A MEASUREMENT-BASED STUDY OF MEMORY USAGE AND GARBAGE COLLECTION IN A LISP SYSTEM

Rene L. Llames
Ravi K. Iyer

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Approved for Public Release. Distribution Unlimited.
An empirical analysis of paging, memory allocation, and garbage collection in a Lisp system is presented, based on sampled memory system activity. Workload of varying complexity, consisting of the Boyer benchmark and QPE, a large AI program, is used to derive empirical models for garbage collector performance. The models allow a prediction of garbage collection time for a given amount of scanning and copying work, page faults, and other software overhead. The models account for greater than 90% of the variation in collection time. With some exceptions, the models correctly predicted, within a 95% confidence interval, the mean collection time of a particular run. The models also express the high time cost of a page fault cost; show how much longer copying takes than scanning a word; and show that the cost to scan a word depends on which class of spaces to be scanned it belongs to. In addition, the time-varying characteristics of memory allocation, survival, collector work, and efficiency are presented. Collector efficiency is quantified as words of discovered garbage per unit time (or work) expended by the collector. The variation of collector efficiency over the execution of the measured programs suggests the potential for improved..
Abstract (continued)

performance by more closely matching garbage collection policy with program characteristics.
A Measurement-based Study of Memory Usage and Garbage Collection in a Lisp System

Rene L. Llames and Ravi K. Iyer
Center for Reliable and High Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 West Springfield Avenue, Urbana, Illinois 61801
Phone: (217) 333-9732
iyer@bach.csg.uiuc.edu

Abstract

An empirical analysis of paging, memory allocation, and garbage collection in a Lisp system is presented, based on sampled memory system activity. Workload of varying complexity, consisting of the Boyer benchmark and QPE, a large AI program, is used to derive empirical models for garbage collector performance. The models allow a prediction of garbage collection time for a given amount of scanning and copying work, page faults, and other software overhead. The models account for greater than 90% of the variation in collection time. With some exceptions, the models correctly predicted, within a 95% confidence interval, the mean collection time of a particular run. The models also express the high time cost of a page fault; show how much longer copying takes than scanning a word; and show that the cost to scan a word depends on which class of spaces to be scanned it belongs to. In addition, the time-varying characteristics of memory allocation, survival, collector work, and efficiency are presented. Collector efficiency is quantified as words of discovered garbage per unit time (or work) expended by the collector. The variation of collector efficiency over the execution of the measured programs suggests the potential for improved performance by more closely matching garbage collection policy with program characteristics.

Index Terms

Garbage collection, Lisp systems, memory management and usage, performance measurement and analysis, statistical modeling, virtual memory.
1 Introduction

This paper describes measurements and analysis of memory usage and performance in a Lisp system. The work presented here is part of research whose overall objective is to better understand and model memory system behavior and improve memory management efficiency, for object-based, dynamically allocated, automatically garbage collected, virtual memory systems, such as Lisp.

Our primary focus is on garbage collection in a virtual memory environment. Garbage collection is performed for two purposes: to reuse address space, and to increase the spatial locality of objects. Incremental and generational garbage collection have been available on microcoded Lisp machines for some time now [10], and have recently been developed for Lisp implementations on conventional architectures. Generational collectors have been shown to be less disruptive and more efficient than earlier techniques, and have contributed to the attractiveness of Lisp for developing large and complex applications. Nevertheless, with the ever increasing size and demands of such applications, garbage collection in virtual memory remains an important issue. Recent papers have reported on experiences and techniques in optimizing garbage collection and paging performance on existing systems, and on developing efficient implementations on conventional architectures [1, 6, 14, 15, 18, 3, 2, 19].

In this paper, we present an analysis of memory system and garbage collection activity on a Symbolics 36201 Lisp machine. The data consists of regular samples of activity times, event counts, and memory space sizes maintained by the system. Programs measured include the Boyer benchmark and QPE, a large AI program, run with two different input data sets. The analysis is organized into three parts:

1. An overall analysis of processor utilization, page faults, and other page management overhead is given.

2. An analysis of time and page faults in the garbage collector is presented. Empirical models to relate garbage collection time to collector work, page faults, and other overhead are derived using linear regression. The model parameters quantify the duration (time cost) of various operations. The accuracy of the models in predicting overall garbage collection time is tested.

3. We examine the time-varying characteristics of memory allocation, survival, collector work, and efficiency. Collector efficiency is quantified as words of discovered garbage per unit time (or work) expended by the collector. The utility of the cost-benefit metrics in suggesting how or where (in time) to tune the garbage collector is demonstrated.

Section 2 contains background information on garbage collection and its implementation on the measured Lisp system. The data collection method and measured programs are described in Section 3. Sections 4–6 correspond to the three main parts of the analysis outlined above. The conclusions appear in Section 7.

1Symbolics, Symbolics 3600, and Genera are trademarks of Symbolics, Inc.
2 Background

This section provides background information on garbage collection and its implementation on the measured system, a Symbolics Lisp machine. The reader is referred to [10, 11, 12, 16] for more system details.

The Symbolics system implements an incremental copying garbage collection algorithm, generational as well as non-generational garbage collection, approximately depth-first copying, and uses a tagged architecture and special hardware.

Incremental Copying Garbage Collection

The incremental copying technique is based on 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 delays that would result if garbage collection were to be performed without interruption. The Baker algorithm divides the heap into two equal-sized spaces, 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. Copying improves locality by increasing the spatial density of live data.

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 only has to do (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, in order to “pull in” any accessible structures still in oldspace. After both the root set and copyspace have been scanned, 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 consing-driven, in order to ensure that consumption does not outpace production of free space.

**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 likely to result in discovery of oldspace references, the scavenger temporarily suspends its normal linear scan of the root set and copyspace in order 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 [6], 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.

**Generational Garbage Collection**

The system provides the *ephemeral* garbage collector, and the *dynamic* collector, which are generational and non-generational, respectively. 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 run time 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 much 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 high percentage of garbage, and thus little transporting work required. When collecting all generations younger than a given level, the root set need only 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 feasible to keep track of these backward inter-generational references.

In ephemeral collection, the policy for initiating collections is capacity-based: a collection is begun when the first level exceeds its pre-specified 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 only 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

• 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 in order to avoid unnecessarily fetching on-disk pages during scavenging.²

Tagged Architecture and Special Hardware

To allow the above techniques to be implemented with acceptable overhead, the Symbolics 3600 relies on tagged architecture and special hardware. The processor provides for hardware detection of

• oldspace object references during memory reads (to know when to trap to transporter microcode); and
• ephemeral object references during memory writes (to know when to update the GCPT).

This hardware “barrier” includes memory 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 software. Much of the recent effort in implementing garbage collection on conventional architectures has been on minimizing this degradation without requiring extra hardware.

²In other generational collection schemes, the entity serving the function of the GCPT and ESRT has gone by such names as entry vector, remembered set, and indirection cells.
3 Measurement Methodology

This section describes the data collected and the programs used to obtain the results presented in succeeding sections. The issue of interference and steps taken to minimize it are considered.

Data

Our study is based on periodically sampling a large number (about 80) of software-accessible meters. These meters track memory management, garbage collection, and disk access activity. A meter may be an activity timer, such as the cumulative number of milliseconds spent servicing page faults; a resource counter, such as the number of pages of a specified kind; or an event counter, such as the cumulative number of page faults. The meters are system-wide, rather than process-specific, and are updated by the underlying Lisp system.

