Paging on an Object-oriented Personal Computer for Smalltalk

Ricki Blau

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720

June 1, 1983

ABSTRACT

A high-performance personal computing environment must avoid perceptible pauses resulting from many page faults within a short period of time. Our performance goals for a paged virtual memory system for the Smalltalk-80™† programming environment are both to decrease the average page fault rate and to minimize the pauses caused by clusters of page faults. We have applied program restructuring techniques to the Smalltalk-80 object memory in order to improve the locality of reference. The analysis in this paper considers the clustering of page faults over time and distinguishes between steady-state behavior and phase transitions. We compare the effectiveness of different restructuring strategies in reducing the amount of main memory needed to obtain desired levels of performance.

This research has been supported in part by the National Science Foundation under Grant MCS80-12900.

A version of this paper entitled "Paging on an Object-oriented Personal Computer" will be presented at the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems in Minneapolis, Minnesota, August 29-31, 1983.

†Smalltalk-80 is a Trademark of Xerox Corporation.

1. Introduction

This paper reports the results of performance studies for the design of a virtual memory system for the Smalltalk-80 system, an integrated, object-oriented programming environment designed for personal computers [Gold83]. In the Smalltalk environment all objects are in a single virtual address space, and no distinction is made between the system and the user's applications. Existing implementations keep all objects resident in main memory. Address space and memory size limitations have restricted the growth of applications. In addition, these limitations have forced users to segregate programs into separate object memories and to reboot the system when switching to a new application. To alleviate this problem, researchers at Xerox PARC are developing LOOM, an object-oriented Smalltalk-80 virtual memory [Kaeh83]. Virtual memory is also available in experimental implementations that run on top of existing paging systems. Among these are systems from Digital Equipment Corporation [Ball83] and UC Berkeley [Unga83a].

Smalltalk objects are much smaller than pages. In the worst case for a paged virtual memory, each page in main memory contains only one useful object and most of the space is wasted. For this reason other designers of object-oriented virtual memories ([Kaeh83], [Stam82], [Snyd79]) have chosen to swap individual objects, rather than pages, between primary and secondary memory. In contrast, we have designed a paging virtual memory that relies on program restructuring to minimize unused space in main memory by clustering related objects.

The Berkeley Smalltalk interpreter [Unga83a] is written in C and makes use of the underlying virtual memory support of the Berkeley UNIX system [Baba81]. Berkeley Smalltalk, or BS, serves as a testbed for the development of SOAR (Smalltalk on a RISC [Patt82]), a microprocessor-based personal computer for the Smalltalk-80 environment [Unga83b] [Blau83]. The long-range goal is to run Smalltalk thirty times faster than BS runs on a VAX 11-750 and at lower cost. SOAR will provide a larger virtual address space than previous implementations of the Smalltalk-80 system, but it does not support a very large virtual space. Instead, we have designed a virtual memory whose size is a small integer multiple of the physical memory size.

The SOAR design is based on the following view of a personal computer. During interactive use, a personal computer must respond quickly and without noticeable pauses. In a Smalltalk implementation, pauses might occur for garbage collection or page fault handling. Since multiprogramming is not common, the existing system is usually idle while waiting for a page or input. In contrast, there are large segments of time when a personal computer is unused. Housekeeping activities performed at these times do not interfere with the user's work. This philosophy leads to two performance goals for the virtual memory. First, it is desirable to lower the average page fault rate, because the effective speed of the system is decreased when it waits for pages to be read from the disk. Second, the system should reduce the number of noticeable pauses due to page fault activity. Pauses result if many page faults occur within a short interval, so it is necessary to consider peaks of paging activity as well as averages.

This paper focuses on measuring the improvement of paging behavior achieved by applying program restructuring techniques to the object memory. We analyze the results of paging simulations to predict the amount of physical memory required to provide a given level of performance under various restructuring schemes. In order to estimate how well the system meets the criterion of avoiding pauses, we examine the distribution of page faults over time. Other researchers have studied the clustering of page faults over time, for example [Hatf71], [Chu76], and [Hagm82]. In this paper, we propose new ways of analyzing and interpreting this data. We emphasize the responsiveness of a

personal computer system, rather than the performance of an individual program in a multiprogramming environment.

Previous work in program restructuring

Program restructuring is a means of improving the paging performance of a program by improving its locality of reference [Ferr74], [Ferr76], [Hatf71]. One divides a program into blocks of code or data and then assigns related blocks to the same virtual page. When the principles of program restructuring are applied to Smalltalk, objects serve naturally as blocks. Because they are small, many related objects can be grouped onto one page. However, it is common for objects to be shared by several parts of the system, and there are no distinct boundaries between separate programs. Thus, a restructuring scheme must consider the entire environment, and not simply an isolated program. For the purposes of restructuring, we view the entire object memory as a single program, and apply the restructuring algorithm to a very large number of blocks. Smalltalk allocates and frees objects dynamically, requiring a policy that is robust not only for different inputs but also over time. Online, dynamic restructuring of the object memory is a difficult problem that we are not yet addressing, but we have proposed algorithms that require little advance preparation and could be applied effectively to a personal computer system on a regular basis.

In the model described by Ferrari [Ferr76], program restructuring considers dynamically observed block reference strings. A *restructuring algorithm* computes the desirability of placing pairs of blocks in the same page based on the temporal proximity of references. A large variety of strategies have been proposed for this phase of the restructuring process. The results of the restructuring algorithm are used in clustering blocks onto virtual pages. Our approach to restructuring Smalltalk object memories has not followed this model for the reasons listed below. First, the object memory contains a large number of objects, currently as many as thirty thousand. The normal clustering algorithms are quadratic in the number of objects, so we have explored less expensive methods. Second, we wish to restructure the entire, integrated object memory. A reference string that exercises a large and representative set of the system's facilities would need to be much longer than any that we could easily trace and analyze. Finally, our goal has been to find algorithms which could be applied regularly to a production system without requiring deliberate action by the user and without slowing down the system with extensive measurement. If monitoring activities were to decrease the system's responsiveness, they would interfere with the user's normal interactive behavior and affect any measurements that were taken.

One study has been concerned with restructuring the Smalltalk object memory [Stam82]. The algorithms developed in that study by Stamos use static information about the relationships among Smalltalk objects to group them onto virtual pages. His paging simulations show that the grouping of objects can reduce the page fault rate and suggest that a paged virtual memory will perform poorly without a satisfactory placement of objects onto virtual pages. However, practical constraints limited the length of the reference traces that Stamos was able to obtain and the results were scaled down to artificially small memory sizes. Measurement facilities in Berkeley Smalltalk have enabled us to extend the previous work by tracing longer interactive sessions. In addition, we have been able to study the behavior of an application package with a larger virtual address space. Our results consider larger memory sizes, in a range closer to the sizes planned for the SOAR system.

2. Smalltalk Background

This section introduces the Smalltalk virtual machine and presents a conceptual model of the memory system.

The virtual machine

The Smalltalk-80 system is specified by a virtual machine and a virtual image of Smalltalk objects [Gold83]. A Smalltalk *method* (i.e., procedure) is compiled into a sequence of one-byte instructions called *bytecodes*, that is interpreted by the virtual machine. The speed of a virtual machine implementation is commonly expressed in bytecodes executed per second. As with the instruction sets of other architectures, the bytecode instructions vary in their complexity. Nevertheless, virtual time can be expressed as satisfactorily in bytecodes as in any other unit of measure that has been proposed [Deut82b]. Currently, the fastest Smalltalk system is a microcoded implementation on the Xerox Dorado computer that interprets about 300,000 bytecodes a second. This paper assumes a system with equivalent speed.

