Palo Alto Research Centers

# A Large Object-Oriented Virtual Memory: Grouping Strategies, Measurements, and Performance

James William Stamos

XEROX

# A Large Object-Oriented Virtual Memory: Grouping Strategies, Measurements, and Performance

by James William Stamos

SCG-82-2 May 1982

Corporate Accession P82-00053



**Abstract**: See page 2.

This report is a slightly revised version of a thesis submitted to the Department of Electrical Engineering and Computer Science at MIT in partial fulfillment of the requirements for the degrees of Bachelor of Science and Master of Science.

**CR Categories**: 4.35, 4.6, 6.34.

**Key words and phrases**: virtual memory, paging, object-oriented system, program restructuring, initial placement, static grouping, measurement, reference behavior, Smalltalk.

XEROX

PALO ALTO RESEARCH CENTERS
3333 Coyote Hill Road / Palo Alto / California 94304

## Abstract

The Smalltalk-80 system is an object-oriented programming environment for which a novel virtual memory is being constructed. This Large Object-Oriented Memory (LOOM) maintains two distinct address spaces and compresses pointers when an object is swapped from disk to core. Although objects are the logical unit of transfer between disk and core, LOOM swaps disk pages between the disk and the in-core disk buffer. The grouping of objects on disk pages can be a critical factor in the determination of speed of Smalltalk application programs.

An examination of the reference behavior of Smalltalk-80 provides useful insights for designing grouping strategies and explaining virtual memory performance. After reviewing the experimental methodology and the Large Object-Oriented Memory, this thesis describes nine static grouping techniques and a reference stream compression algorithm. Performance measurements taken from simulations of LOOM and a conventional page-swapping virtual memory are discussed and compared. The effects on the page fault rate of modifying parameters and policies of LOOM are described and an evaluation of LOOM-like virtual memories is offered.

In terms of the number of page faults, an object-oriented virtual memory generally outperforms a page-swapping virtual memory for a range of small memory sizes that depends on the particular grouping strategy. The existence and stability of good groupings also impact the choice of a virtual memory design.

## Acknowledgments

This thesis came into reality only with the guidance, support, and encouragement of numerous friends. I must thank the XEROX Palo Alto Research Centers (PARC) for their extraordinary facilities. I am deeply indebted to the entire Software Concepts Group at PARC, for both their technical assistance and their pleasant working environment.

Special thanks go to Peter Deutsch and Dan Ingalls for asking probing questions, making invaluable suggestions, and suffering through the first draft of this thesis.

My faculty supervisor, David Reed, provided key insights and helped me transform a jumble of ideas into a coherent whole.

Finally, I must thank my friend and supervisor at PARC, Ted Kaehler, for his continued support, assistance, and guidance throughout this endeavor.

# Table of Contents

# List of Figures

# 1. Preliminaries

## 1.1 Introduction

Because most computing systems are configured with a multilevel memory, a primary problem has been to determine the distribution as well as the movement of information between levels of memory. Early attempts at memory management were manual and became known as the overlay solution. The programmer organized both code and data into blocks and explicitly moved these blocks between memory devices. These techniques became automated when assemblers and compilers analyzed the structure of programs, partitioned them into blocks, and automatically moved the blocks between primary memory and the secondary memories [ACM].

Virtual memories [COFF, DENN, PARM] soon replaced the overlay technique as the primary focus of those researchers and programmers interested in memory management. Sayre claimed that virtual memory techniques were competitive with and likely superior to overlay strategies [SAYR]. Empirical results demonstrated that with only a small bit of knowledge of the actual (not the virtual) environment, programmers could create code with much better paging performance [BRAW68, BRAW70]. Numerous suggestions for programming in a virtual memory environment were proposed and compiled [GUER, KUEH, McKEL]. The key property that accounts for the exceptional level of performance of virtual memories is called *locality of reference* [DENN]. Numerous program behavior studies [BELA66, FINE] have empirically supported the long-observed locality property.

Increasing the locality of programs to realize better paging performance became the next goal. Code was partitioned into blocks [BAER72, KERN, LOWE, RAMA, VERH] and related blocks were assigned to the same virtual page in a process of pagination. Numerous researchers have investigated this process of restructuring programs and then distributing code to maximize locality [COME, FERR74, HATF, INFO, JOHN, TSAO].

With the advent of high level, object-oriented programming languages such as CLU [LISK], recent studies have examined object-oriented, virtual memory management policies. Such schemes manage real memory in terms of objects rather than disk pages. Bishop [BISH] partitions the virtual memory into a number of reasonably-sized areas to allow the efficient garbage-collection of very large address spaces. His dissertation describes a *mover* that dynamically relocates objects in an effort to increase the locality of reference. In a paper design of an object-based personal computer, Luniewski [LUNI] builds upon Bishop's scheme of an object-oriented virtual memory and dynamically relocates objects by the breadth-first traversal of the compacting garbage collector. Snyder [SNYDa], on the other hand, obviates the need for organizing objects in the virtual memory

by assuming the future existence of fast-access secondary storage devices that are efficient enough to swap even small objects instead of conventional pages.

## 1.2 Background

Smalltalk [XERO] is a high level, object-oriented, programming environment developed at Xerox PARC. It runs on personal computers with powerful graphics capabilities, such as Altos and Dorados, that are normally attached to the local network called Ethernet [METC]. Smalltalk is a collection of interlinked objects. Consider, for example, the set of instructions an object executes when it receives a specific message. Both the source text and compiled code for these instructions are objects. User-defined types and primitive types are also represented as objects. All instantiations of these types, including the frames of the run-time stack, are ordinary objects.

The current implementation of Smalltalk will eventually be supported by a 31-bit virtual memory system called the Large Object-Oriented Memory (*LOOM*) [KAEH]. LOOM swaps objects between memory levels on demand. To support this object-oriented view, memory pages are swapped between core and disk (or between core, disk, and a remote file server, such as WFS [SWIN]) on demand. The desired object is then copied to or from the disk page. Since many objects can fit on a single page, one page fault typically transfers a set of objects to the in-core disk buffer. Until this page is flushed from the disk buffer, swapping other objects on that page into primary memory may be accomplished without incurring additional page faults. It is likely that intelligent groupings of Smalltalk objects on disk pages will be a critical factor in the determination of the speed of Smalltalk application programs.

Of prime importance is the persistence of the programming environment. Many Smalltalk objects have lifetimes that transcend single user sessions. It is therefore possible to consider the objects comprising the system and their interactions. This information can then be used to place related objects on the same page. LOOM is not committed to any specific strategy for grouping objects on pages and therefore places no constraints on the set of grouping schemes. Such an environment is conducive to experimenting with various algorithms for grouping objects on pages in an attempt to reduce the frequency of page faults.

A simulation of a conventional page-swapping virtual memory was employed to determine the effects of such static grouping algorithms on that type of virtual memory. In addition, a direct comparison between object-oriented and page-oriented virtual memories under various grouping strategies is reported. In a paged virtual memory, whenever an object on a page is referenced, all objects on that page are automatically transferred into primary memory. If not all of these objects are required, memory will be underutilized. In the object-oriented case, memory will be fully utilized. However, one drawback of such a scheme is that loading a page of objects into primary memory may require as many page faults as there are objects on that page. In terms of paging performance, the better type of virtual memory will depend upon the grouping strategy, execution sequence, and primary memory size.

## 1.3 Novel Extensions

The primary focus of this thesis will be an investigation into the performance of a small but varied collection of static grouping algorithms for object-oriented and page-swapping virtual memories. Much of the work done to support this endeavor concerns realms that previously had been largely unexplored. The object-oriented nature of Smalltalk provides some of the novelty; the remaining investigations are interesting in their own right:

> Memory accessing behavior for an interactive programming environment is analyzed on two levels. In addition to capturing detailed reference tendencies, efforts were directed at studying the paging requirements of Smalltalk. Unlike most published studies, which were concerned with physical attributes such as virtual memory addresses or page references, the emphasis here is on logical structures. Definitions and results are expressed in terms of programmer-chosen units (objects) rather than fixed-size pages.
>
> At times a distinction is made between code and data references. This partitioning allows their low-level behavior in an object-oriented environment to be analyzed in isolation. The two memory accessing behaviors are compared and contrasted, and their individual contributions to the composite picture are determined.
>
> Previous restructuring techniques tended to be concerned only with the code segments of a single program. A few also considered the areas for temporary data structures that were allocated at compile/assembly time. Smalltalk grouping schemes, on the other hand, are designed to accommodate an entire programming environment composed of existing code, data, and support structures.
>
> Finally, empirical measurements of the dynamic paging performance of both object-swapping and page-swapping virtual memories under various grouping algorithms were made. These results are then compared and contrasted.

## 1.4 Some Results

Grouping of code, data, and support structures can have a large impact on performance in an object-oriented programming environment. The simple, efficient grouping techniques employed in this study achieved substantial reductions in the number of page faults for both the page-swapping virtual memory and for LOOM. More complicated schemes that considered reference count information or knowledge derived from actual dynamic behavior did not provide additional improvements.

In the paged virtual memory, any reasonable grouping scheme substantially reduced the amount of paging from the level caused by the random, ungrouped initial placement. Differences between grouping schemes were not too significant. In LOOM, on the other hand, the amount of improvement in paging performance depended heavily on the type of grouping. Although LOOM was more sensitive to the type of the grouping scheme, grouping had less of an effect on LOOM than it did on the paged virtual memory. This result is due to the fact that LOOM selectively swaps information between primary memory and the disk. It does not necessarily swap all the information on a given page.

A direct comparison between the two types of virtual memories indicated that LOOM outperforms a paged virtual memory for a range of small memory sizes. Such results were invariably obtained