The data also includes some process-specific meters for certain interesting processes. Finally, the data includes the sizes of various memory spaces relevant during garbage collection. This allows the growth or shrinkage of these spaces over time to be observed.

Workload

Two programs were used to obtain the results presented here:

- the Boyer benchmark [7], a theorem-proving kernel using a rewrite-rule-based simplifier and a dumb tautology-checker; and
- QPE (Qualitative Process Engine), a qualitative simulator by K. Forbus which uses an assumption-based truth maintenance system as a substrate.

Boyer was measured on a 3620 running the Genera 7.0G1 system with 2 megawords of main memory and about 15 million words of paging area. QPE was measured on the same 3620 running the earlier Release 6.G2 system with about 25 million words of paging area. The earlier system release was used because a version of QPE for Genera 7 was not available at the time the data was collected.

Data from six measurement runs is presented, each run being a particular combination of program, input data, and garbage collector mode. The runs range from about nine minutes to over 12 hours. The machine was cold-booted before measuring each program.

Table 1 shows some general information about the runs. Boyer-\text{n} refers to workload consisting of \text{n} consecutive calls of the Boyer program. Note that Boyer is not dependent on any input data. QPE-short refers to the execution of QPE on input data representing a simple problem; QPE-long, a more complex one. Two measurements of QPE-long were made: one of the entire 12 hour execution, sampled every 15 seconds; and another of only the initial 5 hours of execution, sampled every 5 seconds. The motivation for this set of runs, was to be able to make informal observations on:

- small benchmark vs. large program behavior (Boyer, QPE-short vs. QPE);
Table 1: Measured programs, number of samples, and total real time.

<table>
<thead>
<tr>
<th>Program</th>
<th>Gc</th>
<th>Nominal sampling period (seconds)</th>
<th>Samples</th>
<th>Real time (hh:mm:ss)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boyer-24</td>
<td>off</td>
<td>1</td>
<td>449</td>
<td>8:47</td>
</tr>
<tr>
<td>Boyer-15</td>
<td>dynamic</td>
<td>1</td>
<td>1664</td>
<td>31:45</td>
</tr>
<tr>
<td>Boyer-24</td>
<td>ephemeral</td>
<td>1</td>
<td>820</td>
<td>14:58</td>
</tr>
<tr>
<td>QPE-short</td>
<td>off</td>
<td>1</td>
<td>903</td>
<td>16:12</td>
</tr>
<tr>
<td>QPE-short</td>
<td>ephemeral</td>
<td>1</td>
<td>1083</td>
<td>20:19</td>
</tr>
<tr>
<td>QPE-long full exec.</td>
<td>ephemeral</td>
<td>15</td>
<td>2888</td>
<td>12:05:05</td>
</tr>
<tr>
<td>QPE-long init. part</td>
<td>ephemeral</td>
<td>5</td>
<td>3744</td>
<td>5:16:57</td>
</tr>
</tbody>
</table>

- the effect of input data (QPE-short vs. QPE-long); and
- reproducibility\(^5\) and the effect of sampling period (QPE-long full exec. vs. QPE-long init. part).

Software Sampling and Interference Effects

A sampler process, was used to periodically save the values of selected meters. As with any measurement involving software sampling, the instrumentation software needs to be carefully designed to minimize interference.

The real-time overhead due to the sampling is easy to measure. Since the sampler process is not interrupted while saving the meter values to memory, the time elapsed during this saving provides a good estimate of time overhead. Our data shows that saving the meter values typically takes on the order of one or two milliseconds. With sampling periods on the order of seconds, the relative overhead is small. Other kinds of overhead can cause the paging and memory usage behavior to be different from that in the uninstrumented case. We used a number of system features for manual storage management to reduce or eliminate the influence of the sampler process.\(^6\)

\(^5\)The sensitivity of the results to the initial state of virtual memory is probably the strongest factor affecting reproducibility. Measurements were collected on a given program starting from a freshly booted, minimally mutated world, in order to provide a basis for comparison between programs. However, it is conceivable that even "slight" changes to the world—not to mention large changes occurring after the machine is used for some time—could significantly alter the measurements. Short programs such as Boyer would probably be very sensitive, while longer ones might be able to establish a consistent working set and object population pattern after some initial time period and a few garbage collection cycles. This conjecture seems to be supported by the similarity of the results for the two measurements of QPE-long.

\(^6\)For example, the buffer into which the sampled data is saved is pre-allocated and wired down in main memory, i.e., declared non-pageable. The memory area it uses is declared to be static, so that the garbage collector does not consider it. Temporary structures are consed on the control stack, so that they are automatically reclaimed. To minimize scheduling overhead, the sampler process itself is implemented as an especially efficient kind of process which does not need its own stack group [17].
4 Paging and Storage Allocation Activity

In this section, we focus on low-level virtual memory management activity, such as paging and page allocation. For each activity of interest, the total time, total event count, mean fraction of real time, and mean event rate are given. The time behavior for certain runs are shown. A scatter plot is used to visualize the distribution of fraction of time taken by the major activities.

**Totals and Ratios** The upper halves of Tables 2 and 3 show the total real time spent measuring each of the programs and the average fraction of time spent in various page management activities. The lower halves show the total count and average event rate for various page management events.

Memory management time is broken down into time for

- page faults, which is time spent fixing page faults;
- page creation, which is time spent allocating pages; and
- other page management activities, such as prefetching, destroying, wiring (making non-pageable) and unwiring, and flushing (making replaceable).
### Table 3: Total real time and page management work breakdown.

<table>
<thead>
<tr>
<th>Totals and ratios</th>
<th>QPE-short gc off</th>
<th>QPE-short ephemeral</th>
<th>QPE-long full exec. ephemeral</th>
<th>QPE-long init. part ephemeral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total real time (hh:mm:ss)</td>
<td>16:12</td>
<td>20:19</td>
<td>12:05:05</td>
<td>5:16:57</td>
</tr>
<tr>
<td>Percent of total time in</td>
<td>84.68</td>
<td>85.74</td>
<td>93.11</td>
<td>85.85</td>
</tr>
<tr>
<td>runtime</td>
<td>84.68</td>
<td>85.74</td>
<td>93.11</td>
<td>85.85</td>
</tr>
<tr>
<td>page faults</td>
<td>11.76</td>
<td>9.94</td>
<td>6.17</td>
<td>13.03</td>
</tr>
<tr>
<td>page creation</td>
<td>3.22</td>
<td>4.05</td>
<td>0.55</td>
<td>0.79</td>
</tr>
<tr>
<td>sequence breaking</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>page prefetching</td>
<td>0.00</td>
<td>0.21</td>
<td>0.01</td>
<td>0.03</td>
</tr>
<tr>
<td>page destroying</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>(error/unknown)</td>
<td>0.34</td>
<td>0.06</td>
<td>0.16</td>
<td>0.30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page management work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of</td>
</tr>
<tr>
<td>page faults</td>
</tr>
<tr>
<td>page creations</td>
</tr>
<tr>
<td>page prefetches</td>
</tr>
<tr>
<td>discarded prefetched pages</td>
</tr>
<tr>
<td>forced modified page writes</td>
</tr>
<tr>
<td>Average rate (1/second) of</td>
</tr>
<tr>
<td>page faults</td>
</tr>
<tr>
<td>page creations</td>
</tr>
<tr>
<td>page prefetches</td>
</tr>
</tbody>
</table>
Other time is spent in sequence breaking, which is time spent in the scheduler process, deciding which process to run next. All other time is lumped as “run time”. The run time expressed as a fraction of real time can be considered a measure of average “CPU utilization”. Run time includes the time spent executing both user code and system code such as the garbage collector. Note that memory management time is that expended on behalf of both user and system code. The measures of time here reflect the effect of all active processes. However, system activity is expected to be most heavily influenced by the process running the program being measured and by garbage collection.