The object memory

In discussing the memory needs of a Smalltalk system, it is convenient to divide the memory into regions that serve different purposes (Figure 1).

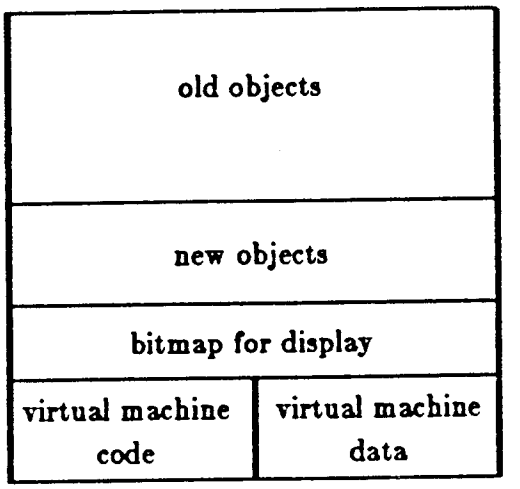

Figure 1. Conceptual Model of Smalltalk Memory

Some of the memory must be devoted to the internal data and code of the virtual machine interpreter. Another region contains the bitmap for the graphics display. Smalltalk objects are located in the two remaining areas of memory.

Old and new objects

Smalltalk objects are dynamically allocated and automatically reclaimed when they are no longer accessible. Table 1 summarizes data obtained by tracing a Smalltalk session. Allocations occur frequently and most objects quickly become inaccessible.

non-context objects allocated dynamically	38,779
length of trace	3,897,289 bytecodes
mean time between allocations	100 bytecodes
median lifetime of objects allocated	202 bytecodes
mean lifetime of objects allocated	11,898 bytecodes
objects that live < 100,000 bytecodes	99.8%

Table 1

Characteristics of dynamically-allocated objects
(excluding activation records)

On the other hand, the initial image contains a core of about 15,000 objects that define the environment and tend to remain unchanged for an indefinite amount of time. We call such long-lived objects *old*, in contrast to recently-allocated *new* objects. User code can add more *old* objects to the image. Both the LOOM virtual memory [Kaeh83] and Berkeley Smalltalk distinguish between *old* and *new* objects. In the SOAR design, new objects are allocated into a separate area of memory. Objects are *matured* into the *old* name space after they have survived a specified amount of time or have met other criteria. We anticipate that this scheme will provide efficient access to new objects, ease storage reclamation, and preserve locality in the old object space [Unga83b].

Everything in the Smalltalk-80 system is viewed as an object, including integers, blocks of compiled code, and activation records. The formal definition of the virtual machine treats all objects uniformly. In practice, large performance gains may be achieved by giving special treatment to activation records [Bade82], [Deut82a]. Consequently, activation records are assumed to be part of the internal data of the virtual machine rather than objects.

The work presented in this paper concentrates on the "old object" memory, that separately-managed portion of the virtual space that contains mature, relatively unchanging objects. References to new objects, procedure activation records, the display bitmap, and the virtual machine's internal memory are ignored. We also ignore references made only for garbage collection, since they depend on the storage reclamation strategy. The scheme used at Berkeley confines most of the garbage collection activity to the "new" area.

In most Smalltalk implementations, an object-oriented pointer, or OOP, accesses an object indirectly through an *object table*. Typically, Smalltalk interpreters record other information about an object, such as its size and class, in either the object table entry or a header within the object itself. The BS interpreter has a paged object table with an entry for each object. Our paging simulations did not consider references made to the object table entries since the design for SOAR does not include an object table. Instead, the SOAR virtual address of an object serves as its OOP. A header at the beginning of each object indicates its size and class, among other information used by the system. Thus, a data object in SOAR will be sixteen bytes longer than its representation in the interpreter instrumented for this project. Appendix A discusses the results of experiments that model the current design for the SOAR object memory.

3. Methodology

The first goal of this study was to measure the improvement in paging performance attainable by restructuring the object memory. The second was to estimate the amount of physical memory needed in order to run a personal computer at a given performance level. Data about page fault rates and the distribution of page faults over time were gathered for these purposes. A mechanism for repeatable experiments allows us to compare these statistics for different restructuring schemes.

The input script facility

Berkeley Smalltalk provides an input script mechanism for performing repeatable experiments. During an interactive session, the BS interpreter creates a script by recording each input event in a file before passing it to the Smalltalk input subsystem. The record describes the input event and indicates the virtual time at which it was received. Because Smalltalk has a graphical user interface, the input events include mouse coordinates and button pushes as well as keystrokes. During an experiment, BS extracts input events from the script file at the indicated virtual times and passes them to the Smalltalk input routines. All of the BS virtual machine code is exercised exactly as it is during normal operation except for the portions that specifically handle interactive input. The Smalltalk applications and system code cannot distinguish between interactive input and input from a script file. Scripts contain enough input and synchronization information that playbacks can reproduce the original sequence of events exactly.

Because BS implements the Smalltalk-80 virtual machine in a higher-level language, it is easy to trace arbitrary internal events and insert code to obtain measurements or perform simulations. In combination, the script facility and event tracing are used to repeat experiments with different parameters or implementation strategies. An alternative would have been to drive simulations from detailed execution traces, such as reference traces. We have found three major advantages to the script approach. First, there is no noticeable decrease in responsiveness when a script is recorded, so the user interacts with the system normally. A second quality is flexibility. The same script may be used for any number of purposes, without having to predict in advance all of the information that must be traced. Finally, input scripts are brief, and very short files can drive long experiments. The potential disadvantage of the approach is the higher cost of re-executing a computation rather than reading a detailed trace. However, the additional execution time is often small compared with the cost of running the measurement or simulation code, and we can always generate an execution trace when necessary.

Currently, our scripts rely on bytecodes as a measure of virtual time. They do not record think time, which is difficult to measure convincingly in our environment. Berkeley Smalltalk is run on a time-shared computer, and the virtual to real time ratio varies greatly. In contrast, our design is for a personal computer that processes Smalltalk at thirty times the speed of BS. Think time is ignored also because the important issue is how well a personal computer performs when the user is waiting for a response.

Paging simulations

A script-driven paging simulator embedded in the BS interpreter is the basis of our measurements of paging behavior. An input script drives the interpreter reproducibly. The interpreter traces references to objects and passes the references directly to the paging simulator. Two categories of references are missed because of an address-caching scheme. First, the interpreter keeps in registers the addresses of a small fixed number of heavily-used objects, such as the current method. Only the first reference to one of these objects is traced within a single activation of a procedure or block. A different block or procedure is entered every ten to fifteen bytecodes on the average, so we miss only references to objects that have recently been used. Second, only one reference to a given object is traced during the execution of a single bytecode. Although we miss references in these two categories, we believe, for the reasons given above, that our reference-tracing mechanism maintains satisfactory accuracy. The majority of the bytecodes executed are instructions which may reference at most one *old* object, but more complex bytecodes may reference several objects. In measurements of three scripts, the average number of traced references to old objects ranged from .58 to .64 per bytecode executed.

As discussed in Section 2, the paging simulations consider only the "old object" area of memory. This restriction is implemented by tracing only references to objects in

the initial image and ignoring references to objects allocated after beginning the script. No objects are matured into the old area in order to avoid measuring the artifacts of a maturing algorithm which is not yet tuned. In addition, the current scripts are so short that only a few objects created during the script mature before the end of the script.

Throughout, we assume an LRU replacement strategy, demand paging, and a page size of 1024 bytes. When an object is referenced, the entire object is brought into main memory. A reference to an object that straddles a page boundary may cause more than one page fault, but this is not likely to occur because of the small average size of Smalltalk objects. The mean size of objects in the standard BS image is thirty-two bytes. Fewer than three objects out of one thousand are larger than one page, and no object occupies more than three pages.