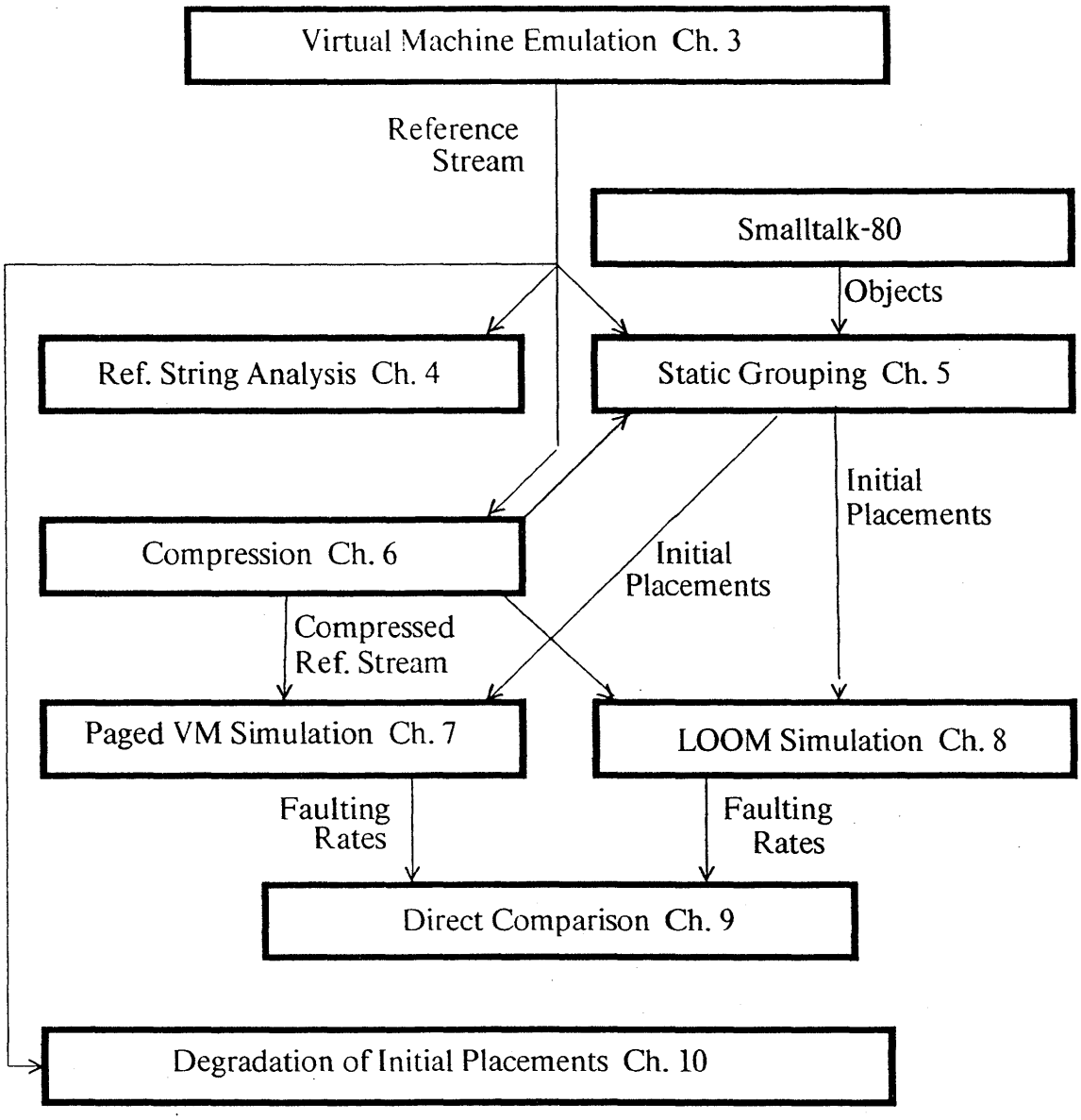


Figure 1-1 A Road Map

for warm starts, cold starts, simulations that did reference counting, and those that did not. In addition to the particular computation, the initial placement computed by the grouping strategy played an important role in determing the memory size interval for which LOOM outperformed the paged virtual memory. Since the length of this interval varied inversely with the quality of the initial placement, the desirability of a LOOM-like virtual memory will depend on the existence and stability of quality initial placements.

## 1.5 A Road Map

The structure of this thesis closely parallels the work whose results are reported herein (Figure 1-1). Chapter 1, the introduction, briefly surveys the history of virtual memories, motivates the problem, and outlines the important research areas and results. Chapter 2 discusses issues relevant to swapping strategies and introduces the LOOM virtual memory. A description of the virtual machine emulator used to generate reference traces is presented in Chapter 3, while a thorough analysis of the reference behavior of the Smalltalk virtual machine is contained in Chapter 4. Chapter 5 describes the grouping strategies and the static analysis performed on them. The next chapter details the algorithm used to compress a full reference trace into one suitable for driving virtual memory simulations. Chapters 7 and 8 discuss the simulated dynamic behavior of a paged virtual memory and LOOM. In Chapter 9, these two types of virtual memories are directly compared. Chapter 10 makes a brief excursion into the areas of memory management schemes and the stability of static groupings. The last chapter then surveys the accomplishments and failures of this endeavor.

# 2. Swapping Strategies and Grouping

## 2.1 Logical versus Physical Swapping

*Physical* units of information are swapped between memory levels if the amount of data transferred at one time depends only upon the characteristics of the hardware, microcode, and software implementing the virtual memory but not upon the data. In other words, the swapping size is data independent. *Logical* elements of information are swapped if computational entities are the units of transfer. Examples of logical swapping include transferring files, sets of objects, single objects, portions of objects, or individual fields. Physical units of swapping include bits, bytes, words, pages, and disk sectors.

Some systems combine these two extremes. LOOM, for example, swaps entire disk pages from the disk to an in-core disk buffer, but swaps Smalltalk objects between this buffer and the rest of core. Conventional virtual memories swap fixed-size pages between core and disk. Predictive schemes, which preload related pages in addition to the one causing a page fault, add a logical flavor to a physical swapping strategy. Another combination is a segment-swapping virtual memory in which each segment is an integral number of disk pages. The appropriate scheme depends on the underlying machine, the position in the memory hierarchy, the expected nature of reference patterns, and the computational overhead and complexity of the chosen algorithms.

## 2.2 Effectiveness of Grouping

As long as the interface to secondary memory dictates that fixed-size sets of bits are to be transferred, it is only reasonable to endeavor to structure both code and data so that a large percentage of the swapped information is used as quickly as possible. This statement holds for both object-swapping and page-swapping virtual memories. In both cases, most or all of the grouping decisions should be automated.

Grouping can only affect performance if the physical units of swapping are larger than the size of the logical elements. Differences between grouping strategies are likely to appear only when these sizes are within an order of magnitude. Consider, for example, grouping Smalltalk objects on disk pages. Because objects directly refer to a small set of other objects and are themselves typically referenced by only one object, most grouping algorithms will tend to partition the system into sets of related objects in a similar manner. One critical indicator of the feasibility of object grouping is the ratio between the size of a disk page and the average size of an object. If this value is extremely large, hundreds or thousands of objects would comprise a single unit of information transferred between memory levels. Most pages would contain many of these sets of related objects and different grouping strategies would have similar effects on performance. The particular technique employed would be the simplest and fastest.

If the length of an object were comparable with page size, some objects would span more than one page, but the average number of objects on a single page would be close to unity. For such systems, conventional page-swapping virtual memories are nearly equivalent to simple object-swapping schemes as far as the average amount of data transferred between object space and disk. Pointer compression, unresolved pointers, and other enhancements found in LOOM (section 2.4), however, are not directly applicable to such conventional systems.

At the other end of the spectrum lie systems in which the objects are far larger than the page size. It is still possible to swap entire objects to and from secondary memory, but the gains realized from such a scheme are highly dependent upon the normal types of reference patterns. Rearranging fields within objects may make sense in a page-swapping environment.

Grouping strategies will have different effects if a small number of objects fit on each page, because a set of related objects will span many pages. Hatfield suggests that the best results for a paged environment occur when 3 to 10 objects fit on a page [HATF]. Similar conditions should also suit object-swapping virtual memories.

## 2.3 A Hierarchy of Groupings

The size and nature of the units of information that are manipulated by grouping schemes depend upon the memory component under consideration as well as the types of transfer between these components. In a multi-level virtual memory configuration, there are numerous possibilities for restructuring information in order to enhance performance.

*Field swapping*, in which single fields are swapped, is similar to object swapping in that the size restrictions imposed by the hardware interface to secondary memory are hidden from the manager of primary memory. This scheme is equivalent to a page-swapping strategy that has a tiny page size and is typical of caches that do not prefetch information. Grouping, however, is not applicable at this memory level.

A more general approach is to partition the fields of each object into blocks such that each field is in exactly one block and fields in the same block are physically adjacent (Figure 2-1). Call such virtual memory schemes *block-swapping*. If the block size is constant, the result is essentially a page-swapping virtual memory when the blocks are the unit of swapping. Block size may differ from object to object and even within an object. If the blocking pattern is the same for all instances of a single, fixed-length class, this information may be kept in the class object and shared by all instances of the class. An interesting question for variable-length classes is the specification of the blocking pattern and its placement. High level blocking information may be retained within the class object but the particular parameters such as block size(s) may be held by each instance. In block-swapping virtual memories, the compiler can group information by reordering fields within blocks and/or blocks within an object.

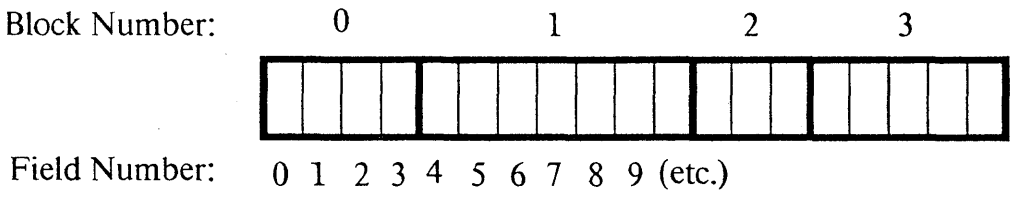


Figure 2-1 An Object Partitioned into Blocks

Hybrid schemes are also possible. For example, the virtual memory manager can group small objects on pages and swap them individually. Very large objects, such as local or remote files, may be partitioned into a number of blocks. Only the most recently used block(s) would remain in core. Although this scheme can place an upper bound on the quantity of information transferred at one time, one drawback is the overhead in time and space required to manage these "partial" objects.

A novel viewpoint is taken by the PIE [GOLD] personal information environment which is implemented in Smalltalk-76 [INGA78]. The smallest unit of information in PIE is an attribute of an object. Changes to the system, called *layers*, are collections of new attribute values for objects. For such a system, grouping, organizing, and/or swapping may be done on a layer basis rather than on an object basis.

Depending on the possible amounts of data transferred between memory levels and the size of the computational items and their substructures, a natural unit of swapping may be identified. If this unit is composed of finer structures which are themselves meaningful entities, these structures may be grouped and/or swapped to enhance performance when data is transferred between a pair of lower memory levels.

For example, if the expected number of objects on a page is ten, then it is feasible to group objects and swap single objects or small collections of objects. If the unit of swapping dictated by the underlying cache hardware is only four words, then fields ought to be grouped and collections of fields swapped.

Grouping at one memory level of the system neither requires nor precludes grouping at other levels. Memory configurations usually consist of a number of physical devices with different transfer rates, access times, error rates, capacities, and costs. Information is continuously shuffled from one layer of memory to another, while the quantity of information transferred is highly dependent upon the two memory levels involved. Therefore, the appropriate choice of entities to group or swap is strongly correlated with the particular aspects of the virtual memory performance being tuned. Once this choice has been made, however, most if not all of the actual grouping decisions ought to be automated.

## 2.4 LOOM

### *2.4.1 Object Swapping*

In order to avoid the complexities of analyzing a three-level memory system (the actual Smalltalk implementation strategy), only two levels will be considered (Figure 2-2). The large, slow secondary memory is a collection of disk pages which may be confined to the local disk, located on a remote file server, or replicated and distributed across the local network. For purposes of simplicity, the secondary memory will be modeled as a single repository containing all the pages required for any computation. Call this memory *disk*.

