitable for reordering under the system workload remained unsuitable for the same reason. Discounting their effect, the remaining areas have $\rho = 58\%$.

As in the system workload, lists were more fragmented than structures, but the reduction in working set size due to structures was greater because of the larger number of structure pages referenced (Tables 4.7 and 4.8). For lists, $\Delta = 890$ pages and $\rho = 88.4\%$, while for structures,

[5] An alternative to copying the objects which survive a particular phase into a unique ephemeral *level* in WORKING-STORAGE-AREA, is to copy them to a unique (nonephemeral) *area*. This alternative will negate the possibility of later garbage collecting them efficiently using the ephemeral garbage collector. However, it may be the better alternative if the objects are long-lived anyway, and if maintaining large populations in many ephemeral levels results in a large number of inter-level pointers—and a corresponding increase in the size of the ephemeral root set tables, and degradation in ephemeral garbage collector performance.

Table 4.6 Transport statistics for pre-existing objects under SRW.

Pre-existing objects in area	All objects						
	Flipped (words)	Accessed (words)	Percent accessed	PAG_{old} (pages)	PAG_{copy} (pages)	Δ (pages)	ρ (percent)
WORKING-STORAGE-AREA	2796059	172676	6.2	1607	676	931	57.9
PNAME-AREA	577180	7353	1.3	300	29	271	90.3
WHO-CALLS-DATABASE-AREA	162883	28385	17.4	345	112	233	67.5
FLAVOR-AREA	2093542	57211	2.7	436	225	211	48.4
DEBUG-INFO-AREA	1583483	43055	2.7	279	169	110	39.4
PROPERTY-LIST-AREA	218901	1817	0.8	116	8	108	93.1
PATHNAME-AREA	649074	54388	8.4	256	213	43	16.8
NAMESPACE-OBJECT-AREA	18275	5368	29.4	61	22	39	63.9
PERMANENT-STORAGE-AREA	40475	1355	3.3	40	7	33	82.5
PKG-AREA	600872	250566	41.7	1006	980	26	2.6
HANDLER-DYNAMIC-AREA	134557	37052	27.5	170	146	24	14.1
PRESENTATION-AREA	438659	1087	0.2	23	5	18	78.3
PRESENTATION-TYPE-AREA	15539	2378	15.3	28	10	18	64.3
HANDLER-TABLE-AREA	11332	1433	12.6	20	6	14	70.0
SHEET-AREA	39360	1421	3.6	16	7	9	56.3
FONT-AREA	73150	3148	4.3	18	13	5	27.8
DISK-ARRAY-AREA	74786	33225	44.4	134	130	4	3.0
SAGE-COMPLETION-AREA	200320	0	0.0	0	0	0	-
EDITOR-LINE-AREA	4809	0	0.0	0	0	0	-
BIT-ARRAY-AREA	430730	0	0.0	0	0	0	-
Total	10163986	701918	6.9	4855	2758	2097	43.2

Table 4.7 Transport statistics for pre-existing structure objects under SRW.

Pre-existing objects in area	Structures						
	Flipped (words)	Accessed (words)	Percent accessed	PAG_{old} (pages)	PAG_{copy} (pages)	Δ (pages)	ρ (percent)
WORKING-STORAGE-AREA	2281090	161812	7.1	1304	633	671	51.5
PNAME-AREA	577180	7353	1.3	300	29	271	90.3
WHO-CALLS-DATABASE-AREA	85707	25526	29.8	107	100	7	6.5
FLAVOR-AREA	1329596	52315	3.9	319	205	114	35.7
DEBUG-INFO-AREA	135652	41572	30.6	164	163	1	0.6
PROPERTY-LIST-AREA							
PATHNAME-AREA	549509	54258	9.9	242	212	30	12.4
NAMESPACE-OBJECT-AREA	6178	2481	40.2	24	10	14	58.3
PERMANENT-STORAGE-AREA	14200	60	0.4	8	1	7	87.5
PKG-AREA	599166	250380	41.8	999	979	20	2.0
HANDLER-DYNAMIC-AREA	125325	36985	29.5	164	145	19	11.6
PRESENTATION-AREA	418657	1087	0.3	23	5	18	78.3
PRESENTATION-TYPE-AREA	15539	2378	15.3	28	10	18	64.3
HANDLER-TABLE-AREA							
SHEET-AREA	37688	1393	3.7	14	6	8	57.1
FONT-AREA	73150	3148	4.3	18	13	5	27.8
DISK-ARRAY-AREA	74786	33225	44.4	134	130	4	3.0
SAGE-COMPLETION-AREA	113328	0	0.0	0	0	0	-
EDITOR-LINE-AREA	4809	0	0.0	0	0	0	-
BIT-ARRAY-AREA	430730	0	0.0	0	0	0	-
Total	6872290	673973	9.8	3848	2641	1207	31.4

Table 4.8 Transport statistics for pre-existing list objects under SRW.