The scripts

Three different input scripts were created for our experiments (Table 2).

Name	length (bytecodes)	#references to old objects	image	#objects in image	#old objects used	% objects used
Benchmark	8.47 million	8.10 million	Smalltalk80	15,816	7,110	45
Choose	5.75 million	3.69 million	Smalltalk80	15,816	3,647	23
Rehearsal	5.61 million	3.31 million	Rehearsal	29,469	4,955	17

Table 2

Characteristics of the Input Scripts

The first two use a virtual image distributed by Xerox, described in Table 3. The memory for this image requires 506 pages when objects are aligned on full-word (4-byte) boundaries. The third script uses a virtual image of 1,007 pages that contains a large application. The average size of the objects used in all of the scripts is less than thirty-five bytes. The small size of the objects in these images reflects Smalltalk's representation of each entity as a distinct object. A *compiled method* object contains the code for a single procedure. The mean size of compiled methods is less than fifty bytes for all the code that we have measured. This observation holds for procedures that represent the style of a variety of programmers, and we suspect that the small size of compiled method objects is due to Smalltalk's emphasis on modularity and separate compilation.

Object Class	description	number of objects	% of objects	aggregate size (bytes)	% of space
CompiledMethod	code and literals	4120	26.1	182126	34.8
Symbol	identifier strings	3473	22.0	46288	8.8
Array	byte and word arrays	2450	15.5	112292	21.4
String	character strings	1540	9.7	39903	7.6
Point	x-y coordinates	496	3.1	3968	0.8
Float	floating point	221	1.4	884	0.2
Other		3516	22.2	138706	26.5

Table 3

Classes of Objects in the Smalltalk-80 Image

Smalltalk objects are represented as a sequence of integers and pointers to other objects. The images used in the paging simulations represent both integers and pointers as 32-bit words, as does the SOAR architecture. Compiled methods also contain a sequence of bytecodes. Most bytecodes consist of one eight-bit byte, but some are two

or three bytes in length. It is expected that the SOAR Smalltalk compiler will generate native SOAR code. SOAR instructions are all thirty-two bits and do not correspond exactly to Smalltalk bytecodes. The compiled SOAR code for a procedure will probably be several times larger than the same procedure compiled into bytecodes. Appendix A reports some results of experiments that consider the anticipated code expansion.

Smalltalk still has a small user community, and the principal interest of most Smalltalk-80 programmers is in developing the Smalltalk-80 environment. We cannot yet measure how eventual real users will interact with real applications packages and with the programming environment. Activities were selected for the three scripts by making guesses based on informal experience with the system and our desire to vary the tasks. The first script, Benchmark, executes a procedure that evaluates the speed of the bytecode interpreter [McCa83]. A sequence of editing, compiling, decompiling, searching, and formatting tasks is performed. A large variety of the system's facilities are exercised, but there is relatively little interactive input. The second script, Choose, records an interactive session during which a method is read from a file, edited, tested, compiled, and saved. The third script, Rehearsal, uses an application called Programming by Rehearsal, which allows curriculum designers to create graphical, computer-aided instruction packages [Goul82].

Restructuring schemes

This paper emphasizes the measurement of paging improvement and proposes one new restructuring algorithm, the DB grouping described below. In addition, we have studied two of Stamos' object grouping schemes, described as FIRSTUSE and DFS [Stam82]. FIRSTUSE assigns contiguous addresses to objects in the order of first reference. A FIRSTUSE reorganization must therefore be performed individually for each script. FIRSTUSE causes a minimum of page faults under demand-paging when there is sufficient memory to hold all objects used. It is the algorithm that gives the best results among the schemes we analyzed, but it is unrealizable because it depends on foreknowledge about the order in which objects will be referenced. The grouping called RANDOM was produced by assigning objects to virtual addresses in random order. We use it as a worst-case bound on paging performance. The other scheme proposed by Stamos, DFS, uses static information contained within the objects as a basis for restructuring. Smalltalk objects contain pointers to other objects, and we apply a depth-first search algorithm to visit all objects accessible within the image, since Stamos argues that a depth-first reorganization of the image improves paging behavior.

The DFS reorganization uses only information which is available statically. However, procedure call destinations are determined at runtime in Smalltalk, and a static analysis fails to show the dynamic relationship between procedure objects. The DB (dynamic binding) scheme augments the DFS strategy by considering information about the run-time binding of procedures to calls. In the SOAR design, the system maintains information about the procedure most recently associated with each call. To determine a DB grouping, we ran a script so that dynamic relationships could be observed. For every procedure we generated a list of procedures invoked from each calling location. Given this information, we perform a depth-first search of the object memory. When the search reaches a procedure object, the algorithm searches its list of called procedures as well as the object pointers contained within it.

A final grouping is the one currently used by BS. The image used for the scripts Benchmark and Choose was created by a depth-first search of the object space, so for these scripts BS and DFS groupings are the same. The Rehearsal virtual image has been

heavily changed over time, so the BS and DFS groupings differ. In a sense, the Rehearsal image demonstrates the degradation of of an initial restructuring due to the dynamic allocation and deallocation of objects. The results for the Rehearsal script with the BS and DFS groupings show the difference in performance before and after restructuring the current object memory.

4. Results of paging simulations

The number of page faults versus memory size is shown in Figure 2 for all scripts and object memory organizations. Main memory is assumed to be empty at the start of the simulations. Page faults taken during the first 200,000 bytecodes, while the system is initialized, are not counted in the results plotted in Figure 2. The page fault rate is greater for the Benchmark script than for Choose across all memory sizes because the benchmarks deliberately exercise a larger portion of the system. In general, the DFS and BS reorganizations give lower page fault rates than a random organization. The DB organization, which uses a limited amount of dynamically-acquired information, shows a small but consistent improvement over the static DFS grouping, as illustrated in Figure 2b. Figure 2c shows the improvement gained by applying DFS to the initial BS configuration of the Rehearsal image. Over the range of memory sizes from 275 to 600 pages, DFS requires fewer than half as many page faults as BS.

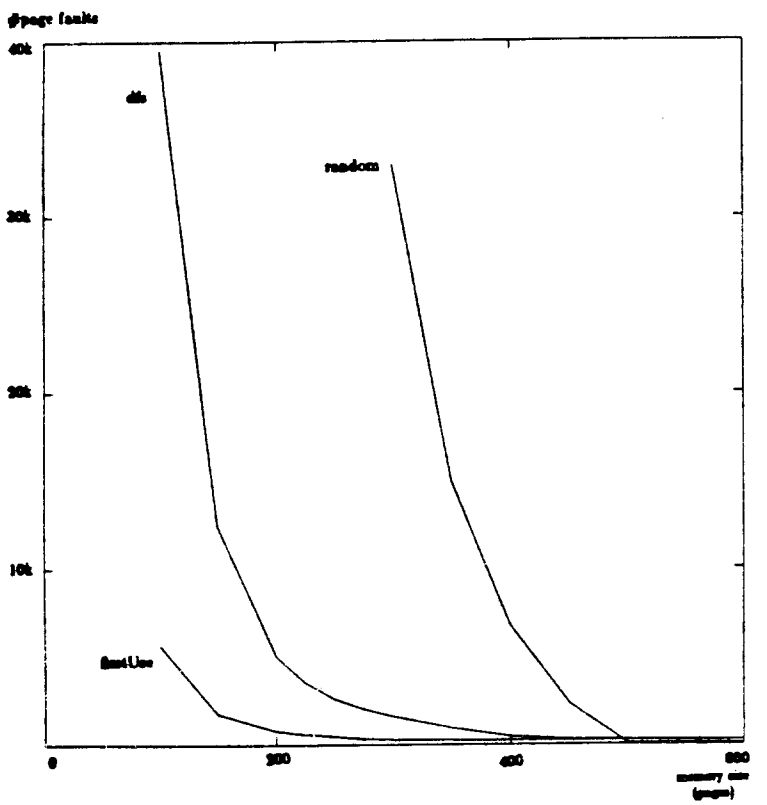

Figure 2a.