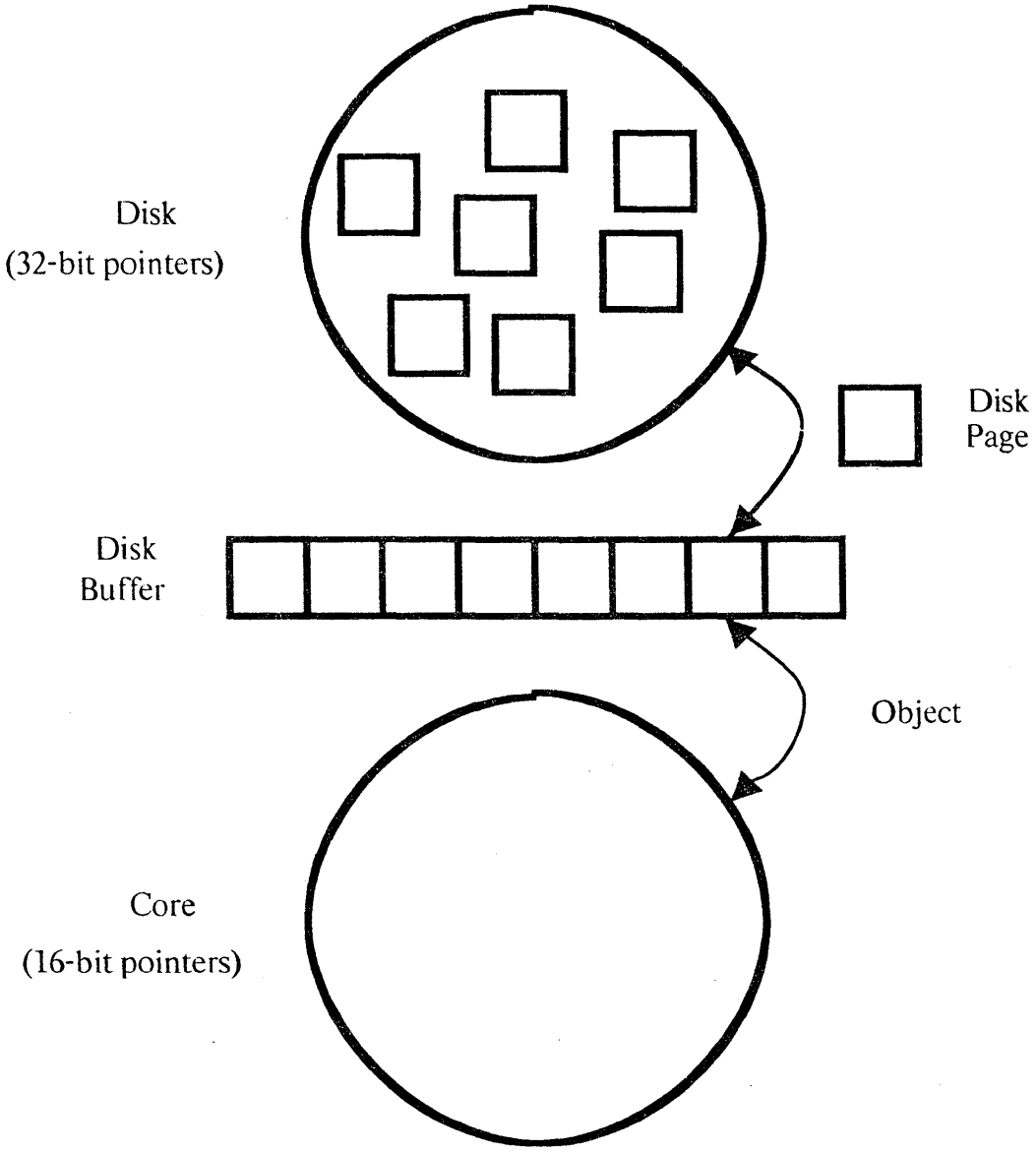


Figure 2-2 LOOM Virtual Memory

Primary memory is composed of an *in-core disk buffer* and *object space (core).* The disk buffer is a small, fixed-size collection of disk pages. Pages are swapped between the disk and this buffer on demand. The buffer acts as a simple cache for disk pages.

Objects are the unit of swapping between the disk buffer and core. At most, one copy of an object may be in core. Fields of an object may be read or written only if the object is in core. If the Smalltalk interpreter needs to access an object that is not in core and is therefore inaccessible, an *object fault* occurs. The required disk page is read into the disk buffer and the object is copied into core.

If there is not enough free space in primary memory to swap in an old object or create a new object, compaction is done. The memory manager collects all free words in primary memory into one contiguous block. If this free block is large enough to accommodate the memory allocation request, space for the new object is carved from the front of the block. Otherwise, some objects are removed from core and purged to the disk. Each object has an *in-use* bit that is set if the object has been touched by the virtual machine since the last time it was considered as a candidate for purging. If this bit is set, the virtual memory manager clears the bit and considers another object. Otherwise, the bit is not set. The object was not used in the recent past and is purged. If the object is clean, it is simply discarded. Otherwise, an object fault occurs. The appropriate disk page is read into the buffer and the current contents of the object are copied from core into the buffer.

One implication of object swapping is that the optimal object-fetching policy is not necessarily a demand policy. Formal proofs of the demand nature of the optimal strategy for page-swapping virtual memories exist [MATT]. However, since the cost is measured in page faults rather than object faults, all such proofs are not applicable to the object-swapping portion of such systems. Note that the optimal *page-fetching* policy for LOOM will always be a demand strategy.

### *2.4.2 Pointer Compression*

Interesting naming considerations arise when a Smalltalk object is transferred from core to disk, especially when the disk is a remote file server of a distributed database supported by the Ethernet. All objects have two formats: their representations in core and on the disk. In-core pointers are 16-bit indices into a table of in-core objects called the *OT.* Disk pointers are 32 bits long and are essentially disk addresses. Objects that change size are accommodated on the disk by using forwarding markers.

Whenever an object is brought into core, all of its pointers must be converted from a 32-bit format to a 16-bit format. The OT is used as a hash table with the 32-bit address of an object as its key. Since all in-core objects contain their 32-bit disk address, establishing the presence or absence of the desired object is straightforward. If such a 32-bit pointer references an in-core object, then the 16-bit name of the object is used.

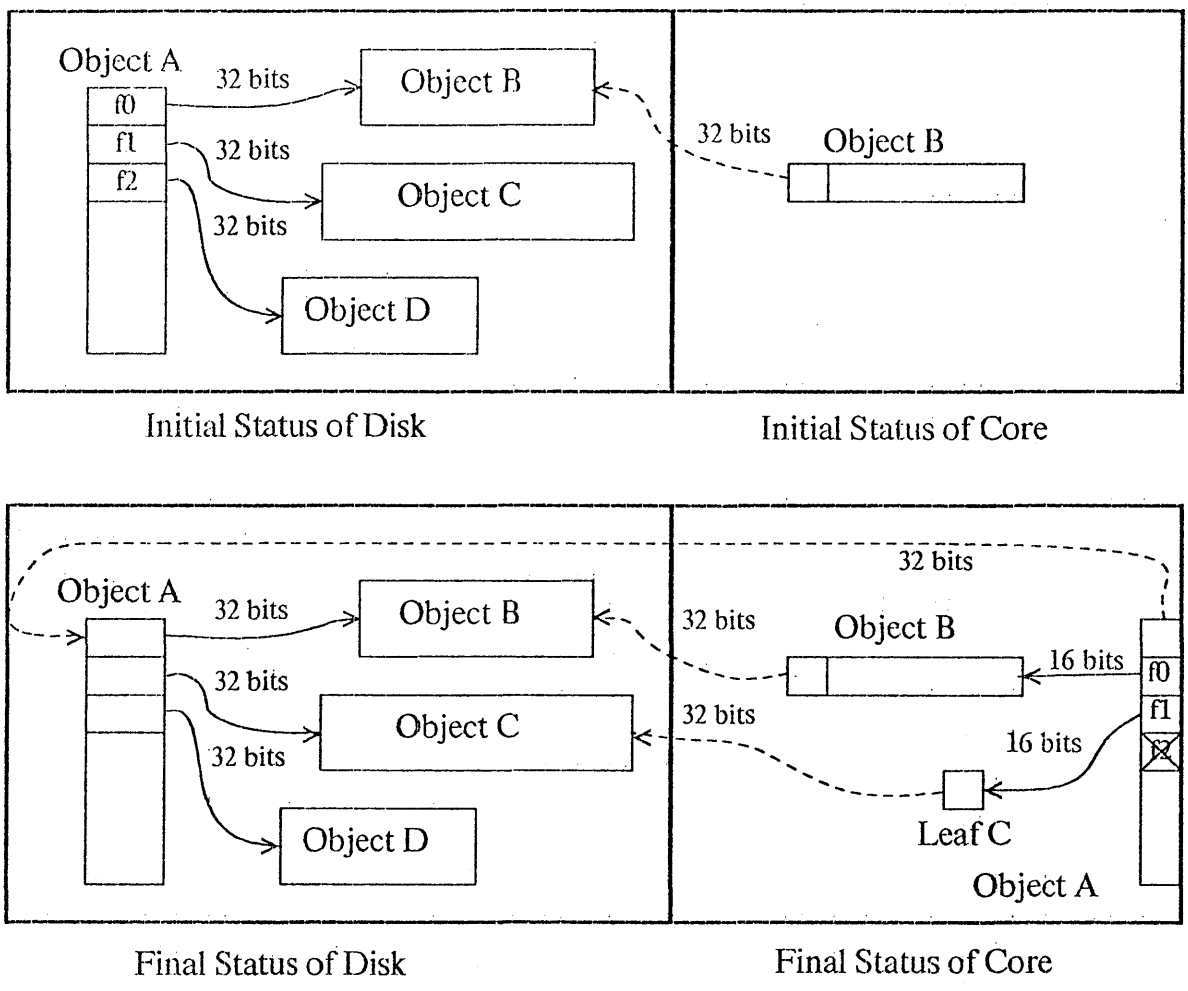


Figure 2-3 Pointer Compression

Consider the example shown in Figure 2-3, in which objects A, B, C, and D are all on the disk. Solid lines indicate object references, while dashed lines represent the 32-bit disk address held by each in-core object. These backpointers are the only type of disk reference allowed in primary memory. Note that there are no references from disk to core. Assume object B only is initially in main memory and that object A is needed for a computation. When A is swapped into core, all of its 32-bit pointers must be converted to a 16-bit format. Since B already has a 16-bit name, this name is placed in field f0 of the in-core version of A.

If the referenced object is not present, there are three alternatives. First, the referenced object may itself be swapped into core. While some objects may be prefetched and thus swapped into core before they are actually referenced, this scheme cannot always be used. The complete set of objects may not fit into core or the desired object may not be immediately available.

A second alternative is to create a *leaf* for the object, which is essentially the object without any of its data fields. This alternative was chosen for the reference to C contained in object A. An object may exist in core as an object or as a leaf, but not as both simultaneously. The small size of leaves allows a representative for an object to be in core that does not occupy the space needed by the full object. Leaves provide a level of indirection and allow the existence of short pointers to nonresident objects. For example, if an object being purged has a non-zero in-core reference count, it is replaced by a leaf in a process called *contraction*.

Another alternative is to replace the pointer with a *lambda*, as was done for the reference in A to object D. Lambdas are special references with a constant value used to indicate unresolved pointers. Whenever the Smalltalk interpreter comes across such a reference, the virtual memory manager refetches the disk representation of the object containing the lambda to determine the true pointer value. Since this process of *lambda resolution* can generate many page fetches per "in-core" object, its frequency of occurrence is a critical performance factor.

Dynamically created objects initially receive only a 16-bit name. Since these temporary objects tend to have relatively short lifetimes and are rarely purged, 32-bit names and a home location on the disk are not granted unless they are required. In order to purge a temporary object that has no counterpart on disk, disk space is first allocated. The disk address of the object becomes its external name.

Whenever an object is transferred to or from the disk, all pointers must be converted to the appropriate format. This name translation penalty trades off against increased pointer size and the reduced number of objects in core. Another benefit provided by this two-level name space is the lack of a fixed relationship between the name and location of an object in core and its location on the disk.