Comparing the results for Boyer in Table 2, we observe heavy paging under dynamic garbage collection—64% of real time on average, 5,231 total faults, 2.7 faults/second—and consequently low CPU utilization. This is not surprising. Since oldspace consists of all memory containing non-static objects, a large amount of paging is expected due to having to scavenge a very large copy space and to transport a large number of nongarbage objects. A non-negligible amount of prefetching can also be observed under dynamic garbage collection (5.0% of real time, 41,117 total pages prefetched, 22 pages/second), as a result of the garbage collector’s prefetching policy.

A more interesting question is, how does the performance under ephemeral garbage collection compare with that for no collection? The comparison can be made for Boyer and QPE-short. For Boyer, paging under ephemeral collection is considerably less, and CPU utilization higher, than that with the garbage collector turned off. With no collection, paging took 14% of real time on average and the utilization was 53%. A total of 441 page faults occurred and the average fault rate was 0.84 faults/second. Under ephemeral collection, paging consumed only 2.3% of real time and the utilization increased to 86%. There were only 148 faults and the rate was 0.16 faults/second.

The reduced amount of faults is probably due to increased locality arising from the copying and compaction of nongarbage objects. However, even though there were fewer faults, the program ran 70% longer with ephemeral collection—898 vs. 527 seconds—due to run time in the garbage collector. In Section 5, it will be seen that a total of 452 seconds, or about 50% of total real time, was spent in the garbage collector.

For QPE-short, the trends observed for Boyer can also be seen. There was a slight reduction in paging—from 12% of real time, 1,768 total faults, and 1.8 faults/second with no collection, down to 10% of real time, 1,697 total faults, and 1.4 faults/second under ephemeral collection. Again, total running time increased due to run time in the garbage collector—from 972 up to 1219 seconds, or a 25% increase. However, the reduction in paging and increase in total running time are less dramatic than for Boyer. Garbage collection consumed a total of 220 seconds, or 18% of total real time.

The reduced paging indicates the effectiveness of the ephemeral collection algorithm in a virtual memory environment and especially for the Boyer program. This reduction is achieved by scavenging a smaller root set and copy space, and transporting nongarbage objects from a smaller oldspace than the traditional dynamic garbage collector. The high percentage of the time in garbage collection suggests that improving the garbage collector can still lead to increased performance.

7No sequence breaking or page destroying times are shown for QPE because the associated meters were not available under Release 6. Page destroying, wiring and unwiring, and flushing times are not shown because they were negligible or zero.
Turning now to page creation characteristics, note that more pages are created in Boyer under ephemeral collection than with no garbage collection—53,152 vs. 42,482 pages. Since Boyer is not a nondeterministic program, we expect it to allocate the same number and sizes of objects regardless of the collection type. We must assume that the additional creation of about 11,000 pages is due to the garbage collector. The bulk of the additional creation probably occurs when copy space pages have to be allocated to contain transported objects.

However, even though almost 11,000 more pages are created in Boyer under ephemeral collection than with no collection, less time is spent on page creation—5.8% vs. 27% of real time on average. The reason for the high page creation time in Boyer with no collection is unclear, and this oddity is not observed in the case of QPE-short. For QPE-short, going from no collection to ephemeral collection results in an increase in the total number of pages created—from 10,336 to 21,920 pages due to garbage collector allocation; but the page creation time also increases—from 3.2% to 4% of the time.

In any case, the small amount of time taken by page creation overhead for all runs with ephemeral garbage collection suggests that improving the operations involved in page creation will result in, at best, only a small increase in performance.

In summary, for both Boyer and QPE-short, running with ephemeral garbage collection reduced paging and increased CPU utilization, but also increased total execution time over the case with no collection. The longer execution time is due mostly to run time in the garbage collector. Although ephemeral algorithm is indeed much less intrusive than the original dynamic algorithm, its run time overhead was significant for two of the runs—50% and 18% of real time in Boyer and QPE-short, respectively. Page creation time overhead was found to be small for all runs with ephemeral garbage collection.

**Time Behavior**  A time series plot is useful for showing when the activities or events of interest occur; whether their intensity or frequency of occurrence is uniform throughout, or localized to certain portions of, the measured period; and whether there is any pattern or regularity.

Figures 1–4 show the time series plots for utilization, paging, and page creation activity for the some of the runs.

The high paging overhead noted earlier for Boyer under dynamic garbage collection is evident in Figure 2. The reduced paging for Boyer under ephemeral collection versus no collection can also be observed by comparing Figures 1 and 3. All plots for Boyer show that page management overhead is fairly uniform throughout the measured period. The uniformity is expected, since the workload consists of data-independent repetitions of the same program.

While Boyer incurs system overhead uniformly, QPE-short and QPE-long do not. In particular, the time series for the full execution of QPE-long (Figure 4) reveal that most of the paging occurs during the first 3:40 hours of execution. In optimizing the paging performance of this particular program, one would concentrate on this initial period, since 92% of the total 48,677 page faults occur during this time.

Over the full 12:05 hours of execution of QPE-long, Table 3 shows that paging takes 6.2% of real time on average (with 13% standard deviation) and the mean rate is 1.1 faults/second (3.5% std. dev.). When calculated over the first 3:40 hours only, the corresponding figures increase to 17% of real time (19% std. dev.) and 3.4 faults/second (5.4% std. dev.). Clearly, statistics measured over the entire execution lifetime may not be representative if there are
Figure 1: Run time, page fault, and page creation time expressed as a fraction of real time; and rate (1/second) of page fetches and creations for Boyer-24, gc off.

Figure 2: Run time, page fault, and page creation time expressed as a fraction of real time; and rate (1/second) of page fetches and creations for Boyer-15, dynamic gc.
Figure 3: Run time, page fault, and page creation time expressed as a fraction of real time; and rate (1/second) of page fetches and creations for Boyer-24, ephemeral gc.

Figure 4: Run time, page fault, and page creation time expressed as a fraction of real time; and rate (1/second) of page fetches and creations for QPE-long (full exec.), ephemeral gc.
large variations in the intensity of the activity over time.8

These results are a reminder that complex and long-running applications may exhibit quite different paging characteristics over time; and they identify periods of frequent paging over which paging reduction techniques (e.g., choosing alternative data representations; adjusting policies for page replacement, prefetching, and garbage collection; and object reordering [1]) could be profitably applied.

Scatter Plots The sum of run time, paging, and page creation accounts for 91.2–99.9% of real time. We can visualize the dynamic relationship between them by representing each sample as a point in three-dimensional space, one axis for each activity.

Figure 5 shows the resulting scatter plots. They basically depict how much the two major types of low-level storage management overhead (paging and allocation) detract from run time. The lower either type of overhead, the higher the system can be up along the run time axis. As expected, most points lie on or slightly beneath the triangular surface whose vertices are at unity on each axis. Also, the dominance of paging in Boyer under dynamic garbage collection is evident.