Benchmark script, number of page faults vs. memory size.

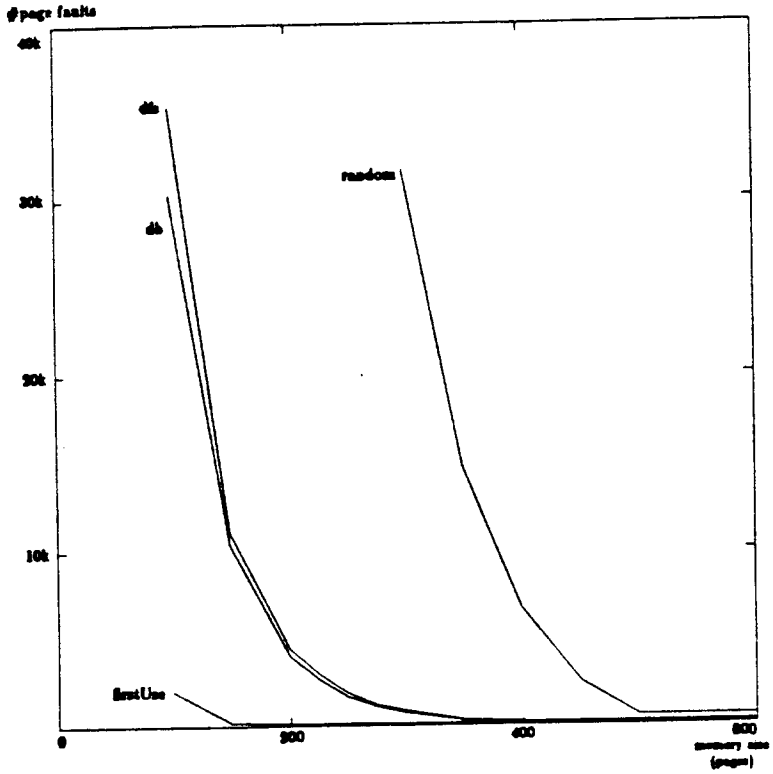

Figure 2b.

Choose script, number of page faults vs. memory size.

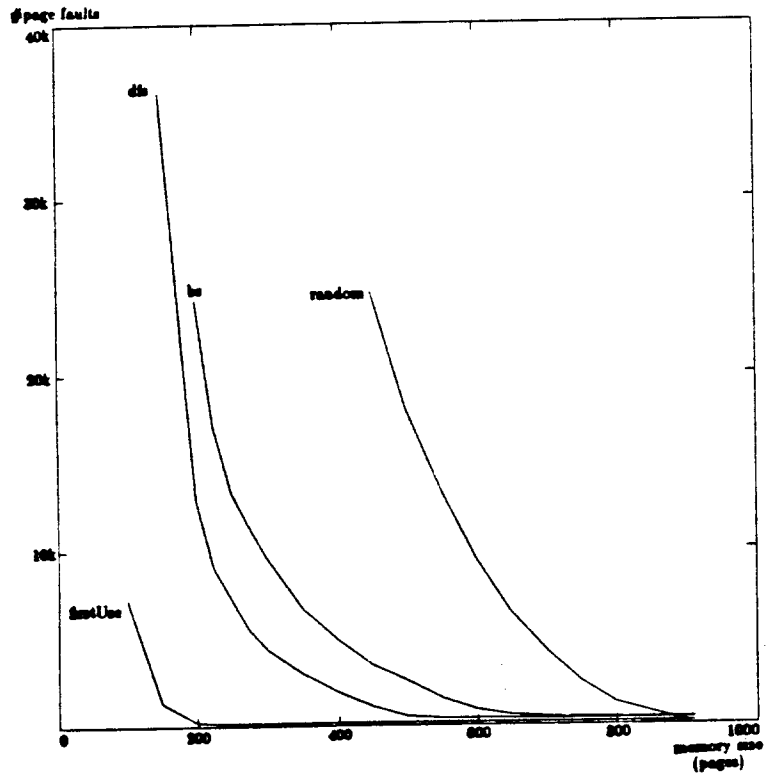

Figure 2c.

Rehearsal script, number of page faults vs. memory size.

Intuitively, the operating system of a personal computer must avoid pausing for intervals of time that are noticeable to the user. The only statistic presented in Figure 2 is the total number of page faults over the duration of an entire script. This information is inadequate for estimating the potential for noticeable pauses because it averages the page faults over the length of a script. Further information is obtained by examining the distribution of page faults over time. It is necessary to translate the intuitive notion of "minimizing pauses" into performance indices that consider the clustering of page faults in time and distinguish between steady-state and worst-case behavior. Transitions between program phases and high-level tasks [Denn75] [Madi76] are responsible for a large proportion of the page faults, represented by the spikes in Figure 3. This graph shows the number of page faults in each interval of 100,000 bytecodes for the Choose script with the DFS organization. On a Dorado, this is the equivalent of one-third to one-fifth of a second.

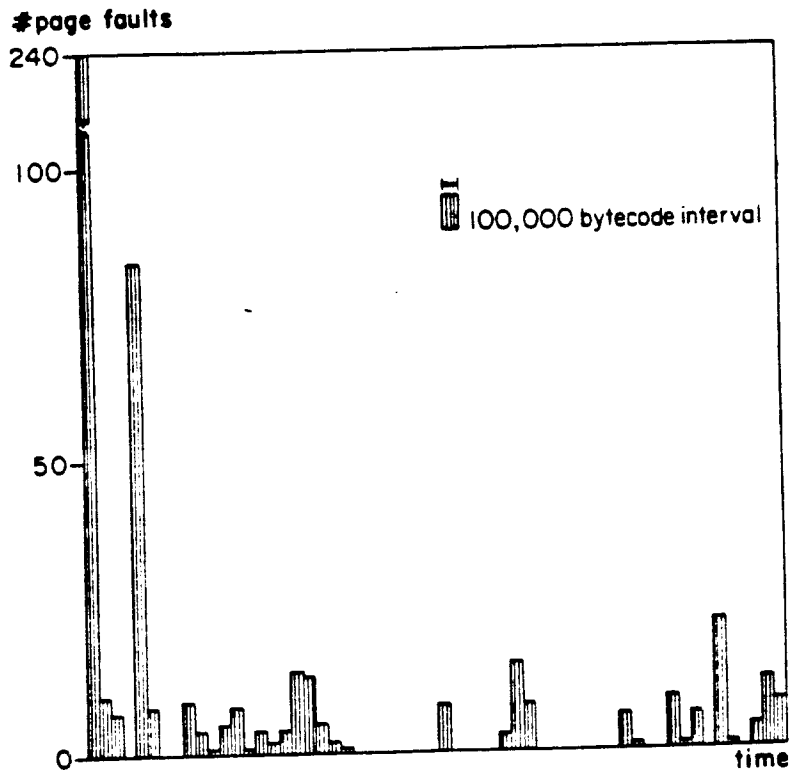

Figure 3.

Page faults in each interval.
Choose script, DFS organization, memory size = 350 pages.

The user will notice fewer pauses if the virtual memory system reduces the number of transitions that initiate page faults by keeping more localities in main memory. A user who begins a large task for the first time in a long while should expect a pause before it starts up. On the other hand, a user who is frequently switching between editing, Smalltalk's rapid incremental compilation, and program testing deserves responses for each activity without interruptions.