### *2.4.3 Storage Management*

Memory is reclaimed in LOOM by using reference counts. At times, three separate reference counts may be maintained for a single object. The true reference count is actually the sum of the individual counts. Disk objects contain the count of the holders of their 32-bit external name. In-core reference counts, which represent the number of holders of 16-bit internal names, are maintained in the OT. Leaves are also involved with LOOM's complex reference counting scheme and their role is briefly described in section 8.1.

### *2.4.4 Relevant Smalltalk Terminology*

A small number of terms with particular Smalltalk meanings need to be defined. Objects communicate by sending *messages*. The name of a message is called the *selector*. Upon receipt of a message, the *receiver* performs some set of actions, possibly sends other messages, and then returns a value. This sequence of instructions performed by the receiver is called a *method*. Sending a message in Smalltalk corresponds to calling a procedure in a conventional programming language. The context is changed in both cases by pushing a new frame on the stack and updating the program counter (PC) to point to the first instruction in the new method/procedure. The source-level representation for a method is a sequence of Smalltalk statements that may be compiled to form object code intelligible to the virtual machine. This object code is represented as a sequence of compile-time literals followed by a collection of 8-bit bytecodes and is a called a *CompiledMethod*. A CompiledMethod may be decompiled to produce a close approximation of the original source code. While the word "method" may refer to either the source-level statements or the object code, the appropriate meaning will usually be clear from the context.

Every Smalltalk object belongs to exactly one *class*. This class determines the internal representation of the object as well as the set of messages such an object is prepared to accept. All *instances* of each class share a collection of methods that are located in a single object held by the class object. This repository, called the *message dictionary*, maintains a mapping between selectors and CompiledMethods. Under this naming convention, the receiver of a message determines the context in which to resolve the specific selector. Therefore, a single selector appearing in a method may refer to any number of CompiledMethods. The binding of the selector to a specific CompiledMethod is not done until run time. Classes are arranged in a superclass hierarchy for the purpose of code inheritance. This arrangement further complicates the binding between selectors and CompiledMethods.

Consider the hypothetical number hierarchy depicted in Figure 2-4. Assume a SmallInteger receives a message with selector M. The Smalltalk interpreter first determines the class of the receiver, which in this case is SmallInteger. Each class maintains the mapping from selectors to methods in its own message dictionary. If M is found in the message dictionary associated with the class SmallInteger, the corresponding method is executed. Otherwise, the Smalltalk interpreter looks in the *superclass* field of the object that is the class SmallInteger and finds a reference to the Integer class. The message dictionary of this class is then checked for M. This search process continues

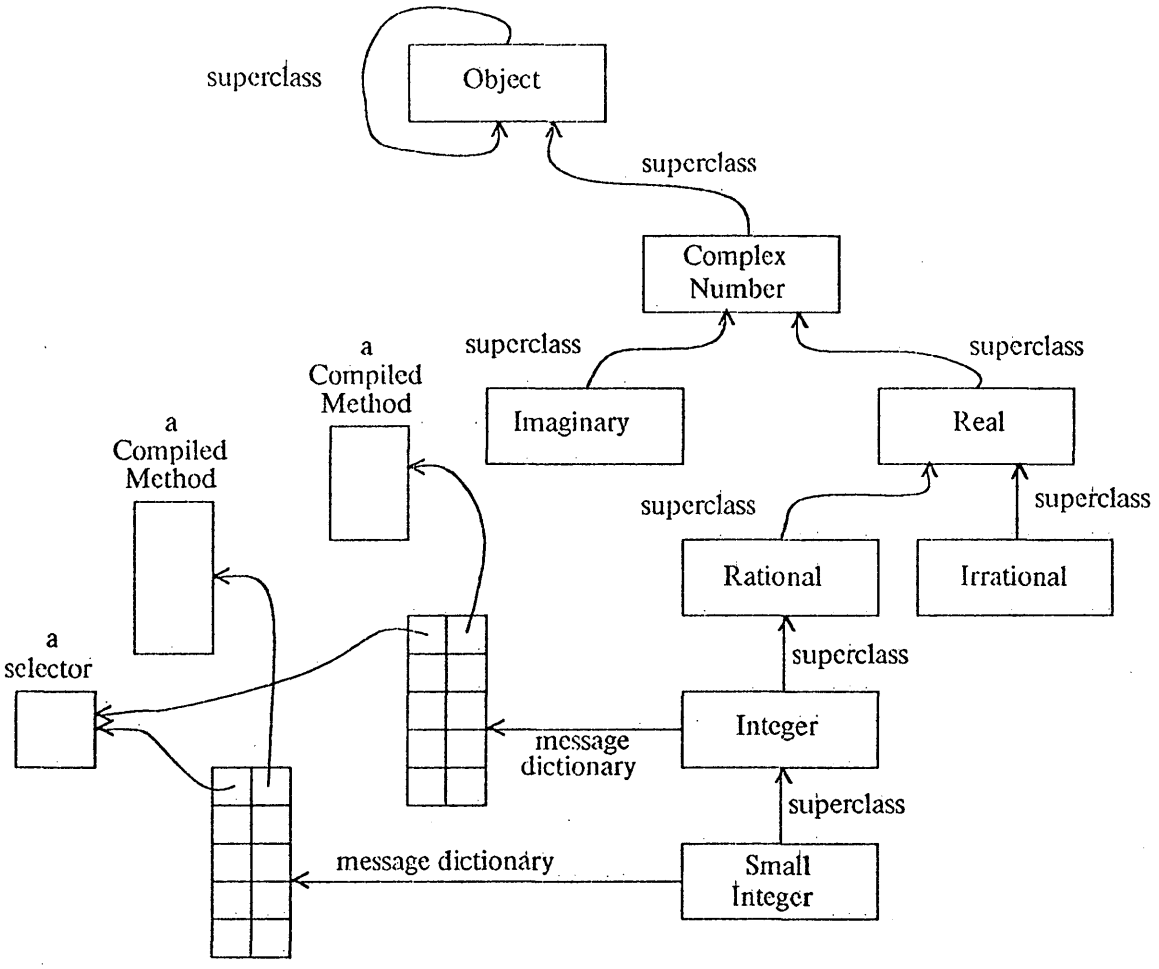


Figure 2-4 Inheritance Hierarchy Example

toward the root of the system (class Object) until either a method is found or the root is reached. If the latter event occurs, an error message is sent and the execution is suspended. Since the user may insert and delete message dictionary entries at run time, the binding between a selector and a method cannot be established before the particular message is sent.

# 3. Virtual Machine Emulation

Precise empirical comparisons of the paging performance of an object-oriented and a page-oriented virtual memory require a reproducible sequence of memory references. The inability to exactly duplicate the timing and nature of user input derived from the keyboard and mouse prevent the direct use of two Smalltalk implementations with different virtual memory configurations. Such a testing environment would have required the construction of two virtual memories. Implementing a paged virtual memory that dealt with 32-bit references would have necessitated a complete restructuring of the existing Smalltalk-80 virtual machine which deals exclusively with 16-bit pointers. For these reasons, a decision was made to use a fixed set of inputs and to simulate both virtual memories. This option required substantially less implementation and debugging effort and provided flexibility at the cost of the degradation in performance associated with simulations. Since any set of memory references is independent of both the original layout of objects in the virtual memory and the type of virtual memory, objects could be grouped by any static algorithm and policies and parameters of the virtual memory could be modified without invalidating the input trace.

## 3.1 Empirical Data versus Mathematical Models

Data to drive virtual memory simulations may be obtained from mathematical models of program behavior or actual execution traces. Analytic techniques have compact, flexible representations, low construction and usage costs, and the ability to reproduce an output sequence. Such models also tend to be mathematically tractable and convenient by making simplifying assumptions that in many cases do not accurately reflect the true operation of the program [BATS76, SALT]. However, without model calibration and validation, any results derived from such a model are only of theoretic interest [SPIR].

Alternatively, an *execution trace* or *event trace* may be extracted from a running system and used to drive a virtual memory simulation. This information is a time-ordered collection of events whose occurrence is (or may be) relevant to a concurrent or future simulation. Advantages of an event trace include reproducibility and accuracy. Offsetting these benefits are the problems of excess detail, lack of flexibility, and substantial difficulties in generating and employing the data.

Event traces were selected over mathematical models for three reasons:

> There is no generally accepted procedure for validating program models. Ferrari [FERR78] has summarized the current set of tools as ad hoc methods, partial tests, and common sense.
>
> No empirical measurements have been reported for a single-user, display-oriented, object-based, interactive programming environment such as Smalltalk. This lack of data would have prohibited any serious attempts at model validation.

Finally, the analysis of a detailed reference trace would yield statistical data for an object-oriented system that may be compared with similar information derived from programs written in other languages and executing in vastly different run-time environments.

### 3.2 Emulation of Smalltalk-80

Two methods of obtaining event traces are to modify the existing microcoded virtual machine and to construct an emulator that simulates the exact behavior of the virtual machine. Changing the existing microcode has a number of benefits. Most of the design, implementation, and debugging is already accomplished. The one remaining task, augmenting a running system, would cause only a slight degradation in performance. Smalltalk would remain a useful programming environment. Timing characteristics of an interactive session or a connection with a remote server would remain well within the ranges established by previous releases of Smalltalk running on lower-performance, personal computers. In contrast, an emulator written in Smalltalk would execute instructions orders of magnitude more slowly and thereby prevent normal patterns of usage from occurring.

Changing the virtual machine also has a number of unattractive features. Space limitations in the microcode store would tend to restrict the class of data compression techniques available, while the low level of the microcode language would inhibit program debugging, modification, and maintenance. Smalltalk, on the other hand, would provide all the advantages of a high level language, the simulation facilities found in an object-oriented programming environment, and more computational power by essentially removing the restrictions on program size.

I chose neither extreme. I wrote the bytecode interpreter of the emulator in Smalltalk, but used the virtual machine for its storage management facilities and execution of the low-level messages called *primitives*. This configuration supported the full generality of a program written in a high level language, allowed the use of a good deal of existing Smalltalk code, and provided some of the speed of the actual machine. Once constructed, the emulator was readily adapted to making measurements quite different from those initially envisioned. The emulator has become a general tool for obtaining statistics and has been integrated into a new display-oriented, interactive debugger with a single-step capability. Except for the tremendous cost in speed, the emulator has been invaluable for both its intended purpose and new applications.

### 3.3 Execution Trace Validity

Primitive methods are written in microcode and serve two purposes. By providing the basic arithmetic and logic functions, primitives end the infinite regress of message sending. Primitives also enhance performance, since a number of heavily used methods are written in microcode and thus execute rapidly.

Restricting emulation to the bytecode interpreter meant that the actions of the microcode when performing primitives had to be integrated into the event trace. In order to avoid compromising the validity of the statistics derived from the traces, a small subset of the primitives (about one-third)

were chosen according to usage frequency. For these primitives, the emulator recorded the memory references made by the microcode.

When a primitive method is encountered, the emulator instructs the virtual machine to attempt to perform the necessary actions. If the primitive succeeds, the microcode has performed the desired actions. The corresponding method is not executed and there is no new stack frame. A primitive fails when a situation occurs that the microcode is not prepared to handle. For example, one or more of the arguments in the message may not be of the expected type or within the appropriate value range. When a primitive fails, the corresponding method is executed by pushing another frame on the stack and setting the program counter to the first bytecode in the new method. For both successful and unsuccessful primitives, the emulator determined and recorded the set of references made by the microcode.