Pre-existing objects in area	Lists						
	Flipped (words)	Accessed (words)	Percent accessed	PAG_{old} (pages)	PAG_{copy} (pages)	Δ (pages)	ρ (percent)
WORKING-STORAGE-AREA	514969	10864	2.1	303	43	260	85.8
PNAME-AREA							
WHO-CALLS-DATABASE-AREA	77176	2859	3.7	238	12	226	95.0
FLAVOR-AREA	763946	4896	0.6	117	20	97	82.9
DEBUG-INFO-AREA	1447831	1483	0.1	115	6	109	94.8
PROPERTY-LIST-AREA	218901	1817	0.8	116	8	108	93.1
PATHNAME-AREA	99565	130	0.1	14	1	13	92.9
NAMESPACE-OBJECT-AREA	12097	2887	23.9	37	12	25	67.6
PERMANENT-STORAGE-AREA	26275	1295	4.9	32	6	26	81.3
PKG-AREA	1706	186	10.9	7	1	6	85.7
HANDLER-DYNAMIC-AREA	9232	67	0.7	6	1	5	83.3
PRESENTATION-AREA	20002	0	0.0	0	0	0	-
PRESENTATION-TYPE-AREA							
HANDLER-TABLE-AREA	11332	1433	12.6	20	6	14	70.0
SHEET-AREA	1672	28	1.7	2	1	1	50.0
FONT-AREA							
DISK-ARRAY-AREA							
SAGE-COMPLETION-AREA	86992	0	0.0	0	0	0	-
EDITOR-LINE-AREA							
BIT-ARRAY-AREA							
Total	3291696	27945	0.8	1007	117	890	88.4

Table 4.9 Other transport statistics for pre-existing objects under SRW.

Pre-existing objects in area	Structures					Lists
	Count	Min size (words)	Max size (words)	Mean size (words)	Std. dev. (words)	Normal conses
WORKING-STORAGE-AREA	3589	1	65537	45.1	1130.1	3670
PNAME-AREA	1739	2	14	4.2	1.8	
WHO-CALLS-DATABASE-AREA	12	11	11437	2127.2	3814.6	1400
FLAVOR-AREA	624	1	3095	83.8	241.9	23
DEBUG-INFO-AREA	2	4	41568	20786.0	29390.2	8
PROPERTY-LIST-AREA						61
PATHNAME-AREA	64	2	42887	847.8	5438.1	20
NAMESPACE-OBJECT-AREA	316	2	236	7.9	13.6	7
PERMANENT-STORAGE-AREA	10	2	7	6.0	2.1	36
PKG-AREA	89	1	93214	2813.3	11046.0	27
HANDLER-DYNAMIC-AREA	116	7	11437	318.8	1338.2	0
PRESENTATION-AREA	174	5	21	6.2	2.0	0
PRESENTATION-TYPE-AREA	58	41	41	41.0	0.0	
HANDLER-TABLE-AREA						0
SHEET-AREA	13	5	186	107.2	63.2	8
FONT-AREA	17	5	573	185.2	153.7	
DISK-ARRAY-AREA	55	15	1165	604.1	555.9	
SAGE-COMPLETION-AREA	0	-	-	-	-	0
EDITOR-LINE-AREA	0	-	-	-	-	
BIT-ARRAY-AREA	0	-	-	-	-	
Total	6878	1	93214	98.0	1710.2	5260

Table 4.10 Transport statistics for new objects under SRW.

Objects created during phase	All objects							
	Flipped (words)	Accessed (words)	Percent accessed	PAG_{old} (pages)	PAG_{copy} (pages)	Δ (pages)	ρ (percent)	
Load	374847	5667	1.5	60	24	36	60.0	█
File1	94480	1111	1.2	56	6	50	89.3	█
File2	139670	14126	10.1	162	56	106	65.4	█
File3	820323	22405	2.7	360	89	271	75.3	█
File4	1242786	31679	2.5	409	125	284	69.4	█
Total	**2672106**	**74988**	**2.8**	**1047**	**300**	**747**	**71.3**	█

Table 4.11 Transport statistics for new structure objects under SRW.