To quantify this, we distinguish between the *initial* fault for a page and *repeated* faults after the page has been removed from main memory. Figure 4 is based on the same data as the previous graph. The upper line indicates the total of *repeated* and *initial* page faults in each interval. The lower line plots *initial* page faults only. The black area between the two lines represents *repeated* page faults.

Object grouping strategies affect the amount of both total and repeated page faults. Figures 5a and 5b compare the DFS and BS reorganizations for the Rehearsal trace with 500 pages of old object memory. The upper line shows the number of page faults per interval for BS, and the black area below the lower line represents DFS. Figure 5a shows all page faults, whereas Figure 5b considers only repeated page faults. At this memory size, DFS represents an improvement over BS because it reduces especially the

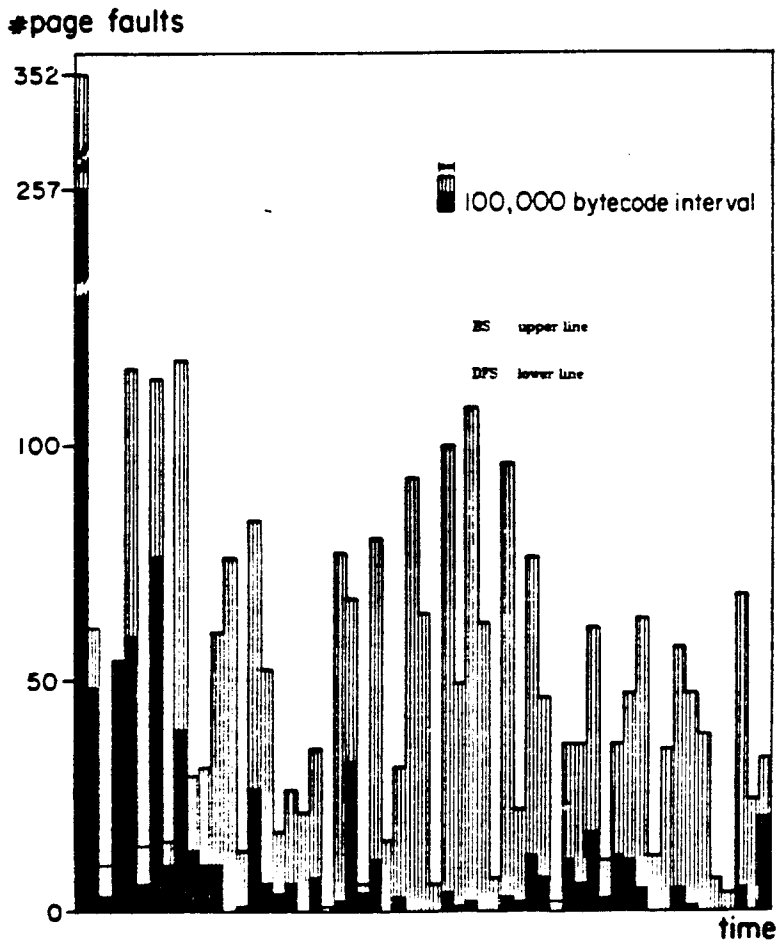

Figure 4.

Initial and repeated page faults.
Choose script, DFS organization, memory size = 350 pages.

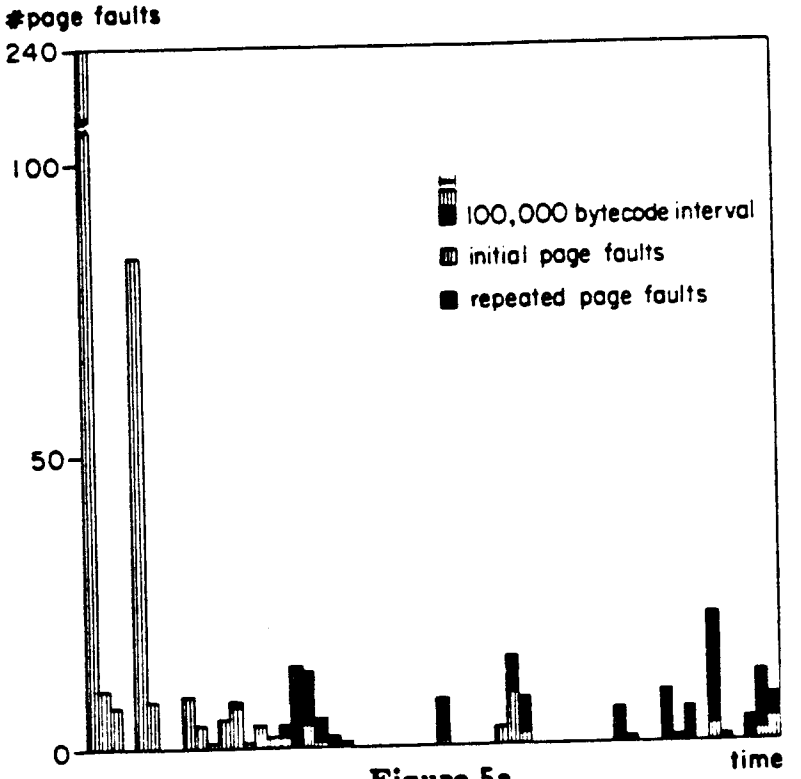

Figure 5a.

Comparison of DFS and BS for Rehearsal script.
Total page faults in each interval. Memory size = 500 pages.

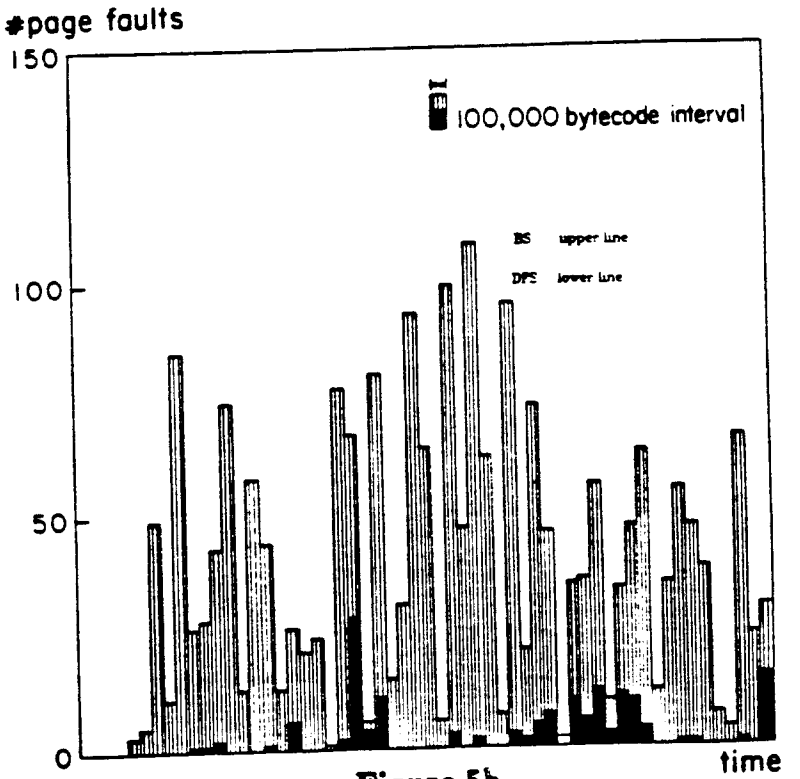

Figure 5b.

Comparison of DFS and BS for Rehearsal script.
Repeated page faults in each interval. Memory size = 500 pages.

number of repeated page faults. Holding constant the object grouping and replacement strategies, we can reduce the occurrence of repeated page faults by increasing the main memory size so that fewer objects are purged from main memory.