When running Boyer without collection, the system spends more than a quarter of the time (27%) in page creation (Table 2). This characteristic can be seen in Figure 5 as a concentration of points along the page-creation-run-time plane. In contrast, for QPE-long, paging time is the more likely overhead, as manifested by the concentration of points along the paging-run-time plane.

---
8Note that while the standard deviation provides a useful and convenient measure of the degree of fluctuation in a quantity, it must be specified together with the sampling period chosen. If the sampling period were made very large (or, alternatively, if smoothing transformations are applied to the raw data), high frequency variation would effectively be “averaged out” resulting in a lower standard deviation.
A less obvious observation is that very few points are clustered near the bottom edge of the imaginary triangular surface, i.e., along the paging-page-creation plane. This result indicates that, for the programs measured and sampling periods chosen (1, 5, and 15 seconds), paging and page creation time overhead tend not to appear in intensive amounts simultaneously. During a sampling period, there may be considerable faulting or considerable page allocation activity, but not both.
5 Garbage Collector Analysis

Ephemeral garbage collection is much more efficient and less intrusive than dynamic collection. However, running with the ephemeral collector can still take significantly longer than without it (see Table 1 or Figure 8). For the programs for which a comparison can be made, Boyer and QPE-short took 70% and 25% longer, respectively, with garbage collection accounting for 50% and 18% of real time. In this section, a performance analysis of the constituent tasks involved in garbage collection is presented. Regression models for garbage collection time in terms of work done, page faults, and other overhead are developed and evaluated.

Figure 6 shows the decomposition of garbage collection tasks in the form of a state chart. At the highest level, storage reclamation involves scavenging and transporting. Scavenging may be consing-induced, i.e., performed to keep up with mutator allocation, or done while the machine is idle. Two memory spaces need to be scavenged: the root set and copy space. Because of the approximately depth-first copying technique explained in Section 2, copy space scavenging can be decomposed into “last” and “normal” page components. For ephemeral garbage collection, root set scavenging can be further divided into scavenging of in-memory pages (flagged by the GCPT) and on-disk pages (flagged by the ESRT).

Transporting can be triggered by scavenging or normal computation. A call to the transporter occurs when an oldspace object reference is read, either by the scavenger while scanning...
memory, or by the mutator. If called, the transporter will, if not previously copied, copy the oldspace object and install a forwarding pointer; and will replace the oldspace reference with a reference to the copyspace version of the object.

Table 4 shows the breakdown of time, page faults, and work done. The entries are total values measured over the entire program execution. For all runs, scavenging was almost entirely consing- rather than idle-time-induced, since none of the programs involved any pauses for user input.

Total garbage collection time varied from 3.5% of total real time for QPE-long (full exec.) to 74% for Boyer under dynamic collection. For all programs, most of garbage collection time (93–99%) was spent in the scavenger. The programs differed in how scavenging time was distributed among the various types of pages to be scavenged, and how scavenging time was distributed between scanning memory and transporting objects.

Under ephemeral collection, for Boyer and QPE-short, little time (0.3%, 0.2% of scavenging time) was spent scavenging ESRT pages, suggesting that most of the pages into which ephemeral object references were stored did not get purged from main memory. In contrast, for QPE-long, about one fourth of scavenging time was due to scavenging disk-resident pages.

As expected, transporting one word is much more time-consuming than scanning one word. For example, for QPE-short, the words scanned to words transported ratio was 5.8 to 1, but the scavenger took almost as much time transporting as scanning. Assume a simple linear relationship between scanning (transporting) time and words scanned (transported). Then, for Boyer with dynamic collection, transporting one word could be considered time-equivalent to scanning 1.2 words. For the four ephemeral runs, the equivalent scanning work would be 11, 5.8, 8.9, and 4.3 words. This result could be considered in defining a more realistic single measure of collector work for use in efficiency metrics such as discussed in Section 6. In an algorithm which pegs the rate of collector work to the rate of consing, defining work with greater weight on transporting could make the variation in time taken by the scavenger, on every call to do a specified amount of work, more predictable.

The time in mutator-induced transporting indirectly reflects the amount of oldspace objects discovered to be nongarbage by the mutator. This time is 1.0% of total garbage collection time for Boyer, and from 6.5–6.8% for QPE. Courts [6] has proposed the heuristic that oldspace objects accessed by the mutator are likely to be active, i.e., in the working set, in contrast to those accessed by the scavenger, and that such objects occupy only a small fraction of total memory. After flipping all of dynamic and static space into oldspace, and running interactive workload with the scavenger inhibited, he found that only 13% of oldspace had been referenced. In our measured system, the number of oldspace objects touched by the mutator is only a lower bound on the total number of active objects—since the scavenger was not inhibited and could discover active objects before the mutator does. Nevertheless, the relatively small amount of mutator-induced transporting in our CPU-bound programs lends support to the view that only a small amount of oldspace is being actively referenced.

Regression Models It seems reasonable to assume a linear relationship between garbage collection time and the amount of collector work performed, page faults, and other system

---

10This isn’t quite true in general, as will be shown later.
<table>
<thead>
<tr>
<th>Totals and ratios</th>
<th>Boyer-15 dynamic</th>
<th>Boyer-15 ephemeral</th>
<th>Boyer-24 ephemeral</th>
<th>QPE-short ephemeral</th>
<th>QPE-long full exec. ephemeral</th>
<th>QPE-long init. part ephemeral</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Real time</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total real time (hh:mm:ss)</td>
<td>31:45</td>
<td>14:58</td>
<td>20:19</td>
<td>12:05:05</td>
<td>5:16:57</td>
<td></td>
</tr>
<tr>
<td>Percent of total time in garbage collector</td>
<td>74.41</td>
<td>50.38</td>
<td>18.08</td>
<td>3.48</td>
<td>6.95</td>
<td></td>
</tr>
<tr>
<td>Percent of gc time in scavenger</td>
<td>99.91</td>
<td>98.96</td>
<td>93.51</td>
<td>93.38</td>
<td>93.16</td>
<td></td>
</tr>
<tr>
<td>Percent of gc time in mutator-induced transporting</td>
<td>0.09</td>
<td>1.04</td>
<td>6.49</td>
<td>6.62</td>
<td>6.84</td>
<td></td>
</tr>
<tr>
<td><strong>Page faults</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total page faults</td>
<td>5,231</td>
<td>148</td>
<td>1,697</td>
<td>48,677</td>
<td>49,561</td>
<td></td>
</tr>
<tr>
<td>Percent of total faults in garbage collector</td>
<td>84.32</td>
<td>6.76</td>
<td>1.47</td>
<td>24.90</td>
<td>27.59</td>
<td></td>
</tr>
<tr>
<td>Percent of gc faults in scavenger</td>
<td>99.86</td>
<td>60.00</td>
<td>100.00</td>
<td>98.89</td>
<td>99.69</td>
<td></td>
</tr>
<tr>
<td>Percent of gc faults in mutator-induced transporting</td>
<td>0.14</td>
<td>40.00</td>
<td>0.00</td>
<td>1.11</td>
<td>0.31</td>
<td></td>
</tr>
<tr>
<td><strong>Work</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total words scanned</td>
<td>15,293,327</td>
<td>50,723,692</td>
<td>11,297,323</td>
<td>142,813,693</td>
<td>51,871,841</td>
<td></td>
</tr>
<tr>
<td>Total words transported</td>
<td>1,552,663</td>
<td>2,701,806</td>
<td>1,929,255</td>
<td>6,818,632</td>
<td>6,759,927</td>
<td></td>
</tr>
<tr>
<td>Total oldspace reclaimed</td>
<td>3,526,906</td>
<td>13,123,890</td>
<td>2,759,212</td>
<td>39,428,316</td>
<td>16,358,752</td>
<td></td>
</tr>
<tr>
<td>Percent of words scanned in scavenging</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Percent of words scanned in GCPT pages</td>
<td>69.70</td>
<td>73.06</td>
<td>72.84</td>
<td>64.17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Percent of words scanned in ESRT pages</td>
<td>6.46</td>
<td>0.46</td>
<td>0.24</td>
<td>17.44</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Percent of words scanned in last page</td>
<td>6.06</td>
<td>6.75</td>
<td>9.80</td>
<td>5.30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Percent of words scanned in normal page</td>
<td>93.94</td>
<td>23.10</td>
<td>16.90</td>
<td>4.79</td>
<td>13.09</td>
<td></td>
</tr>
</tbody>
</table>
software overhead. To evaluate this assumption, we consider various regression models for collection time.