Logical functions, integer arithmetic, and comparison of two object identifiers are examples of operations that only access the top element(s) of the stack. These heavily used primitives do not contribute any references to object space. Determining the references made by the microcode is straightforward for most of the remaining operations. For example, array and string subscripting, object creation, and stream accessing are all characterized by a fixed set of references made to the receiver of the message and the arguments. Only one primitive required an extensive algorithm to compute the references made by the microcode. BitBlt, an acronym for bit boundary block transfer, is the primitive that manipulates, clips, and transfers areas between bitmaps. A complete description may be found in [INGA81]. Between 80% and 95% of all calls on primitive methods were detected and compensated for in this manner. Since primitives are a fraction of all methods and send instructions are a subset of all bytecodes, the missing references are not substantial. The event traces may be considered both representative and complete.

# 4. Detailed Reference Behavior

A number of measurements of a running Smalltalk system were taken and subsequently analyzed in order to investigate the basic reference tendencies of an object-oriented system. Results of the analysis were used to suggest, evaluate, and compare *static* grouping algorithms and had implications for cache management, swapping granularity, and event trace compression. Dynamic characteristics, such as the distribution of object lifetimes and the distribution of pointer distances (where the measure is the difference in creation time), are discussed in Chapter 10.

Although data was obtained from a wide variety of computations, one important result was the high degree of similarity between the measurements for different execution sequences. These statistics report characteristics of the Smalltalk-80 system rather than the peculiarities of the particular computations. Another conclusion drawn from the analysis is that the quiescent characteristics, measured when the system is not active, are a valuable but not infallible predictor of their run-time counterparts.

Each distribution compiled from these measurements was characterized by seven numbers: the median, mean, minimum, maximum, two quartiles, and the standard deviation. The entire collection of statistics is presented in Appendix A.

## 4.1 Definitions

Let an *object reference* be the reading or the writing of one field of any object except a stack frame done strictly for computational purposes. In this chapter only, all references made for storage management purposes are neglected. For example, all operations concerning reference counts, garbage collection, compaction, indirection through forwarding markers, contraction, expansion, purging, and faulting are not considered. This definition isolates the set of object references that are essential to the computation and that would occur if core were unlimited and the storage manager did not exist.

Define a *reference string* (*stream, trace*) to be a time-ordered collection of object references. In most virtual memory studies, reference strings are sequences of virtual memory addresses or virtual page references. Our object-oriented notion abstracts from the particular organization of objects in memory and is a more fundamental characterization of the accessing tendencies of the virtual machine. Batson [BATS76] and Ferrari [FERR76] have called such high-level information *symbolic*.

Even though a code fragment in Smalltalk is represented by an object that responds to messages as does any data object, access patterns are different. Methods are referenced in a highly regular manner (i.e., sequential bytecodes occasionally interrupted by jumps, returns, literals, and method/block activations), while most data objects are not. Most memory systems and previous

studies, however, do not make a distinction between data and code in reference strings and/or working sets. Spirn and Denning [SPIR] have identified this partitioning as an important research topic.

In the statistics presented in this chapter, a distinction is made between code and data in an attempt to separate the accessing patterns of each and determine the relative influence each exerts on the total picture. A somewhat arbitrary (but strictly enforced) decision was made partitioning objects into the two sets: *code* and *data*. Code objects are CompiledMethods and message dictionaries. Message dictionaries, which contain the mapping from message selectors to CompiledMethods, are implemented as conventional hash tables in order to support efficient lookups when messages are sent. While not actually executable code, these dictionaries are so closely associated with the CompiledMethods that a decision was made to include them in the code set. Except when source code is compiled or some CompiledMethods are discarded, the code set represents a static, read-only structure. Objects not present in the code set are by definition in the data set.

Although the 32K-field bitmap that supports the display is an actual Smalltalk object, it is considered to be a permanent resident of primary memory as is any other interface to an I/O device. For our purposes, the bitmap is not an entity that may be swapped in and out of core. Unless the contrary is explicitly stated, the 32K bitmap will not be considered hereafter.

In order to compare run-time characteristics with those corresponding to a quiescent state, three types of distributions were computed for most of the statistics: static, dynamic, and quasi-static. A *static* distribution is the distribution of counts of *objects in existence* whose attribute is a given value. A *dynamic* distribution is associated with an event trace and is the distribution of counts of *references to objects* in that trace whose attribute is a given value. A *quasi-static* distribution is associated with an event trace and is the distribution of counts of *objects used in the trace* whose attribute is a given value. I chose the word "quasi-static" in order to avoid the confusing term *average* used by Batson and Brundage [BATS77].

In a lengthy computation involving many cycles of purging and fetching, the dynamic weightings may be indicative of the set of older objects in core that survive many purge attempts. One shortcoming of this interpretation is the possibility that most or all accesses to an object were made during a very short time span and were not uniformly spread throughout the computation. The total ordering defined by the number of times an object was accessed may or may not be correlated with the probability of avoiding being purged. Quasi-static distributions offer useful comparisons with both the static and dynamic cases. If primary memory is initially empty at the start of the computation and objects are swapped instead of pages, the quasi-static distributions reflect the contents of core before any objects have been purged.

## 4.2 Trace Data

Five radically different execution sequences provided a representative set of statistics that could be used to investigate the (non)uniformity of the reference patterns in an object-oriented programming

environment. These traces would either highlight the differences or underscore the similarities between small windows on the reference stream generated by a typical user session. Compiling, entering the debugger, browsing through source code, editing text, and displaying characters in a window comprise the chosen operations.

In all graphs, charts, and text, the symbol *K* is used as an abbreviation for the integer 1024. All traces were exactly 144K references in length. Except for one case in which only 13K bytecodes were executed, between 31K and 39K instructions were interpreted by the emulator for each trace. The one exception represents the potential of the BitBlt primitive to require enormous quantities of references in order to accomplish a single instruction. Discounting the presence of the bitmap display object of size 32K, the span of the set of the objects touched ranged from 14K to 21K fields. Between 422 and 1240 distinct objects were touched in each trace, of which 85 to 393 were dynamically created.

### 4.3 Object Size

The distribution of object sizes is useful when deciding whether grouping is applicable and whether different grouping strategies are likely to have different effects on paging performance. Both questions were answered affirmatively.

*Size* is defined to be the number of fields of an object. Each object contains two or more fields. *Length* and *class* fields are present in all objects and require one word each. The size and core requirements are numerically equal for objects with word fields. For classes with byte fields, the space in primary memory occupied by an instance of size n is $\lceil n/2 \rceil + 1$.

Since the BitBlt operation has a potential to cause many thousands of references to a 32K bitmap, the dynamic size results are presented twice. One set contains all references; the presence of the bitmap is clear in all four traces in which it appears. A second data set ignores references to bitmaps and instead concentrates on the remaining objects, each of which has a size of at most 2K fields.

Objects tend to be small, with data objects smaller than code objects. In all three object partitions, dynamic size measurements are larger than both their static and quasi-static counterparts (Figure 4-1). The quasi-static sizes roughly equal (and generally slightly exceed) the static figures. As far as size is concerned, the computations involved typical objects. Substantially larger dynamic sizes imply that the distribution of references is not uniform but correlated with size. A more thorough analysis of the dependency is presented in section 4.7.2.

Of primary importance is the fact that objects tend to be much smaller than disk pages. A static median (mean) of 10 (19) fields, coupled with a 2:1 expansion of pointers for very large address spaces and 512-byte pages, means that one can expect roughly 6-13 objects to be placed on the same disk page. Since many of the byte fields will remain bytes, the factor of two is a strict upper bound on the expansion of objects transferred from core to disk. Although most objects are

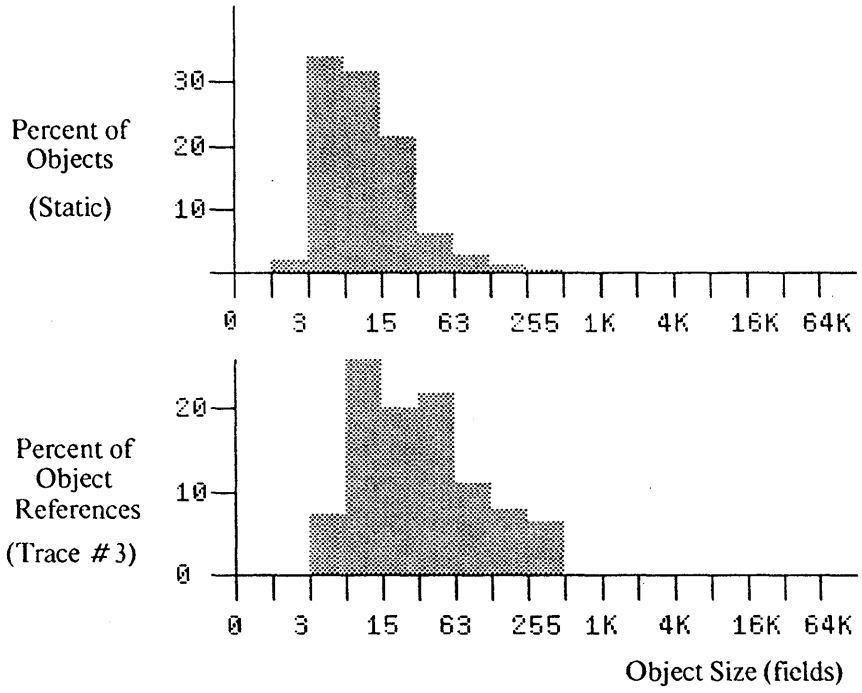


Figure 4-1 Object Size

smaller than a page, hundreds of objects are not able to fit on a page. If this were the case, then the distinction between any pair of reasonable grouping algorithms that traverse the directed graph composed of objects and pointers would be blurred. Smalltalk-80's ratio of page length to object size suggests that pages are the appropriate size for grouping objects.

In an empirical study of Algol programs, Batson and Brundage [BATS77] reported size distributions of arrays, program segments, and contour data segments. Mean sizes were larger than median sizes in both the Algol and Smalltalk size distributions. However, the dynamic statistics did not dominate their quasi-static counterparts in Algol as they did in Smalltalk. In Tables 4.1 and 4.2, the Algol measurements are expressed in words while the Smalltalk figures are in fields.

| | Dynamic Mean | Dynamic Median | Quasi-Static Mean | Quasi-Static Median |
|---|---|---|---|---|
| Program Segment Size | 38.8 | 23 | 93.1 | 33 |
| Array Segment Size | 343.1 | 16 | 616.6 | 30 |
| Contour Data Segment Size | 17.9 | 9 | 13.2 | 6 |
| Total Memory/Contour | 28.1 | 9 | 724.8 | 7 |

Table 4.1 Segment Sizes in Algol Programs

| | Dynamic Mean | Dynamic Median | Quasi-Static Mean | Quasi-Static Median | Static Mean | Static Median |
|---|---|---|---|---|---|---|
| Code Objects | 64-103 | 34-41 | 32-40 | 16-20 | 31 | 17 |
| Data Objects | 19-62 | 10-10 | 9-18 | 4-10 | 14 | 8 |
| All Objects | 51-80 | 27-30 | 16-26 | 9-13 | 19 | 10 |

Table 4.2 Object Sizes in Smalltalk Computations

Earlier empirical work by Batson et al. [BATS70] also found surprisingly large numbers of small segments in a university computing environment. Sixty percent of the in-use segments were smaller than 40 words. McKeeman [McKEE] measured a mean program segment size of 60 words in a collection of compiled scientific programs. Snyder [SNYDa], on the other hand, reported 3 as the average size of dynamically created objects in a small number of CLU programs.