An extension of this idea considers simultaneously the effects of different restructuring strategies and memory sizes. In our approach, a *threshold* value specifies the number of page faults per interval that can be processed without introducing pauses. Among the factors affecting this threshold are the processor speed, the disk speed, and the time required to process a page fault. A paging simulation determines the number of *acceptable* intervals for which the number of page faults is no greater than a specified threshold. Figure 6 shows the percentage of acceptable intervals observed for different threshold values while running the Rehearsal script with the DFS reorganization. The value of the threshold is shown as a subscript and ranges from zero to five page faults. As the memory size increases, the percentage of acceptable intervals also increases.

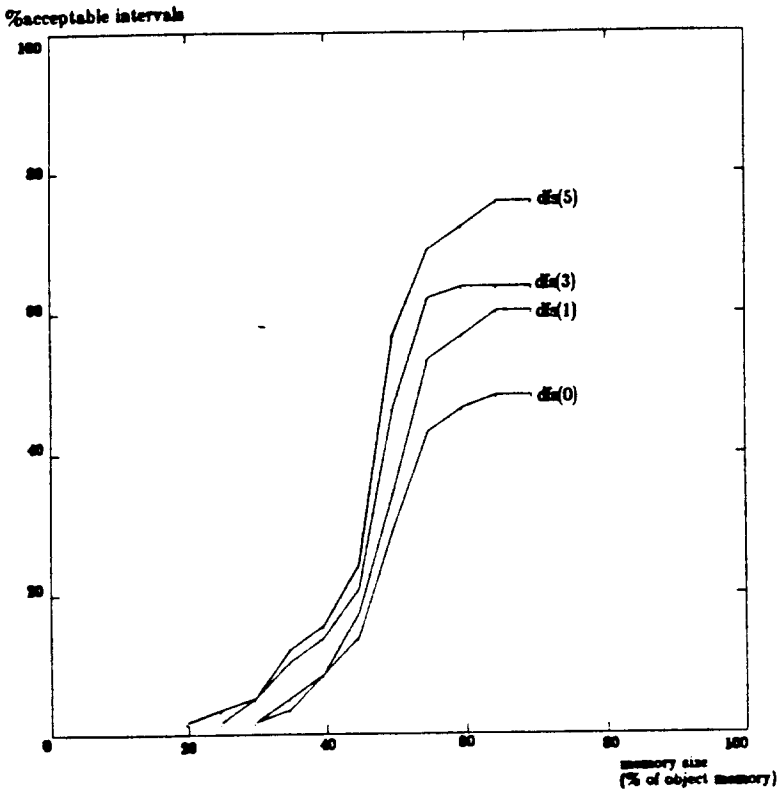

Figure 6.

Effect of threshold value, Rehearsal script.

The percentage of acceptable intervals may be compared for different combinations of memory size and object memory organization. Figures 7a and 7b show results for the Choose and Rehearsal scripts, respectively. Each line is labeled by the object organization scheme and subscripted by the threshold. Random placement is not adequate unless all, or nearly all, of the object memory fits in main memory simultaneously. A random organization requires that many pages be brought into main memory at the beginning of the script. If the memory is so large that pages are seldom replaced, few page faults occur during the remainder of the script. In Figure 7b, we again see the

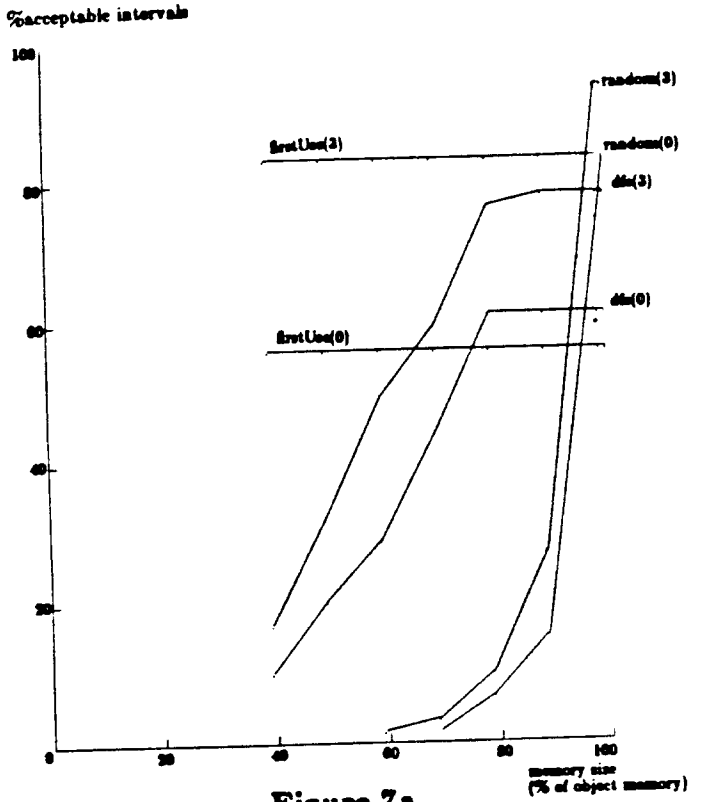

Figure 7a.

Comparison of restructuring schemes at different threshold values.
Choose script, total page faults.

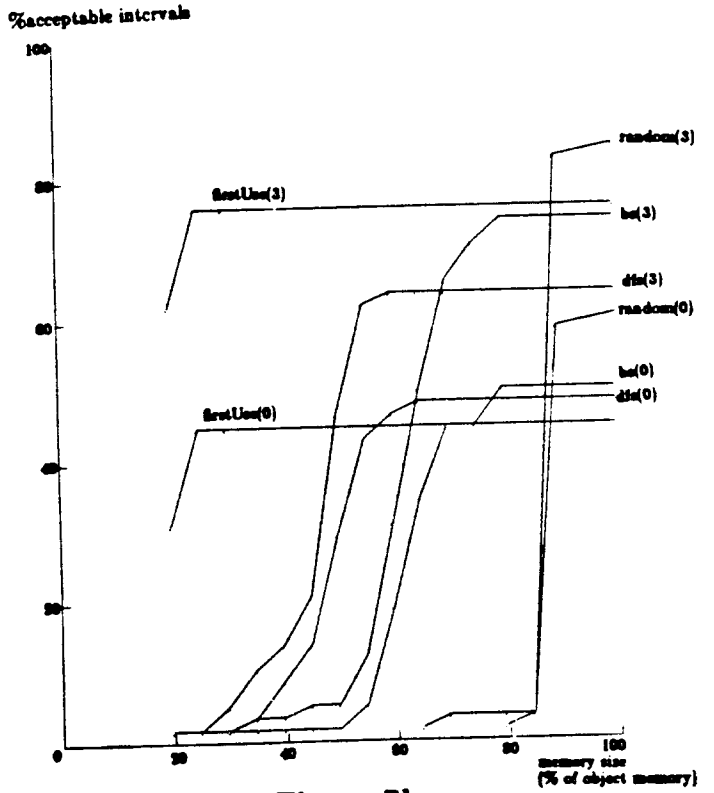

Figure 7b.

Comparison of restructuring schemes at different threshold values.
Rehearsal script, total page faults.

improvement of DFS over BS for the Rehearsal script, since the knee of the DFS curves come at memory sizes of about 600 pages, in contrast to 700 for BS.

In Figures 7a and 7b, no curve reaches one hundred percent. Memory is initially empty in the paging simulations; thus, some initial page faults are inevitable. Figures 8a and 8b present the same graphs for repeated page faults only. With a main memory large enough to contain about 70% of the old object space, the Choose script (Figure 8a) causes no more than three repeated page faults in any interval. For the Rehearsal script and the DFS grouping, similar performance is achieved when main memory holds about 60% of the image. Seventy percent of the object memory size is needed by the Rehearsal script to perform at the same level with the BS organization.