A linear regression model expresses a response (or "dependent") variable as a linear combination of "independent" variables, plus an error term. The error term is assumed to be a normally distributed random variable with mean 0 and variance $\sigma^2$. For example, a model for a response $Y$ in terms of two variables $X_1$ and $X_2$ could have the form:

$$Y = \beta_0 + X_1\beta_1 + X_2\beta_2 + \epsilon$$

Given a set of observations for $Y$, $X_1$, and $X_2$, a least-squares analysis is typically used to compute estimates for the unknown constant parameters $\beta_i$ and error variance $\sigma^2$. The goodness of fit of the model is measured by $R^2$, which is the fraction of the variation in the response variable explained or accounted for by the model without the error term. $R^2 = 0$ implies complete lack of fit and $R^2 = 1$ implies perfect fit.

Our model is developed as follows. Within a given interval of time, such as a sampling period, the time in garbage collection can be expressed as the sum of scanning and transporting time (see Figure 6):

$$Y_{GC} = \text{expected scanning time} + \text{expected transporting time} + \epsilon$$

Scanning time is the time spent in the scavenger, excluding the time spent in scavenger-induced transporting. As for any application, scanning time consists of run time, paging, page creation, and other page management overhead as discussed in Section 4. Run time can be modeled as the sum of

- time in reading memory sequentially (proportional to the number of words scanned);
- some overhead in the scavenger (proportional to the number of calls to the scavenger);
- some overhead in calling the transporter from the scavenger (proportional to the number of scavenger-induced transporter calls).

Paging time can be assumed proportional to the number of scavenger page faults, excluding faults during scavenger-induced transporting. Finally, assume that no page creation time is incurred when scanning memory, and ignore other page management overhead, which was found to be negligible. The expected scanning time is therefore assumed to be of the form:

$$\text{expected scanning time} = f(X_{ScWords}, X_{ScCalls}, X_{STcalls}, X_{ScPagef})$$

where $f$ denotes linear combination:

$$f(X_1, X_2, X_3, \ldots) = X_1\beta_1 + X_2\beta_2 + X_3\beta_3 + \cdots$$

A more detailed model for expected scanning time is possible by including separate terms for the words scanned during GCPT, ESRT, last page, and normal page scavenging; and for the number of calls to the corresponding scavenging routines:

$$\text{expected scanning time} = f(X_{GCPTwords}, X_{ESRTwords}, X_{LastWords}, X_{NormWords},$$

$$X_{GCPTcalls}, X_{ESRTcalls}, X_{LastCalls}, X_{NormCalls},$$

$$X_{STcalls}, X_{ScPagef})$$

Similar arguments lead to a model for expected transporting time:
Table 5: Independent variables.

<table>
<thead>
<tr>
<th>Model for scanning time (simple)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Words scanned</td>
</tr>
<tr>
<td>Scavenger calls</td>
</tr>
<tr>
<td>Scavenger-induced transporter calls</td>
</tr>
<tr>
<td>Page faults while scanning</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model for scanning time (detailed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCPT words scanned</td>
</tr>
<tr>
<td>ESRT words scanned</td>
</tr>
<tr>
<td>Last page words scanned</td>
</tr>
<tr>
<td>Normal page words scanned</td>
</tr>
<tr>
<td>GCPT calls</td>
</tr>
<tr>
<td>ESRT calls</td>
</tr>
<tr>
<td>Last page calls</td>
</tr>
<tr>
<td>Normal page calls</td>
</tr>
<tr>
<td>Scavenger-induced transporter calls</td>
</tr>
<tr>
<td>Page faults while scanning</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model for transporting time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Words copied</td>
</tr>
<tr>
<td>Transporter calls</td>
</tr>
<tr>
<td>Page faults while transporting</td>
</tr>
</tbody>
</table>

**expected transporting time** $= f(X_{\text{TrWords}}, X_{\text{TrCalls}}, X_{\text{TrPagef}})$

Table 5 summarizes the independent variables involved. The regression parameters could be interpreted as the average

- seconds per word scanned by the scavenger, or copied by the transporter (e.g., $\beta_{\text{ScWords}}$);

- seconds per call to the scavenger, or to the transporter (e.g., $\beta_{\text{ScCalls}}$); and

- seconds per page fault during scanning, or transporting (e.g., $\beta_{\text{ScPagef}}$).

In the transporter model, note that the term involving the number of words copied accounts for both time spent copying objects and time allocating copyspace. The regression parameter $\beta_{\text{TrWords}}$ has the meaning of average seconds per word copied due to the memory read and write, and to forwarding pointer processing and copyspace allocation amortized over each word transported.

**Regression Results** For each program, we fit both the simple and detailed model to data from each sampling interval, using a statistical analysis program [13]. The data was first normalized with respect to real time; the dependent variable $Y_{\text{GC}}$ was defined to be the fraction of time in garbage collection; the independent variables $X_i$ were defined to be event rates.

Values of $R^2 > 0.9$ were obtained, as shown in Table 6, suggesting that the linear relationships are quite accurate. The detailed models have slightly better fit than the corresponding simple models. We found that both simple and detailed models suffer from multicollinearity,
Table 6: $R^2$ values and estimates of error standard deviation. All models are statistically significant at the level of $\alpha < 0.0001$.