One final remark is the visibility of well-known classes of objects in the size distributions. The popularity of 10 for quartiles and means in the size statistics is due to class objects. Class objects are frequently accessed, since they are touched at least once every time a message is sent or a new object is created. Message dictionaries, which have a capacity slightly larger than some integral power of two, caused spikes just beyond 64, 128, and 256.

### 4.4 Fractional Utilization

Memory may be underutilized because of *fragmentation*, which is caused by filling memory with unneeded information [KUCK]. Randell [RAND] has partitioned this problem into *internal* and *external fragmentation* ("checkerboarding"). Internal fragmentation corresponds to the storage wasted by rounding up the size of each object to the nearest multiple of the smallest memory allocation unit. Except for wasting one byte in the case of objects with an odd number of byte

fields, LOOM avoids this problem by swapping objects instead of pages. External fragmentation occurs when non-relocatable objects prevent a memory allocation request from being satisfied even though the total amount of free storage can accommodate the request. LOOM attempts to avoid this type of memory underutilization by compacting in-core objects when a memory allocation request fails. If the size of the free block created by the compaction routine does not exceed some threshold, then purging occurs before compaction is again attempted.

Memory may also be wasted by information that is potentially useful but happens not to be referenced while the object/page is in core. The presence of such information in core has been called *superfluity* [KUCK] and *temporal fragmentation* [MORR]. From empirically derived functions mapping page size to the minimum number of pages required to perform a given computation [O'NEI, BELA69], superfluity has been determined to be roughly proportional to the page size [KUCK]. However, the author knows of no published reports that examine superfluity in logical objects as opposed to physical pages.

Define the *fractional utilization* of an object used during a computation as the percentage of fields of the object that were touched (read or written), neglecting any initialization for dynamically created objects. All fields of objects are considered, even self-descriptive aspects such as the length and class fields. This definition formalizes the *density of reference* notion discussed by Morrison [MORR].

### *4.4.1 Discussion*

Fractional utilization distributions examine memory usage and may be used to evaluate proposed units of memory allocation and information swapping. Even though the computations did not reference all the fields in all objects touched at least once, the average usage was high enough to dictate against pure field swapping or hybrid schemes that swap both fields and objects. No firm conclusions could be drawn concerning the correlation between object size and fractional utilization. An incremental analysis indicated the observed usage levels approximated equilibrium values and were not highly dependent upon the length of the trace.

All objects used by the system, except for message dictionaries, were included in the following analysis. Heavy use of hash tables in the form of message dictionaries, arbitrary load factors, and ambiguity in defining appropriate fractional usage levels led to the exclusion of such hash tables from the fractional utilization statistics.

Quasi-static measurements are the natural choice for fractional utilization results, which were initially tabulated for the entire length of each trace (144K references). Section 4.4.2 deals with shorter window sizes. On the average, only 39% to 51% of the fields of an object were accessed during the five computations. Data objects spanned the entire range from 0% to 100%, while the code objects did not. Objects with a zero percent value were newly created entities that had not yet been accessed. No code object showed a complete utilization. Three trailing bytes in each CompiledMethod serve as hints for the location of the source code and play no part in the

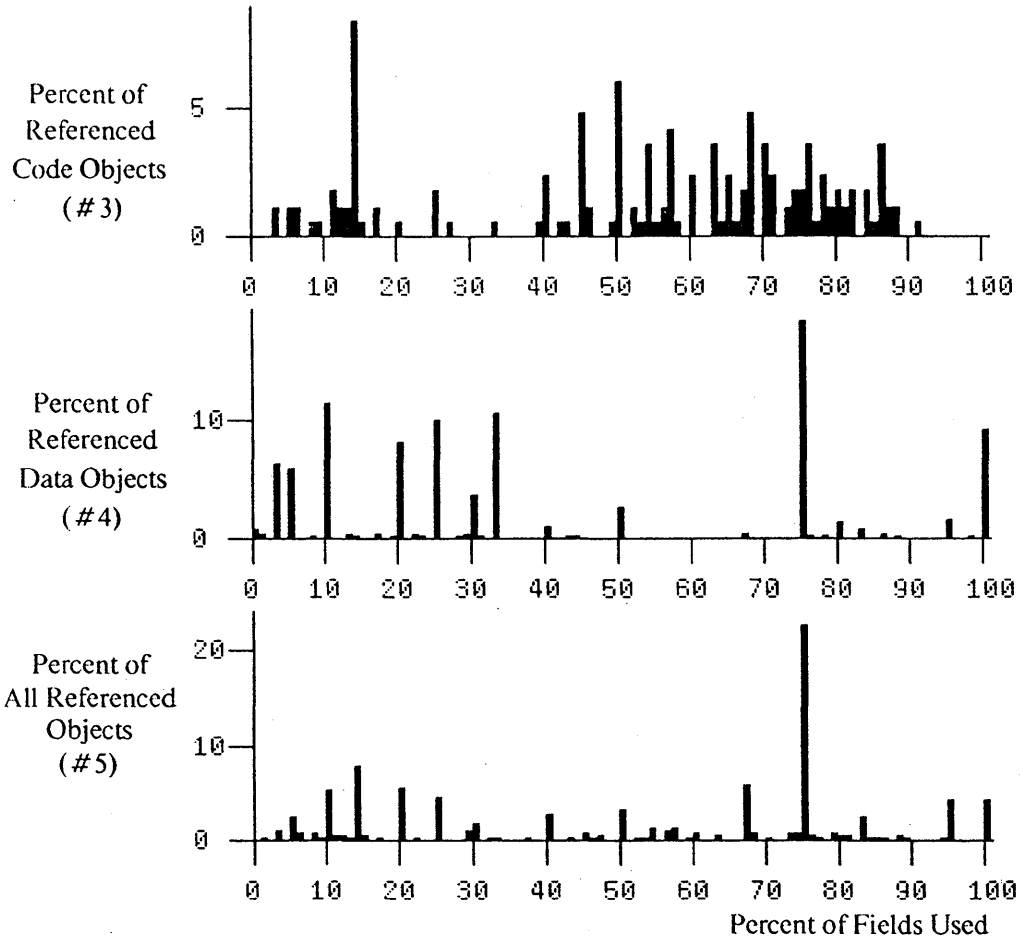


Figure 4-2 Fractional Utilization

execution of the method. Since the length and class fields of CompiledMethods are also infrequently accessed, not all fields in a CompiledMethod are typically used.

Two interesting spikes in the fractional usage plots were values at 75% for data objects and 14% for code objects (Figure 4-2). Instances of class *Point*, which have 4 fields, account for the bulk of the first anomaly. Three fields, the x, y, and class fields, were heavily used. The length field of a point, which is only needed for storage management purposes, was not explicitly used during the computation. Quick methods, which have no bytecodes but instead contain the number of a field in the receiver that is the returned value of the method, have only 7 fields. Only one of these seven fields is touched by the microcode. These quick methods, which are not shared between classes, contributed heavily to the large value that occurred at 14% for code objects.

Except for the spike at 14%, most methods had 40% to 90% of their fields touched. Conditional branches, early returns, and iterators operating over empty sets of data prevented all bytecodes from being executed and all literals from being needed.

Data objects, on the other hand, had a much larger spread in their fractional usage values. In all five cases the standard deviation for data objects exceeded the corresponding number for code objects. There was a striking difference between code and data for the first quartile and median in the first four reference traces. For example, the median fractional usage for data varied from 25% to 33%, while for code it varied from 51% to 57%. Ranges for the first quartile were 5% to 20% and 15% to 40%, respectively. However, the intervals were nearly equivalent by the third quartile: 67%-75% and 68%-73%. The fifth trace is anomalous because data fractions exceeded code fractions at both quartiles as well as the median. The impact of the large quantity of point objects (at least 25% of all data objects in the fifth trace) skewed the last distribution and also significantly raised the third quartile in the other four cases.

Fractional utilization is important since the apparent size of memory may be viewed as the product of the physical size and the utilization rate. If the utilization of core is low, then alternate swapping strategies based upon smaller structures, such as fields, may increase the apparent size of core by increasing the fraction of core used. As the unit of swapping gets smaller, however, there is typically more required overhead in time and/or space. This requirement is due to the larger number of units of data in core at one instant. More time, space, or hardware is required to determine if a particular unit of information is immediately available or not.

Other problems are aggravated when the unit of swapping is increased. For example, as the granularity of swapping gets coarser, the fractional usage tends to fall. Usually only a subset of the objects on a page are touched during the time the page remains in core. The same also holds true for the fields of an object. When the fields in a page are considered, the two effects are multiplicative. Utilization can get no larger when coarser structures are swapped, since any configuration in the coarser system may be duplicated in the finer system. Offsetting this advantage, however, is the fact that finer systems require much more overhead to manage a larger set of swapping units and may increase the likelihood of thrashing.

### *4.4.2 Incremental Analysis*

In the preceding analysis, a fixed window size of 144K references was employed. On the average, only half of the information in an object was used during this time span. Larger primary memories, which increase the average residence time for objects, enlarge this window and should shift the distribution of fractional usage to the right. Longer periods of time will increase the probability that formerly unused fields will be touched. Narrower time slots correspond to smaller core sizes and should tend to push the distribution to the left for the obvious reasons. In general, the measured utilization level will be a function of the window size.

For very small windows, the distribution may shift in apparently the wrong direction. This anomaly would be due to the influence of the relatively large number of (newly created) objects touched for the first time in the computation. Inclusion of these objects by using larger time slots would shift the distribution to the left. For smaller time slots, their absence would translate the function to the right. A close examination of the fractional utilization of objects as a function of window size supports the original set of intuitive hypotheses.