Objects created during phase	Structures							
	Flipped (words)	Accessed (words)	Percent accessed	PAG_{old} (pages)	PAG_{copy} (pages)	Δ (pages)	ρ (percent)	
Load	271782	4871	1.8	42	20	22	52.4	█
File1	89382	587	0.7	46	3	43	93.5	█
File2	131503	13001	9.9	149	51	98	65.8	█
File3	784826	20249	2.6	344	80	264	76.7	█
File4	1186954	27746	2.3	385	109	276	71.7	█
Total	2464447	66454	2.7	966	263	703	72.8	█

Table 4.12 Transport statistics for new list objects under SRW.

Objects created during phase	Lists							
	Flipped (words)	Accessed (words)	Percent accessed	PAG_{old} (pages)	PAG_{copy} (pages)	Δ (pages)	ρ (percent)	
Load	103065	796	0.8	18	4	14	77.8	█
File1	5098	524	10.3	10	3	7	70.0	█
File2	8167	1125	13.8	13	5	8	61.5	█
File3	35497	2156	6.1	16	9	7	43.8	█
File4	55832	3933	7.0	24	16	8	33.3	█
Total	207659	8534	4.1	81	37	44	54.3	█

Table 4.13 Other transport statistics for new objects under SRW.

Objects created during phase	Structures					Lists
	Count	Min size (words)	Max size (words)	Mean size (words)	Std. dev. (words)	Normal conses
Load	177	2	1304	27.5	100.7	196
File1	168	2	66	3.5	5.2	25
File2	323	2	12037	40.3	669.6	125
File3	859	2	17677	23.6	603.0	51
File4	635	2	25837	43.7	1025.2	670
Total	2162	2	25837	30.7	721.5	1067

$\Delta = 1207$ pages and $\rho = 31\%$. The data also clearly shows the desirability of reordering lists but not structures in DEBUG-INFO-AREA and *WHO-CALLS-DATABASE-AREA*.

For the objects created during execution, an overall ρ of 71% was realized, and each of the object populations corresponding to the execution phases had $\rho \geq 60\%$ (Table 4.10).

Transport statistics were collected not only at the end of the program but also at the end of each phase to observe the time variation in the measures. In particular, we were interested in testing the assumption made in Section 4.4.2, namely, that Δ is indicative of the reduction in the average working set size over an arbitrary subinterval of (t_{flip}, t_{eval}), because of the long duration over which reordering occurs and the consequent high probability of rereferencing. From Table 4.14, we see that the amount of objects transported in each population grows only very slowly after the first measurement of the population. This behavior suggests (but does not prove) that a significant amount of repeated accessing of objects is occurring, which is consistent with the assertion regarding Δ.

Table 4.14 Time variation in transport statistics under SRW.

Measured at end of phase	All objects						
	Flipped (words)	Accessed (words)	Percent accessed	PAG_{old} (pages)	PAG_{copy} (pages)	Δ (pages)	ρ (percent)
Pre-existing objects							
Load	10163986	657385	6.5	4307	2580	1727	40.1
File1		688396	6.8	4699	2704	1995	42.5
File2		690454	6.8	4724	2712	2012	42.6
File3		694705	6.8	4786	2728	2058	43.0
File4		696439	6.9	4800	2735	2065	43.0
File5		701918	6.9	4855	2758	2097	43.2
Objects created during phase *Load*							
File1	374847	5200	1.4	56	21	35	62.5
File2		5617	1.5	57	23	34	59.6
File3		5651	1.5	58	24	34	58.6
File4		5664	1.5	60	24	36	60.0
File5		5667	1.5	60	24	36	60.0
Objects created during phase *File1*							
File2	94480	945	1.0	53	5	48	90.6
File3		961	1.0	54	5	49	90.7
File4		1099	1.2	56	6	50	89.3
File5		1111	1.2	56	6	50	89.3
Objects created during phase *File2*							
File3	139670	13245	9.5	88	53	35	39.8
File4		13946	10.0	153	56	97	63.4
File5		14126	10.1	162	56	106	65.4
Objects created during phase *File3*							
File4	820323	20960	2.6	240	83	157	65.4
File5		22405	2.7	360	89	271	75.3
Objects created during phase *File4*							
File5	1242786	31679	2.5	409	125	284	69.4

4.6 Summary

Dynamic reordering is an attempt to improve the locality of reference by grouping together objects which are being actively used. The basic architectural support required for its realization is the same as that for incremental copying garbage collection, namely, the read barrier and efficient handling of invisible pointers.

A new memory management system has been designed and implemented for the Symbolics Lisp computer, which integrates generation-based garbage collection with dynamic reordering. The problem of preserving object order when garbage collecting is solved approximately by a scheme for maintaining togetherness, and exactly by an operation called compaction. Compaction reclaims memory taken up by forwarding pointers created during reordering, but not by other garbage.