<table>
<thead>
<tr>
<th>Regression model</th>
<th>Boyer-24</th>
<th>QPE-short</th>
<th>QPE-long full exec.</th>
<th>QPE-long init. part ephemeral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple $R^2$</td>
<td>0.9970</td>
<td>0.9713</td>
<td>0.9330</td>
<td>0.9027</td>
</tr>
<tr>
<td>Simple $\hat{\sigma}$</td>
<td>0.0334</td>
<td>0.0494</td>
<td>0.0238</td>
<td>0.0491</td>
</tr>
<tr>
<td>Detailed $R^2$</td>
<td>0.9994</td>
<td>0.9736</td>
<td>0.9455</td>
<td>0.9141</td>
</tr>
<tr>
<td>Detailed $\hat{\sigma}$</td>
<td>0.0146</td>
<td>0.0474</td>
<td>0.0215</td>
<td>0.0462</td>
</tr>
</tbody>
</table>

i.e., certain subsets of variables were significantly correlated, thus contributing redundant information [9]. While multicollinearity is not a problem if the goal is to make good predictions, it caused some parameter estimates to be negative, and therefore physically uninterpretable. The variables that tended to be correlated were the “words scanned” and “times called” variables. Apparently, more or less the same number of words are scanned on every call to the function to scavenge ESRT pages, for example.11

To reduce multicollinearity and observe whether the parameter estimates are reasonable, we discarded the “times called” variables, allowing the effect of per-call overhead to be absorbed by the “words” variables. Table 7 shows the parameter estimates for the reduced models.12

The regression parameter estimates in the reduced models reveal differences in the modeled time costs of scanning a word depending upon where the word is. Since the simple model lumps all words scanned into one term ($X_{\text{ScWords}} = X_{\text{GCPTwords}} + X_{\text{ESRTwords}} + X_{\text{LastWords}} + X_{\text{NormWords}}$) it is not able to represent these differences and consequently exhibits poorer fit. The highest cost is seen for words in the last page of copy space. The lowest cost is for root set words in physical memory and for words scanned during the normal linear traversal of copy space. Root set words on disk have intermediate cost. Note that these costs include garbage collector overhead associated with scanning the various areas, amortized over the words scanned.

Transporting a word cost more than scanning a word by a factor of $\frac{\beta_{\text{TrWords}}}{\beta_{\text{ScWords}}}$. This ratio could be used to defining a time-equivalent measure of collector work. If scanning one word defines one unit of work, then transporting one word could be said to perform $\frac{\beta_{\text{TrWords}}}{\beta_{\text{ScWords}}} = 70$, $11$, $10$, and $5.6$ units of work for the four programs run under ephemeral collection, respectively. Note that this scheme ignores the large differences in the per-word time among the various spaces to be scanned and excludes paging time. These ratios are to be compared with $11$, $5.8$, $8.9$, and $4.3$ obtained earlier based on the total scanning and transporting time, both of which include paging time.

The estimates for $\beta_{\text{ScPagef}}$ and $\beta_{\text{TrPagef}}$ show that the time cost of a fault is several orders of magnitude greater than for a unit of collector work. Interestingly, the average time per page fault is similar across programs (in the 19–50 ms range) except for QPE-short, which exhibits higher values (143–159 ms). It is not clear whether this unusual result is simply

11Strong correlations between words scanned and calls to the appropriate function were noted for GCPT, ESRT, and last page scavenging, but not for normal page scavenging.
12An estimate for $\beta_{\text{ScPagef}}$ for Boyer-24 is missing because there were no page faults while scanning. All faults that were counted in the garbage collector occurred while transporting. See Table 4. All other missing parameters are not statistically significant at the $\alpha = 0.025$ level.
Table 7: Parameter estimates in reduced models for garbage collection time. All models are statistically significant at the level of $\alpha < 0.0001$.

<table>
<thead>
<tr>
<th>Regression parameter estimate</th>
<th>Boyer-24</th>
<th>QPE-short</th>
<th>QPE-long</th>
<th>QPE-long</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>units</td>
<td>ephemeral</td>
<td>ephemeral</td>
<td>ephemeral</td>
</tr>
<tr>
<td>Simple model</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\beta_{\text{ScWords}}$</td>
<td>$\mu$sec/word</td>
<td>1.833</td>
<td>5.250</td>
<td>4.626</td>
</tr>
<tr>
<td>$\beta_{\text{TrWords}}$</td>
<td>$\mu$sec/word</td>
<td>128.5</td>
<td>56.34</td>
<td>46.76</td>
</tr>
<tr>
<td>$\beta_{\text{ScPagef}}$</td>
<td>$\mu$sec/fault</td>
<td>—</td>
<td>158.7</td>
<td>50.48</td>
</tr>
<tr>
<td>$\beta_{\text{TrPagef}}$</td>
<td>$\mu$sec/fault</td>
<td>—</td>
<td>—</td>
<td>22.35</td>
</tr>
<tr>
<td>$R^2$</td>
<td></td>
<td>0.9864</td>
<td>0.6905</td>
<td>0.8183</td>
</tr>
<tr>
<td>$\hat{\sigma}$</td>
<td></td>
<td>0.0716</td>
<td>0.1618</td>
<td>0.0392</td>
</tr>
<tr>
<td>Detailed model</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\beta_{\text{GCPTwords}}$</td>
<td>$\mu$sec/word</td>
<td>1.729</td>
<td>3.117</td>
<td>2.589</td>
</tr>
<tr>
<td>$\beta_{\text{ESRTwords}}$</td>
<td>$\mu$sec/word</td>
<td>11.47</td>
<td>18.76</td>
<td>9.690</td>
</tr>
<tr>
<td>$\beta_{\text{LastWords}}$</td>
<td>$\mu$sec/word</td>
<td>45.67</td>
<td>108.4</td>
<td>118.6</td>
</tr>
<tr>
<td>$\beta_{\text{NormWords}}$</td>
<td>$\mu$sec/word</td>
<td>2.008</td>
<td>2.864</td>
<td>—</td>
</tr>
<tr>
<td>$\beta_{\text{TrWords}}$</td>
<td>$\mu$sec/word</td>
<td>73.64</td>
<td>23.27</td>
<td>26.82</td>
</tr>
<tr>
<td>$\beta_{\text{ScPagef}}$</td>
<td>$\mu$sec/fault</td>
<td>—</td>
<td>142.8</td>
<td>41.50</td>
</tr>
<tr>
<td>$\beta_{\text{TrPagef}}$</td>
<td>$\mu$sec/fault</td>
<td>—</td>
<td>—</td>
<td>24.38</td>
</tr>
<tr>
<td>$R^2$</td>
<td></td>
<td>0.9907</td>
<td>0.9223</td>
<td>0.8842</td>
</tr>
<tr>
<td>$\hat{\sigma}$</td>
<td></td>
<td>0.0592</td>
<td>0.0812</td>
<td>0.0313</td>
</tr>
</tbody>
</table>
Table 8: Predicted garbage collection times and 95% confidence intervals.

<table>
<thead>
<tr>
<th>Run</th>
<th>Actual gc time</th>
<th>Gc time predicted by detailed model developed from</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Boyer-24 QP &lt;sup&gt;e&lt;/sup&gt;-long QP &lt;sup&gt;e&lt;/sup&gt;-long QP &lt;sup&gt;e&lt;/sup&gt;-long</td>
</tr>
<tr>
<td>Boyer-24</td>
<td>.5038</td>
<td>.5035 ±.029 .5690 ±.095 .6218 ±.066 .6529 ±.105</td>
</tr>
<tr>
<td>QPE-short</td>
<td>.1808</td>
<td>.1745 ±.093 .1901 ±.042 .1802 ±.090</td>
</tr>
<tr>
<td>QPE-long full exec.</td>
<td>.0348</td>
<td>.0458 ±.094 .0355 ±.042 .0370 ±.090</td>
</tr>
<tr>
<td>QPE-long init. part</td>
<td>.0695</td>
<td>.0959 ±.094 .0679 ±.042 .0677 ±.090</td>
</tr>
</tbody>
</table>

due to the shortness of the measurement, or whether it indicates some undesirable interaction between program and system behavior, such as high disk latencies occurring as a consequence of pathologic placement and referencing of disk blocks.