The threshold criterion presents the results of the simulations in a way that allows the system designer to understand the effects of memory size and object grouping. It helps the designer figure out how far past the knee of the curves in Figure 2 the memory size must be in order to obtain a desired level of performance.

Complete listings of the simulation results appear in an appendix to the MS project report [Blau83].

5. Conclusions and directions for future research

We expect consistently good response from a high-performance personal computer. Each page fault delays execution; when page faults occur in clusters, the pauses become noticeable. We have analyzed the clustering of page faults over time to reveal these peaks of paging activity. Our simulations indicate that program restructuring can decrease both the average page fault rate and the number of noticeable pauses. We believe that a paging virtual memory system that exploits program restructuring can achieve high performance for an object-oriented personal computer.

The object memories and software that we studied were developed at Xerox on Smalltalk-80 systems that do not provide enough virtual address space. It will be important to measure how these results scale when larger object spaces are available to users. The Rehearsal script uses an image that is approximately twice as large as the image used by the Choose script. Comparing results for the two scripts suggests that the need for physical memory may grow more slowly than the increase in virtual memory size. According to the analysis in the previous section, we would like at least 350 pages of main memory for the Choose script, enough to hold 69% of the virtual pages. Comparable performance is found for the Rehearsal script with 600 pages of memory, or 59% of the image. The largest existing object memories are not much larger than one megabyte, and measurements of larger object memories are needed before stronger conclusions can be drawn. Because larger virtual memories have never been available to Smalltalk-80 programmers, there is no experience to suggest what the contents of those memories will be nor how they will be used.

The simulation results indicate that the grouping of objects on a page must reflect referencing patterns in order for paging to work successfully. Depth-first search of the object space is one grouping strategy that improves the locality of reference, but other algorithms should be explored. Since a purely static analysis of the object memory inadequately characterizes the dynamic relationships among procedures, it seems particularly important to explore schemes that consider information obtained by monitoring dynamic behavior. Our dynamic procedure binding algorithm that considers a limited amount of dynamic information improved on depth-first search by about ten to fifteen percent in our experiments. Further improvements might be obtained by refining this algorithm or considering other dynamically observed information.

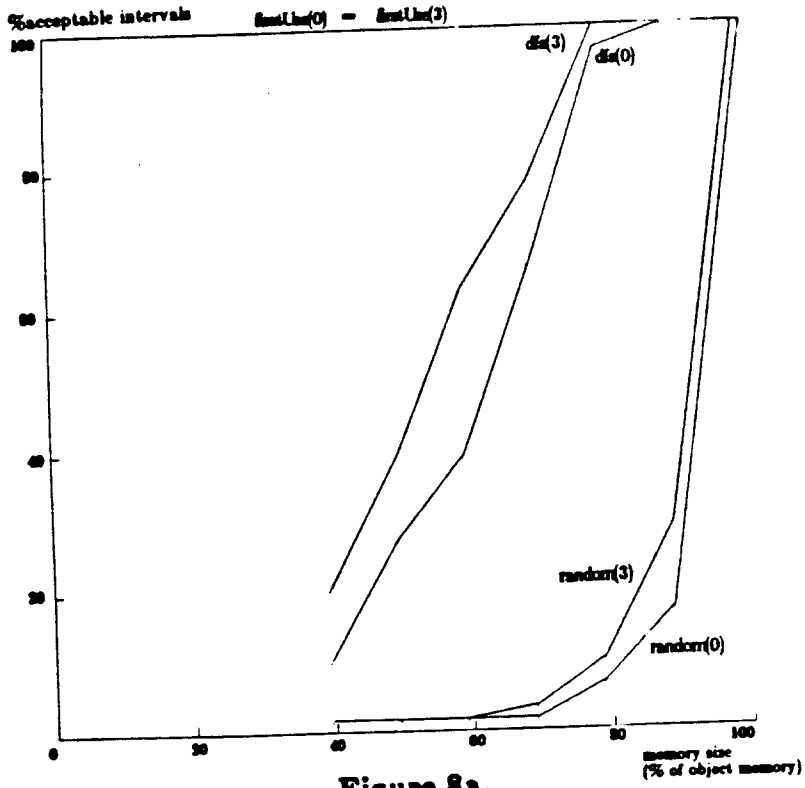

Figure 8a.

Repeated page faults, Choose script.

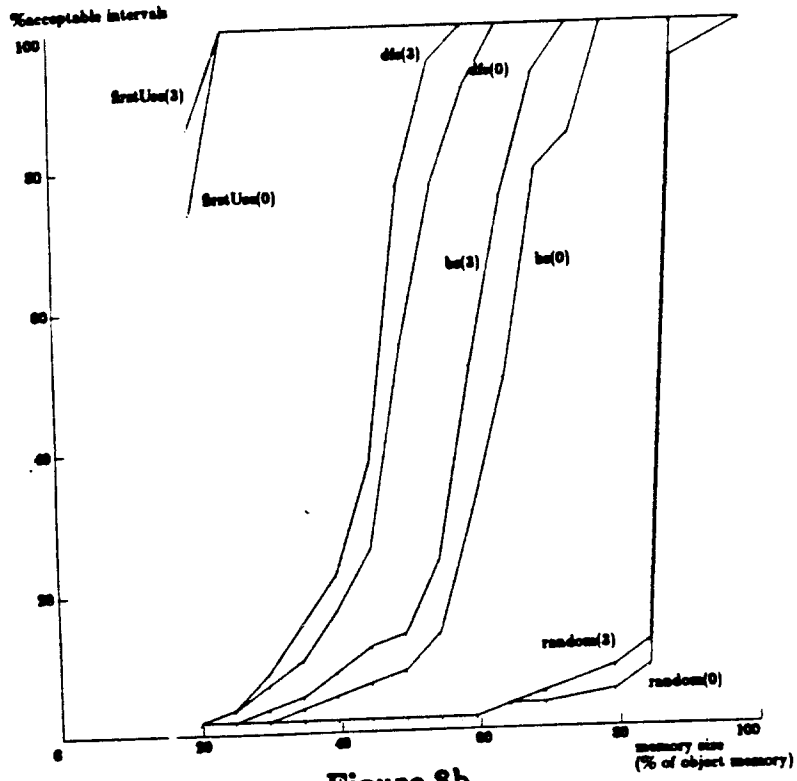

Figure 8b.

Repeated page faults, Rehearsal script.

The simulations measured improvements due to an initial static restructuring of the object memory performed before the beginning of the script. In a system supporting real applications, restructuring must be robust under dynamic changes to the "old object" area of memory. Further data are needed about the degradation of statically-restructured object memories. The BS organization for the Rehearsal image is an example of an object memory whose organization deteriorated as changes were made to the image. The improvement achieved by performing a DFS reorganization demonstrates the need to periodically regroup objects. Monitoring systems over the course of hours or days is one approach to estimating how frequently restructuring must be applied to the object memory. The Smalltalk group at Berkeley plans to develop a reorganization and garbage collection procedure that can be performed when a personal computer is not being used interactively, for example at night.

Another topic for research involves the separation of old and new objects. Only a small percentage of dynamically-allocated objects ever enter the "old" address space. Algorithms that exercise some care in assigning virtual addresses to mature objects may help maintain locality in the object memory, and such schemes should be explored. Think time or the time spent waiting for objects to be paged in from disk might be used to detect dynamic change in the object memory or to apply a restructuring strategy. Large-scale changes to the system will probably require explicit reorganization.