Each trace was subsequently analyzed using only the first 1K, 2K, 4K, 8K, 16K, 32K, or 64K references. The quartiles, medians, and means of the fractional usage distribution rapidly converged to the values corresponding to the full 144K trace (Figure 4-3). This convergence was normally asymptotic from below, but small oscillations above and below the final value occurred. One remarkable fact was the relative invariance of the general shape of the distribution. Spikes in the graph appeared as the analyzed portion of the traces became longer, yet their only apparent effect was a proportional reduction in the value of the function at all the other points of the domain. While the complete data set may be found in Appendix A, the examples in Table 4.3 are indicative.

| Trace Length | 1K | 2K | 4K | 8K | 16K | 32K | 64K | 144K |
|---|---|---|---|---|---|---|---|---|
| All (Median #5) | 20 | 20 | 20 | 25 | 30 | 50 | 54 | 59 |
| Code (Median #5) | 44 | 44 | 40 | 40 | 40 | 43 | 45 | 45 |
| Data (Median #5) | 10 | 10 | 20 | 25 | 25 | 67 | 67 | 75 |
| All (Median #3) | 25 | 25 | 25 | 30 | 37 | 39 | 40 | 29 |
| Code (Median #3) | 39 | 45 | 50 | 54 | 52 | 54 | 54 | 57 |
| Data (Median #3) | 20 | 25 | 20 | 25 | 25 | 25 | 28 | 25 |
| All (Mean #5) | 23 | 23 | 29 | 36 | 41 | 47 | 49 | 51 |
| Code (Mean #5) | 39 | 39 | 41 | 40 | 40 | 41 | 42 | 42 |
| Data (Mean #5) | 15 | 16 | 22 | 33 | 41 | 51 | 53 | 56 |
| All (Mean #3) | 31 | 33 | 36 | 39 | 42 | 43 | 44 | 39 |
| Code (Mean #3) | 37 | 37 | 44 | 48 | 47 | 49 | 51 | 53 |
| Data (Mean #3) | 26 | 30 | 31 | 35 | 40 | 41 | 41 | 35 |

Table 4.3 Selected Fractional Utilization Values

The fractional utilization function for code objects may be closely approximated by examination of a very short reference trace. Data from the 16K or 32K window sizes are remarkably close to their values for a 144K window. Except for sending messages (subroutine calls), CompiledMethods are

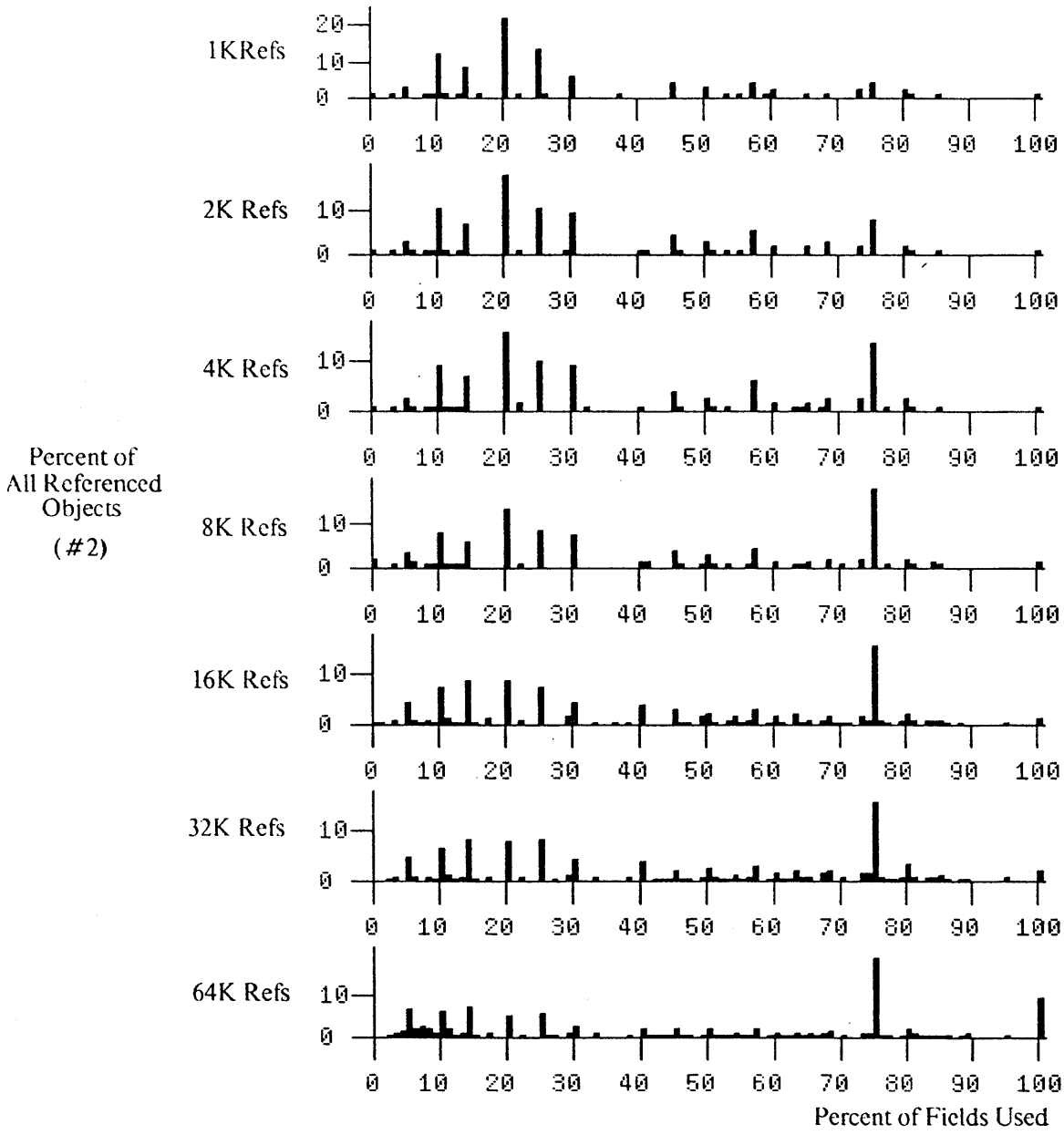


Figure 4-3 Incremental Fractional Utilization

heavily accessed during a short time period and not touched again until a later invocation. The execution of a small number of methods yielded enough data to derive reasonable approximations.

Recently touched objects, on the other hand, usually receive a number of messages in a short period of time. The span of time during which an object is used ranges from one to dozens of method executions. Apparently the short time periods used were not long enough to allow a true equilibrium to establish. Newly created objects and objects touched for the first time tended to reduce the overall fractional utilization. However, the rate of change of the utilization value with respect to the difference in trace length decreased with longer traces.

Comparisons of similar statistics for different traces having the same length indicate only a slight predictive tendency for the final ordering. Results for the 1K windows cannot be used to accurately predict the high to low ordering of the 144K windows. The motion of the spread of the values corresponding to a particular fractional usage statistic was generally monotonically increasing. For example, the endpoints for the band defined by the largest and smallest medians for a particular trace length both grew as the trace length was increased. Although the size of this band fluctuated, its midpoint never decreased. Table 4.4 illustrates this tendency.

| Trace | 1K | 4K | 16K | 64K | 144K |
|---|---|---|---|---|---|
| #1 (Code) | 37 | 46 | 50 | 54 | 54 |
| #2 (Code) | 45 | 50 | 50 | 50 | 51 |
| #3 (Code) | 39 | 50 | 52 | 54 | 57 |
| #4 (Code) | 38 | 45 | 54 | 54 | 54 |
| #5 (Code) | 44 | 40 | 40 | 45 | 45 |
| Range | 37-45 | 40-50 | 40-54 | 45-54 | 45-57 |
| MidPoint | 41 | 45 | 47 | 49.5 | 51 |
| | | | | | |
| #1 (Data) | 20 | 25 | 25 | 30 | 33 |
| #2 (Data) | 20 | 25 | 25 | 25 | 33 |
| #3 (Data) | 20 | 20 | 25 | 28 | 25 |
| #4 (Data) | 20 | 25 | 30 | 30 | 30 |
| #5 (Data) | 10 | 20 | 25 | 67 | 75 |
| Range | 10-20 | 20-25 | 25-30 | 28-67 | 25-75 |
| MidPoint | 15 | 22.5 | 27.5 | 47.5 | 50 |

Table 4.4 Median Fractional Utilization Values for Varying Window Sizes

### *4.4.3 Effect of Object Size*

A second question raised by the initial analysis is the dependence of the fractional utilization value on object size. If the fractional utilization of objects increases with size, object swapping has an advantage over field swapping for most objects larger than some size. The success of an object-swapping scheme depends on the size distribution of objects swapped into primary memory.

A negative slope on the fractional utilization versus object size function implies the opposite. One solution to the underutilization of core by large objects is to use a hybrid scheme. Objects smaller than some threshold where average utilization falls off are swapped, while larger objects have their

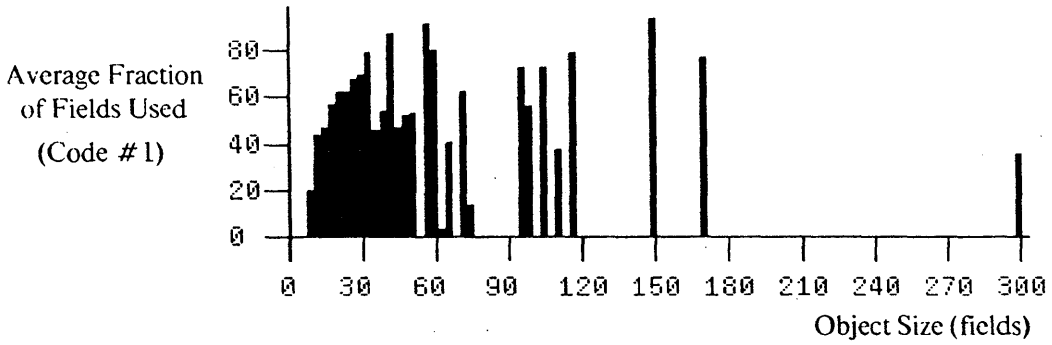


Figure 4-4 Fractional Utilization versus Size

fields swapped only. Performance of such hybrid schemes depends upon the variance of the utilization function as well as the choice of the cutoff level. An alternate solution is to use a per-object indicator (hint) to decide whether to swap the entire object or just the desired field(s). Hints could be static information derived from a number of typical execution sequences. This knowledge could also be updated dynamically, depending on the number of fields accessed the previous time the object or any of its fields were in core.

A flat distribution implies that the fractional utilization is relatively independent of object size. Depending on the actual fractional usage value, pure field swapping may or may not be a better memory management policy than pure object swapping.

For code objects smaller than thirty fields, there was a marked increase in the average utilization as size increased. Much of this difference is directly attributable to the five infrequently used fields of the CompiledMethod, since the remaining fields were heavily utilized. The importance of these fields is inversely proportional to method size and is evident in Figure 4-4. Beyond this simple observation, no firm conclusions may be drawn for large code objects because of the sparseness of the fractional utilization distributions for large object sizes.

For data objects, there was a slight tendency for the fractional utilization to decrease with increasing object size. However, the extremely high utilization of a small number of objects with sizes in the 30-70 range argues against using a hybrid scheme with a sharp division between swapping objects and fields. The wide variation in usage for objects with size less than forty was also disturbing in that pure object swapping or pure field swapping may fail miserably under certain circumstances.

### *4.4.4 Conclusions*

Object swapping may be viewed as an attempt to increase the utilization of core memory by releasing the constraints imposed by the static placement of objects on pages. Just as objects (or code segments) can be grouped on pages to increase the fraction of the page actually utilized, swapping fields or collections of fields on demand may increase the portion of the in-core pieces of the object that are used while in primary memory. The best possible scenario from the standpoint of the fractional utilization measure is a complete utilization of core coupled with the optimum (and hence unrealizable) field-purging policy. Attempting to maximize the fractional usage value by swapping fields on demand and using realizable purging algorithms may lead to extremely poor disk cache performance in addition to an unacceptable computational overhead for each reference. Thrashing and much of this overhead may be reduced or even avoided for both the loading and purging of data by the intelligent grouping of information in secondary memory and/or the swapping of larger units of information.

When a primitive method is handled by the microcode, only two fields of the CompiledMethod are actually accessed. Successful primitives and the five infrequently used fields biased the fractional utilization of code objects. Interpreting the fractional utilization distributions in light of this additional knowledge indicates that methods ought to be viewed as indivisible units and swapped in

their entirety. Analysis of the results from progressively longer traces have shown that the fractional statistics found for code objects represent equilibrium values.