To test the accuracy of the models, especially across programs, we applied them to the mean rate of scanning words, transporting words, page faults, etc., and compared the predicted fraction of time in garbage collection with the actual mean fraction of time in garbage collection. It was anticipated that the model developed from QPE-short, for example, would make a good prediction for the overall mean collection time in QPE-short, but how well would it predict Boyer's collection time?

In all but two cases, the "self-" as well as "cross-program" predictions turned out to be correct, where a prediction is considered correct if the actual value falls within the 95% confidence interval about the predicted value. Table 8 shows results for the full (non-reduced), detailed model only. This model gives the tightest confidence intervals, but the correctness results are identical for the other variations of models discussed. The exceptions were in using the two QPE-long models to predict Boyer's mean garbage collection time. The models overestimate the time. This could be due to differences in program characteristics as well as in the underlying system software. Measurements of other programs are underway to determine the effect of program characteristics on the model.

It has occasionally been assumed in the literature, usually implicitly, that collection time is linearly dependent on the work to be performed, i.e., the number of words to be scanned and transported. To test this assumption, we evaluated the following "work only" models:

\[ Y_{GC} = f(X_{ScWords}, X_{TrWords}) + \varepsilon \]  
(1)

\[ Y_{GC} = f(X_{GCPTwords}, X_{ESRTwords}, X_{LastWords}, X_{NormWords}, X_{TrWords}) + \varepsilon \]  
(2)

For the QPE runs, the simple model (1) yielded values of \( R^2 < 0.52 \), while the detailed model (2) had \( R^2 < 0.68 \). For Boyer, for which there were no faults while scanning memory, and very few faults while transporting, both models produced a high \( R^2 > 0.98 \). These results suggest that the total number of words to be scanned, and number of words to be copied, are

\[ \text{The Boyer-24 model was not tested on any of the QPE means, since this model does not account for page faults. The confidence intervals given in Table 8 are those for predicting an individual value of } Y_{GC} \text{ given a set of } X \text{ values, rather than the expected value of } Y_{GC} \text{ [9]. Confidence intervals for the expected values are much smaller, but inappropriate in the context of our test.} \]
not very good linear predictors of scanning and transporting time respectively, except in the case where there is little paging, such as for small programs.

To summarize, regression analysis showed that collection time is linearly related to collector work, page faults, and run time overhead in the collector routines. The models correctly predicted the overall mean collection times for all ephemeral runs, except for Boyer, whose time was overestimated by the models derived from QPE-long data. Regression analysis also provided estimates of, and comparisons between, the average time costs of various operations in the model. The set of estimates arise from, and may be viewed as a description of, the joint behavior of program and system. Linear models which ignore page fault time and express garbage collection time in terms of collector work alone were found to be inaccurate when there is a significant amount of paging.
# Memory Usage

In this section, the time-varying characteristics of memory usage are considered. The behavior under garbage collection, in particular, the survival of objects, the amount of garbage reclaimed and work done by the collector, is measured. Cost-benefit measures for garbage collection are defined and used to evaluate the efficiency of collection cycles.

Figure 7 shows a comparison between memory usage under dynamic and ephemeral garbage collection. Note that these are idealized, composite plots of the various memory spaces over time, rather than actual measurements. In reality, the various memory spaces are not allocated contiguously as these pictures might suggest.

Initially, the only spaces that exist are newspace and static space (memory containing static objects). A dynamic collection (Figure 7a) begins either by explicit user action, or by default at some "safe" time determined by the system as explained earlier. Newspace is flipped, i.e., turned into oldspace. Scavenging of the root set and copy space begins. While scavenging is in progress, copy space grows to accommodate oldspace objects discovered to be nongarbage. Newly allocated newspace grows to accommodate objects created by the mutator. Some amount of static objects may also be created. When scavenging is complete, the collection ends, and oldspace can be reclaimed. Another cycle can begin immediately or at some later time. In ephemeral garbage collection (Figure 7b), only a small portion of non-static space is garbage collected. In particular, only those ephemeral levels which have exceeded their capacities are flipped. With no garbage collection, the idealized picture is simplified since only newspace exists.

The actual variation in the memory spaces for the programs measured is shown in Figure 8. These plots illustrate several things, which we now discuss, specifically, the differences between dynamic, ephemeral, and no collection; the rate of allocation with no collection; and characteristics of a collection cycle.

### Dynamic vs. Ephemeral vs. No Collection

In Section 4, we compared CPU utilization and page management overhead in Boyer under dynamic, ephemeral, and no collection. Figure 8 compares the memory usage characteristics. Boyer with no collection is the fastest, but fills up virtual memory. With ephemeral collection, the program takes longer, and many collection cycles occur. With dynamic collection, the program takes the longest, and there is only a single cycle.

The memory usage plots for Boyer are probably typical of workload that happens to fit the ephemeral assumptions well. These assumptions are that newly created objects are likely to become garbage, so that time isn’t wasted performing needless collection; that they become garbage quickly, so that they can be caught before surviving all the ephemeral levels; and that there are relatively few ephemeral references to keep track of, so that the ESRT does not become very big.

### Allocation Behavior

To observe the allocation behavior of a program without the influence of garbage collection, consider the runs of Boyer and QPE-short with no collection. Figure 8 shows that Boyer allocates at a much higher rate and more uniformly than QPE-short. For Boyer, the mean rate is 22,800 words/second with a standard deviation of 6,560. For QPE-short, it is 2,690 words/second with a standard deviation of 4,120. A fairly constant allocation rate probably reflects the (lack of) "complexity" or "diversity" in the workload. This
Figure 7: Idealized composite plot of sizes of various memory spaces under garbage collection.
Figure 8: Measured composite variation in sizes of various memory spaces under garbage collection. See Figure 7 for legend.
is certainly the case with Boyer since the workload consisted of repeated execution of the same program.

Can future memory usage behavior, such as allocation of memory for new objects, be predicted from past behavior? The observation that the rate of allocation is relatively stable—certainly this is the case for Boyer but also for QPE—points toward the possibility of predicting future allocation from past allocation. An interesting approach would be to investigate the predictive ability of time-series-based empirical models based on memory usage measurements like the space sizes considered here, but including other variables for representational adequacy. Such a prediction could be useful as part of an overall scheme, also based on empirical modeling, for adaptive tuning of garbage collection.

Collection Cycle Characteristics A number of characteristics of each garbage collection cycle—and of the application—can be observed from the composite plots in Figure 8. These include oldspace size, the growth of copyspace as nongarbage is discovered, the fraction of nongarbage in oldspace, and the duration of a cycle.

These characteristics are shown explicitly, for two of the runs, in Figures 9 and 10, which show several aspects: Consing (or object creation) activity is represented by the plot of the size of newspace. The nongarbage discovery process is represented by the plots of oldspace, copyspace, and for each collection, the surviving fraction of oldspace and the amount of garbage actually reclaimed. The cost of garbage collection is represented by the plots of scavenger work and garbage collection time. Scavenger work is the sum of work done in scanning the root set, and in scanning and transporting objects found to be nongarbage. We adopt Moon's definition [10] that one scavenger work unit equals one word scanned or one word transported, although this definition does not account for the fact that transporting one word is really more "work" than scavenging one word (see Section 5). Finally, two plots of collector "inefficiency" are shown, explained as follows.