The new system requires no changes to hardware, and in particular, to the read barrier. The only change made to the existing virtual memory organization is to define and allow the simultaneous existence of two types of oldspace: true oldspace serves the traditional role as memory being garbage collected; reordering oldspace consists of memory being reordered.

A method, called scanning for transport statistics, was presented for measuring the intrinsic effectiveness of reordering. The method provides an evaluation which is independent of main memory size and allows the reorderability of different memory areas to be determined. Reordering oldspace is scanned for objects which have been transported. The scan yields the number of oldspace pages containing the transported objects and statistics on their sizes, from which can be computed the reduction in working set size due to reordering, and the density with which the objects are packed into pages. The algorithm for parsing oldspace solves the problem of noninvariance of object representation with respect to forwarding.

The results from two experiments, one involving interactive system workload and the other a large application, show a significant reduction in working set size due to reordering, or equivalently, a significant amount of fragmentation among the populations of objects considered. Different memory areas exhibited varying reordering performance. Most areas considered were suitable for reordering; the exceptions were areas containing many large objects or which were already initially well-ordered. Over the suitable areas, a compression in working set of 48% and 58% was measured for the two workloads.

List space exhibited a greater relative reduction in working set size than structure space by a factor of about three, but the absolute reduction due to structure space was greater because of the larger number of structure pages being accessed.

Chapter 5

Conclusions

Dynamic reordering has been incorporated into an existing generation-based garbage-collecting memory management system. The new system supports schemes for preserving object order in virtual memory during garbage collection, both approximately and exactly.

We have presented a technique, called scanning for transport statistics, for evaluating the effectiveness of reordering, independent of main memory size. In this method, a scan of reordering oldspace yields the number of oldspace pages containing the transported objects and statistics on their sizes, from which can be computed the reduction in working set size due to reordering. The relative reduction in working set size is also a measure of the density with which objects are packed into pages, and the extent to which the problem that reordering attempts to address actually exists. Our algorithm for parsing oldspace solves the problem of noninvariance of object representation with respect to forwarding.

While motivated by reordering, scanning for transport statistics can be viewed as a general technique which can be used to evaluate locality improvement in any situation in which objects are dynamically reorganized, including normal copying garbage collection.

Two experiments, one involving interactive system workload and the other a large application, have been conducted and the results show reductions in working set size of 48% and 58% due to reordering, or equivalently, a significant amount of fragmentation among the populations of objects considered. The use of the technique to measure the reorderability of selected portions of memory has been demonstrated. Memory areas suitable for reordering were identified. Relative compression in working set size was greater for list space than structure space, by a factor of about three. Results for certain areas suggest differential treatment of list space and structure space.

5.1 Symmetries Between Garbage Collection and Dynamic Reordering

Generation-based garbage collection and dynamic reordering are two techniques for improving the efficiency of memory management in Lisp and similar dynamic language systems. Both are based on empirical characteristics of objects; the former exploits the phenomenon of short-lifetime and the infrequency of pointers from older to younger objects; the latter exploits the phenomenon of active object fragmentation. Both require similar mechanisms in the underlying memory management system. Garbage collection improves locality by compacting the accessible objects, while reordering improves locality by compacting the actively used objects. For the younger and smaller generations, garbage collection is an efficient management technique, while for the older, larger, and stable generations, reordering may be the preferred technique to avoid the expense of garbage collecting the gigantic virtual memory systems of today and of the future.

5.2 Suggestions for Future Work

The experimental results on the tuning of generation parameters suggest the possibility of adaptive control of level capacity. Since the conditions associated with nonoptimality in capacity have been identified and are easily measured, and since the range of good values for capacity can span a fairly broad fraction of main memory size, rather than have a fixed, user-specified capacity, an allowable range of capacity values could be specified, and the system could dynamically adjust the current value using some appropriate adaptive algorithm.

Dynamic reordering, like incremental garbage collection, is currently prohibitive on systems without processor support for the read barrier and invisible pointer handling. Strategies for efficiently providing these or alternative mechanisms on conventional architectures should be investigated.

After an arbitrary number of reorderings of a given part of memory without any intervening garbage collection or compaction, reordering oldspace will contain a history of accesses made during each reordering interval. In collecting transport statistics, we have been concerned only with the objects forwarded directly to copyspace, and have ignored other internal forwardings.

Doing so yields an evaluation of the most recently initiated reordering interval. However, by considering statistics for other transport internal to reordering oldspace, it is possible to evaluate previous reordering intervals, and to evaluate various other possible scenarios, e.g., not having initiated some specified subset of previous reorderings. The utility of this extension is an open question.