Acknowledgments

I would like to thank my faculty advisor, Prof. Domenico Ferrari, and Prof. David Patterson for their guidance in this research. David Ungar, the author of Berkeley Smalltalk, made sure that BS worked for me and, most importantly, persisted in asking hard questions. His insights into the relationship between garbage collection and paging have driven the design of the SOAR memory system. The implementation of the input script facility followed a suggestion by Dave Patterson. Members of the Software Concepts Group at Xerox PARC have been generous with the time, advice, and software they have given the Smalltalk group at Berkeley. In particular, I would like to thank Peter Deutsch, Ted Kaehler, and Glenn Krasner for sharing their experience with us; Adele Goldberg for giving us a chance to explore the Smalltalk-80 system; and Laura Gould and Bill Finzer for providing Programming by Rehearsal. Bruce D'Ambrosio instrumented BS to extract the dynamic procedure binding information. Bill Reeves suggested many improvements to an early draft of this paper.

References

[Baba81] Babaoglu, O. and W. N. Joy. "Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Referenced Bits," Proc. Eighth Symp. on Operating Systems Principles, in *Operating Systems Review*, 15, 5 (Dec. 1981), 78-86.

[Bade82] Baden, S. "High Performance Storage Reclamation in an Object-Based Memory System," M.S. report, University of California, Berkeley, 1982.

[Ball83] Ballard, S. and S. Shirron. "The Design and Implementation of VAX/Smalltalk-80," in Krasner, G., ed. *Smalltalk 80: Bits of History, Words of Advice*. Reading, MA: Addison-Wesley, 1983.

[Blau83] Blau, R. "Paging on an Object-oriented Personal Computer for Smalltalk," M.S. report, University of California, Berkeley, June 1983.

[Chu76] Chu, W. W. and H. Opderbeck, "Program Behavior and the Page-Fault-Frequency Replacement Algorithm," *Computer*, 9, 11 (Nov. 1976), 29-38.

[Denn75] Denning, P. J. and K. C. Kahn, "A Study of Program Locality and Lifetime Functions," Proc. Fifth Symp. on Operating Systems Principles, in *Operating Systems Review*, 9, 5 (Nov. 1975), 207-216.

[Deut82a] Deutsch, L. P. Lecture at Univ. of California, Berkeley (Feb. 1982).

[Deut82b] Deutsch, L. P. Private communication.

[Ferr74] Ferrari, D. "Improving Locality by Critical Working Sets," *CACM*, 17, 11 (November, 1974, 614-620.

[Ferr76] Ferrari, D. "The Improvement of Program Behavior," *Computer*, 9, 11 (Nov. 1976), 39-47.

[Gold83] Goldberg, A. and D. Robson. *Smalltalk-80: The Language and its Implementation*. Reading, MA: Addison-Wesley, 1983.

[Goul82] Gould, L. and W. Finzer. "Programming by Rehearsal," unpublished manuscript, Palo Alto, CA: Xerox PARC, Software Concepts Group, Dec. 1982.

[Hagm82] Hagmann, R. B. and R. S. Fabry. "Program Page Reference Patterns," ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, 1982, 20-29.

[Hatf71] Hatfield, D. J. and J. Gerald. "Program Restructuring for Virtual Memory," *IBM*Systems J. 10, 3 (1971), 168-192.

[Kaeh83] Kaehler, T. and G. Krasner. "LOOM-Large Object-Oriented Memory for Smalltalk-80 Systems," in *Smalltalk 80: Bits of History, Words of Advice*. Reading, MA: Addison-Wesley, 1983.

[Madi76] Madison, A. W. and A. P. Batson. "Characteristics of Program Localities," *Comm. ACM*, 19, 5 (May 1976), 285-294.

[McCa83] McCall, K. "The Benchmarks," in *Smalltalk 80: Bits of History, Words of Advice*. Reading, MA: Addison-Wesley, 1983.

[Patt82] Patterson, D. A. and C. H. Séquin. "A VLSI RISC," *Computer*, 15, 9 (Sep. 1982), 8-21.

[Snyd79] Snyder, A. *A Machine Architecture to Support an Object-Oriented Language*. Cambridge, MA: M.I.T. Laboratory for Computer Science, March, 1979. MIT/LCS/TR-209

[Stam82] Stamos, J. W. *A Large Object-Oriented Virtual Memory: Grouping Strategies, Measurements, and Performance*. Palo Alto, CA: Xerox PARC, Software Concepts Group, May, 1982. SCG-82-2.

[Unga83a] Ungar, D. M. and D. A. Patterson. "Berkeley Smalltalk: Who Knows Where the Time Goes?" in *Smalltalk 80: Bits of History, Words of Advice*. Reading, MA: Addison-Wesley, 1983.

[Unga83b] Ungar, D. M. *A High-Performance Smalltalk Computer*. Ph.D. dissertation, University of California, Berkeley, in progress.

Appendix A
Increases in Object Sizes on SOAR

Two factors will increase the size of objects in the SOAR Smalltalk system. First, every object will have a four-word (sixteen-byte) header that our paging simulations ignored. Second, Smalltalk methods will be compiled into SOAR instructions rather than bytecodes, and compiled method objects are expected to be several times larger. A pessimistic upper bound on the size of SOAR code is twelve times the length of the corresponding bytecodes. This bound was derived by estimating the number of SOAR instructions required to translate each category of bytecodes into SOAR code. The worst case is for a *send*, where the code expansion factor will be twelve. Many bytecodes, however, correspond to a single SOAR instruction. For other bytecodes, such as those which push constants onto the evaluation stack, it may not be necessary to generate any SOAR code. Work on a SOAR Smalltalk compiler is in progress. Although it is clear that the real code expansion factor will be much smaller than twelve, we do not yet have a good estimate.

Compiled method objects contain both code and data. The size of the data portion is expected to be approximately the same in SOAR as it is in BS. In estimating the effects of code expansion, our algorithm multiplies the entire size of the compiled method by the expansion factor. This gives an inflated size estimate, because we expand data as well as code.

Some size characteristics of the objects used in the Choose script are given in the table below.

Number of old objects referenced	3647
Number of compiled methods referenced	1536
% compiled methods in objects referenced	42%
Aggregate size of old objects referenced	184,344 bytes
Aggregate size of compiled methods referenced	62,020 bytes
% of space taken by compiled methods	34%

The space consumed by the object table is not included in the sizes given above. Each object requires a sixteen-byte entry in a paged object table. The object table is implemented as an array; it has slots for the maximum number of objects that can exist. In the best case, the object table entries for all of the objects used in the script are compacted into as few pages as possible. These entries then would consume fifty-seven pages, or nearly one-third as much space as the objects themselves. If the entries are scattered more randomly in a larger object table, the amount of main memory devoted to object table pages could be quite large and depends on the locality of reference within the object table.

Estimates for SOAR

If we multiply the size of all compiled methods by four and then add sixteen bytes of header to the size of all objects, the objects used in the Choose script will require 414,168 bytes instead of 184,344. Thus, with a code expansion factor of four, we can anticipate a need for 200 to 250% increase in the amount of memory needed to hold all of the objects used by this script. Similarly, a code expansion factor of eight gives a projection of 662,308 bytes of objects, or about 350 to 400% of the current size.

A paging simulation was performed assuming a 400% code expansion and sixteen-byte object headers. The Choose script was run using the DB (dynamic procedure binding) reorganization both with the current object sizes and the estimated SOAR sizes. With the current sizes, there were no repeated page faults for physical memories with at least 400 pages. A similar level of performance was obtained for the SOAR sizes when 900 pages were available. The results of the simulation agree with the static estimates given above, which predicted a need for a 250% increase in memory size.

In conclusion, it is reasonable to expect that the working set size for SOAR will increase by a factor less than the average code expansion. It remains for a working compiler to show what the real code expansion factor will be.