To measure efficiency, the following cost-benefit metrics for each collection cycle are defined and plotted in Figures 9 and 10:

\[ C_{B_{\text{work}}} = \frac{\text{scavenger work done}}{\text{word of garbage reclaimed}} \]
\[ C_{B_{\text{time}}} = \frac{\text{gc time}}{\text{word of garbage reclaimed}} \]

Both metrics quantify benefit as the number of words of garbage discovered and reclaimed by the collection, but differ in the cost measure. \( C_{B_{\text{work}}} \) quantifies cost as the amount of scavenger work required. \( C_{B_{\text{time}}} \) quantifies cost as the time consumed by the garbage collector—a real, "bottom line" cost, since it includes the effect of system overhead such as page faults. Note that only the portion of oldspace that is not transported is considered garbage, not all of oldspace.

For Boyer, we see that collection cycles generally alternate between high and low cost-benefit values, and between collection of the first ephemeral level only and both the first and second levels. Note that the cycles in which the second ephemeral level was flipped can be identified from the plot of oldspace size, considering that garbage collection used the default configuration of two ephemeral levels with capacities of 200,000 and 100,000 words, respectively. Most of the high efficiency cycles are those for which only a small fraction of oldspace survived and in

---

\[ ^{14} \text{New static objects are not included, but their exclusion makes little difference since no significant amount of static object creation was observed in any of the measured programs.} \]
Figure 9: Consing, nongarbage discovery, scavenging work, collection time, and cost-benefit measures for each garbage collection cycle in Boyer-24, ephemeral gc.
Figure 10: Consing, nongarbage discovery, scavenging work, collection time, and cost-benefit measures for each garbage collection cycle in QPE-long (full execution), ephemeral gc.
which both ephemeral levels were flipped. These results suggest that it might be worthwhile to
increase the capacity of the first level, in order to delay flips and allow more time for objects
in the first level to become garbage.

Closer inspection of Figure 9 shows that two collections occur for each of the 24 invocations
of the Boyer benchmark. Clearly, one of them is not very useful since a particular invocation
does not release its objects until it terminates. Flipping once for every integral number of
invocations is what we really want to do for this (admittedly contrived) workload.

QPE-long was shown earlier to incur most of its page faults during the initial hours of
execution (Figure 4). Figure 10 reveals that its memory usage behavior during this initial
period is also remarkably different from that over the remainder of the program’s execution. In
particular, for the first 20 minutes of execution, oldspace sizes are relatively small, on the order
of 300,000 words, and cycles complete in 1–3 minutes. During the next 3:20 hours, oldspace sizes
increase to over one million words and then decrease. Cycle times rise and fall proportionately,
taking as much as 40 minutes. Over the final eight hours, oldspace sizes remain constant at
200,000 words (indicating that only the first ephemeral level is being flipped), collections take
about 2.3 minutes, with one minute between collections, all oldspace is garbage, and dynamic
space does not grow.

As in Boyer, the cost-benefit curves for QPE-long are correlated to the survival of objects
in oldspace. For example, the cycle of lowest efficiency had the highest fraction of nongarbage
in oldspace, about 90%. The high cost-benefit values during the initial 3–4 hour period also
suggest flipping less frequently during this time, to reduce collector work per word reclaimed
and the amount of objects that become garbage after being promoted to dynamic status [18].
It remains to be seen whether the trade-off in decreased locality is significant. One way to view
this trade-off is to consider the effect of not performing a subset of the ephemeral collection
cycles, taking care, of course, not to exceed available virtual memory. There would be two
competing effects on total execution time: Not having to time-multiplex garbage collection
with mutator computation during the eliminated cycles would decrease execution time. On the
other hand, the possible increase in scavenging and transporting during the retained cycles and
possible decreased locality of objects could increase execution time.

Discussion In summary, we presented plots which provide a picture of memory usage vari-
ation. Cost-benefit metrics were defined to determine the efficiency of garbage collection. For
Boyer, the variation in the values of these measures suggested synchronizing flips with invoca-
tions of the benchmark. For QPE-long, the measures identified the initial phase of execution
as the one over which generation tuning, such as increasing ephemeral level capacities, could
achieve significant gains.

These results lead to the problem of optimizing a generational garbage collector in order
to more closely match an application’s object usage characteristics. In the terminology of
ephemeral garbage collection, the tuning problem, in its most general 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 involves various tradeoffs. One tradeoff due to locality considera-
tions was mentioned above. Another appears in the selection of number of levels. Increasing the
number of levels reduces the rate of creation of tenured garbage thereby further postponing
a time-consuming full garbage collection; but increases the amount of copying work for long-lived objects. The best balance among these constraints depends on program and system characteristics and on our performance objectives. We are currently investigating approaches to modeling these effects, which would be useful in both an off-line performance advisor, and in real-time adaptive control of policy.

The behavior of QPE suggests that large programs go through major, distinct phases over their execution, where a phase is characterized by relatively stable and identifiable allocation behavior, survival rate, etc. Assume that the completion of every collection presents an opportunity to effect control over policy. The existence of phases and the observation that they could persist for a period of time much longer than an ephemeral collection cycle is a strong argument for the possibility of dynamically learning about mutator characteristics at each decision epoch and profitably acting on such information.

Our regression analysis presented empirical evidence for a linear relationship between garbage collection time and collection work (scanning and transporting) and page faults. While such a relationship is intuitively obvious, the numerical results as well as our experiences with using various sets of "independent" variables could be useful in the larger task of developing a model relating mutator and system characteristics to collection time. Such a model is required by an adaptive controller in selecting among the range of possible decisions.
7 Conclusions

This paper has presented analyses performed on sampled memory system activity on a Symbolics 3620 Lisp machine. Some of the major results are as follows:

Garbage collection accounted for up to 50% of real time, and up to 28% of all page faults, for the programs measured under ephemeral collection. More than 93% of garbage collection time was spent in the scavenger, including scavenger-induced transporting, with the remaining fraction in mutator-induced transporting. The small amount of transporting due to mutator references supports the view that only a small amount of oldspace is being actively referenced.

Empirical models to predict garbage collection time for a given amount of scanning and transporting work, and page faults were derived using linear regression. The models show the high time cost of a page fault, relative to that of scanning or transporting one word. Transporting a word cost more than scanning a word by a factor of 5.6–70. The time cost to scan one word differed depending upon where that word was. The highest cost was observed for words in the last page of copy space. The lowest cost was for root set words in physical memory and for words scanned during the normal linear traversal of copy space. Root set words on disk had intermediate cost. These differences are attributed to the differences in time spent for bookkeeping overhead associated with scanning the various areas, amortized over the words scanned.

The accuracy of the models was tested by using them to predict the overall mean collection time of the various runs. The actual mean times were within the 95% confidence intervals determined from the models.

We also examined the time-varying characteristics of memory allocation, survival, collector work, and efficiency. Collector efficiency was quantified as words of discovered garbage per unit time (or work) expended by the collector. The variation of collector efficiency over the execution of the measured programs suggested the potential for improved performance by more closely matching garbage collection policy with program characteristics. In particular, the results suggested synchronization of flips with benchmark calls in the case of the Boyer workload, and identified the portion of the program most suitable for generation optimization in the case of QPE-long.
References