Policies for the automatic initiation of reordering, the counterpart of threshold-based policies for automatic initiation of garbage collection, are an area for future research. With reference to preserving object order under copying garbage collection, the related issues of when to maintain order approximately or exactly, and for how long, remain to be addressed.

References

[1] David Lewis Andre. Paging in Lisp programs. Master's thesis, Department of Computer Science, University of Maryland, 1986.

[2] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software—Practice and Experience*, 19(2):171–183, February 1989.

[3] Jean-Loup Baer and Gary R. Sager. Dynamic improvement of locality in virtual memory systems. *IEEE Transactions on Software Engineering*, SE-2(1):54–62, March 1976.

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

[5] C.J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[6] Douglas W. Clark and C. Cordell Green. An empirical study of list structure in Lisp. *Communications of the ACM*, 20(2):78–87, February 1977.

[7] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, September 1988.

[8] Peter J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[9] Domenico Ferrari. Improving locality by critical working sets. *Communications of the ACM*, 17(11):614–620, November 1974.

[10] Domenico Ferrari. Tailoring programs to models of program behavior. *IBM Journal of Research and Development*, 19(3):244–251, May 1975.

[11] Domenico Ferrari. The improvement of program behavior. *Computer*, 9(11):39–47, November 1976.

[12] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*. The MIT Press, Cambridge, MA, 1985.

[13] D. J. Hatfield and J. Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 10(3):168–192, October 1971.

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

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

[16] David A. Moon. Architecture of the Symbolics 3600. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 76–83, July 1985.

[17] Chih-Jui Peng and Gurindar S. Sohi. Cache memory design considerations to support languages with dynamic heap allocation. Technical Report 860, Computer Sciences Department, University of Wisconsin, Madison, WI, July 1989.

[18] Robert A. Shaw. Empirical analysis of a Lisp system. Technical Report CSL-TR-88-351, Computer Systems Laboratory, Stanford University, February 1988.

[19] Patrick G. Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. B.S. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1988.

[20] James W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Computer Systems*, 2(2):155–180, May 1984.

[21] Guy L. Steele Jr. *Common Lisp, The Language*. Digital Equipment Corporation, 1984.

[22] Peter Steenkiste and John Hennessy. Lisp on a reduced-instruction-set processor: Characterization and optimization. *Computer*, 21(7):34–45, July 1988.

[23] George S. Taylor, Paul N. Hilfinger, James R. Larus, David A. Patterson, and Benjamin G. Zorn. Evaluation of the SPUR Lisp architecture. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 444–452, June 1986.

[24] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, May 1984.

[25] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. *ACM SIGPLAN Notices*, 23(11):1–17, November 1988.

[26] Jon L White. Address/memory management for a gigantic Lisp environment or, GC considered harmful. In *Conference Record of the 1980 Lisp Conference*, pages 119–127, August 1980.

[27] Paul R. Wilson. A simple bucket-brigade advancement mechanism for generation-based garbage collection. *ACM SIGPLAN Notices*, 24(5):38–46, May 1989.

[28] Paul R. Wilson and Thomas G. Moher. A "card-marking" scheme for controlling inter-generational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices*, 24(5):87–92, May 1989.

[29] Paul R. Wilson and Thomas G. Moher. Demonic memory for process histories. In *Proceedings of the 1989 ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 330–343, 1989. Published as *ACM SIGPLAN Notices*, Vol. 24, No. 7, July 1989.

[30] Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In *OOPSLA'89 Conference Proceedings*, pages 23–35, 1989. Published as *ACM SIGPLAN Notices*, Vol. 24, No. 10, October 1989.

[31] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 87–98, June 1990.

Vita

Rene Llames was born in [redacted], on [redacted]. He received the B.S. degree in Electrical Engineering (summa cum laude) from the University of the Philippines in 1982 and the M.S. and Ph.D. degrees in Electrical Engineering from the University of Illinois, Urbana, Illinois, in 1985 and 1991, respectively.

While pursuing graduate studies at the University of Illinois, he was a teaching assistant in the Department of Electrical Engineering in 1982, a research assistant in the Department of Computer Science in 1983, and a research assistant in the Coordinated Science Laboratory from 1984 to 1990. He was with the Computer Sciences Center of Honeywell, Bloomington, Minnesota in the summer of 1984, where he developed a simulator for a dataflow implementation of the OPS5 production system language. He was with ESL, Sunnyvale, California, in the summer of 1985, where he helped design a parallel systolic array computer.

Mr. Llames will now join IBM Corporation in Austin, Texas.