Incremental analysis has indicated that more investigation needs to be done for data objects, since they are accessed differently than code objects both intuitively and empirically. Longer execution traces need to be analyzed in order to determine the plateau value and rise time of the utilization function for the data objects. The low utilization values observed may indicate the existence of a natural partitioning of large data objects. References occurring over a small period of time may or may not cluster about certain fields of an object. If this clustering does in fact occur, then an effective memory management scheme is to break large objects into blocks and swap blocks instead of fields or entire objects. A further consideration is the movement of the cluster as well as the invariance of the partition it defines.

One critical insight was provided by the analysis of the progressively longer traces. Some sequences of fractional usage values asymptotically approached their final (i.e., 144K) value from below. The slope of the curve fitted to these points had a positive slope that decreased with increasing trace length. Other sequences first rose and then oscillated before obtaining their final value for a 144K window. Such behavior indicates the presence of a dynamic equilibrium and the inability of a small window size to filter short-term behavior. Rapidly decreasing slopes for the functions of the first kind, fluctuations present in the functions of the second kind, and a general agreement between all traces for 144K values are the key pieces of evidence that most of the transient behavior had been overshadowed by an equilibrium state.

Many of the classical time-space analyses of algorithms and data structures lose their asymptotic relevance for programming environments with small core sizes. Instead, nearness of data in time and space and fractional utilization of memory become the important criteria for purposes of performance evaluation. The importance of fractional utilization decreases for smaller units of swapping but cannot be eliminated completely with conventional mass storage devices. This ideal can be attained only with a secondary memory that supports the efficient transfer of arbitrarily small units of data (fields). For example, if an in-core disk buffer is used and complete pages are swapped between this buffer and the disk, field swapping can substantially reduce but cannot completely eliminate the importance of the fractional utilization of both code and data objects. Fractional usage is by no means the only measure for comparing virtual memory designs. Factors such as pointer compression, computational overhead, paging performance, and implementation requirements also affect the choice of a virtual memory design.

## 4.5 Interreference Headway

Define the *interreference headway distance* of a reference to an object A as one plus the total number of references occurring between that reference and the next reference to A. For example, consecutive references to the same object yield a distance of one. When compiling statistics for data only, code references are ignored and vice versa. When statistics are obtained for all objects, both

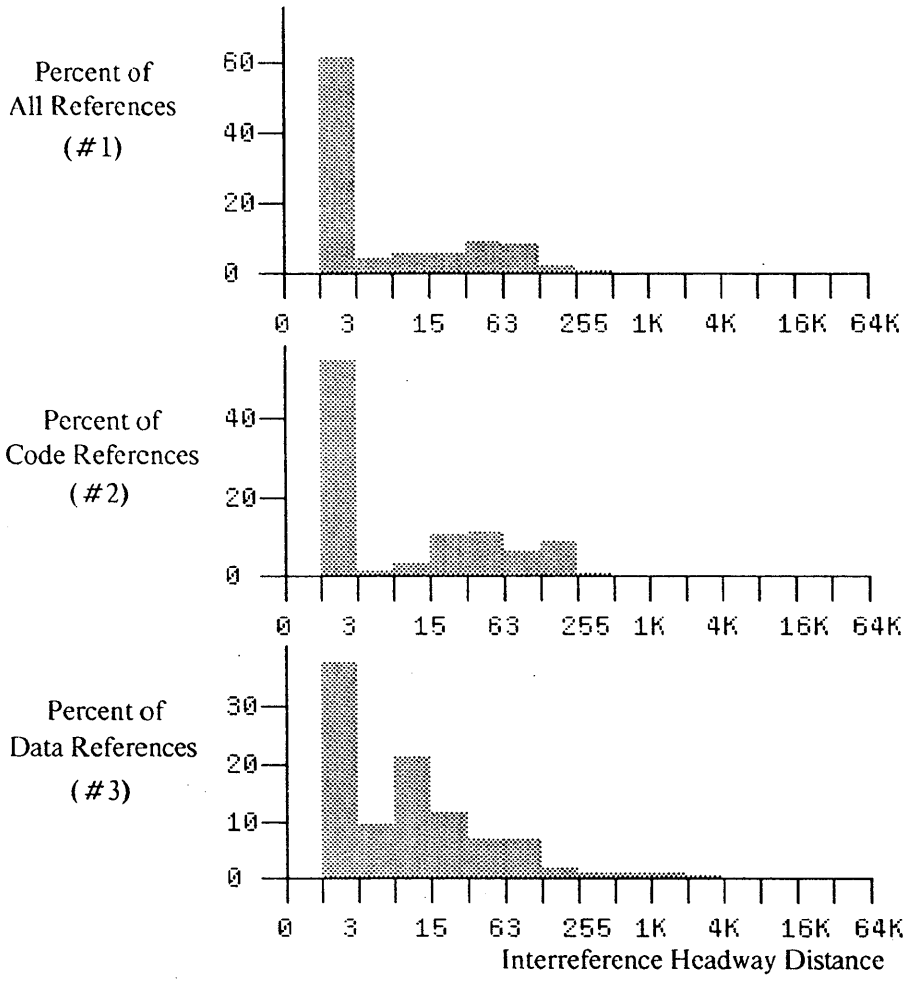


Figure 4-5 Interreference Headway

kinds of references are counted. This definition partitions references into data space and code space and allows investigation into either area without interference from the other.

Distributions of data interreference headways were remarkably similar to those for code references. Locality of reference was clearly present even in the composite reference traces and can be exploited at different levels of the memory system. These results also indicate that simple techniques could be used to substantially compress the reference stream without a great loss of information.

For all five traces, the median interreference headway for code objects was one (Figure 4-5). Except for looking up message selectors in hash tables, returning to the calling/home context, or interpreting a CompiledMethod in a new context, the next code reference was to the same CompiledMethod. Average headways far exceeded the median and even the third quartile. Distributions were skewed by a substantial number of large headways. These values arose from message lookup, since any number of bytecodes may have been executed before another message to an object of the same class (or subclass) was sent. Sending a message, performing the required operations, and returning a result also widely separated two accesses to a CompiledMethod. The portion of the distribution dealing with code headways greater than one is an approximate indication of the distribution of the time in code references required to send a message.

Data headways were likewise characterized by small numbers and highly skewed distributions. Medians ranged from one to eight, while the means were far larger than the third quartiles. The third quartiles and mean values of the headway distribution were lower for data than for code objects. One factor that contributed to this result is the inherent difference between the roles played by messages and the receivers (senders) of messages. Except for the case of (possibly indirect) recursion, a CompiledMethod is not accessed after sending a message until the appropriate computation completes. Normally, some of the objects local to the calling method are parameters to portions of this computation and are referenced before control returns to the original method. Conversely, the remaining local objects are not accessed during this execution sequence and contribute to the large headways.

Mean headways varied between 118 and 307. These figures are significantly larger than the average number of distinct pages between two references to the same page reported by Lewis and Yue [LEWI]. Their means, which varied from 1.2 to 2.2, were much smaller than the mean Smalltalk distances because *distinct pages* were counted instead of all intervening object references.

Composite headway distances are necessarily at least as large as the corresponding code or data results, because all references were included in the composite case. While the first quartile of the composite headway distribution was still only one, the median varied between one and seven. Locality in time and space was evident even in the composite traces.

Performance gains appear to be realizable over a rather large spectrum of memory configurations, from a tiny hardware cache handling dozens of objects to a conventional core memory handling thousands of objects. A large gain may be realized by caching a small number of objects. Since objects tend to be small, swapping small, fixed-size blocks between the cache and core will realize the gains secured from object swapping without most of the computational overhead. Fractional utilization results and the frequency of send instructions indicate that caching blocks of CompiledMethods rather than caching individual fields or the entire object would result in larger performance gains. Lower fractional utilization values for data objects imply that caching fields (or small groups of fields) would be better than caching entire data objects. Enlarging such a cache beyond the capability of containing a small number of objects would yield only moderate improvements in both hit rate and performance. These empirical results also indicate that LRU purging algorithms would be well-suited to the observed reference behavior.

Headways were calculated to explore the amount of locality present in the data-only, code-only, and composite reference streams. Distributions for code, data, and composite headways are markedly similar, exhibiting great locality and indicating that simple techniques can greatly compress the reference stream. Similarities between code and data reference tendencies indicate that neither cache management schemes, virtual memory managers, nor compression algorithms ought to distinguish these two sets of objects.

### 4.6 Instance to Class Compression

Counting the classes of the objects touched during a given computation instead of the particular instances referenced yields an estimate of the portion of the system that played a role in the execution sequence. Any static grouping algorithm that relies on information garnered from such traces will be extremely limited when it deals with instances of classes for which it has no information. Since two of the grouping algorithms presented in Chapter 5 depend on dynamic information, knowledge of the fraction of the system utilized by the monitored computations is important when evaluating the performance improvements realized by these two grouping schemes.

Call the fields owned by every instance of a class the *fixed fields* of that class. For each class, we create a prototypical instance that contains an entry for each fixed field that maintains the number of references made to the corresponding fixed field in all instances of the class. Sorting the reference totals will produce an ordering of fields for each class that ranks the fields in terms of usage, identifies the high traffic fields, and provides data to drive static grouping algorithms. The only information input to the grouping schemes dependent upon dynamic data was the ordering of the fixed fields in each class. This information guided the traversal of the system by the grouping algorithms by ordering the set of offspring referenced by the fixed fields of an object.

Let the *fixed fields extent* of a computation be the percentage of fields in the prototypical objects that would contain nonzero values at the end of the computation. Only a small percentage of all fixed fields was actually touched during any of the monitored execution sequences. Of the roughly 200 classes found in the Smalltalk-80 system, only a small subset had at least 90% of their fixed fields touched during a particular trace. Although there was some overlap between traces, these subsets were largely disjoint. Table 4.5 reports the number of such classes for each trace, as well as the accessed fraction of all fixed fields in all prototypical objects.

| Trace | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Number of Classes Above 90% | 11 | 11 | 13 | 7 | 9 |
| Percent of All Fixed Fields Touched | 11 | 11 | 6 | 8 | 9 |

Table 4.5 Trace Extent as Measured by All Fixed Fields

The above information is dynamic and highlights those fixed fields that were referenced most frequently. A similar but unused technique for obtaining data to drive static grouping algorithms is to record only quasi-static data by ordering the fields according to first-use statistics. For every object involved in a computation, consider only the first reference to any fixed field of that object. The prototypical object for each class could maintain the number of times that each fixed field was referenced first. Identification of such "quick-use" fields will allow grouping strategies to place objects referenced by these fields extremely close to the object that references them.

Static grouping algorithms may use information regarding the stability of fields containing object references. A field is said to be *clean* at the end of a computation if its contents were not written by the computation. Otherwise, the field is *dirty*. One possibility is to order clean fields by some rule and dirty fields by the same or some other rule. Because dirty fields were written at least once, they are potentially unstable. Therefore, the corresponding descendant need not be initially placed near the object in question, and the dirty fields can be placed after all the clean fields in the composite ordering.

Since the monitored computations involved only a small number of classes, dynamic information derived from the traces can have a limited impact only on the initial placement of objects by static grouping algorithms. In most cases, the dynamic data supplies no information for a class, so the grouping algorithm uses the default identity permutation. For short periods of time, much of the user's interaction with the programming environment is supported by a small number of classes. Perhaps monitoring a few key classes for a short duration will yield enough information to allow a static grouping algorithm to enhance substantially virtual memory performance.

Only the fixed portion of classes with variable-length instances were monitored, because reference statistics collected from relatively short computations would probably not be significant for the variable-length portions. In the static grouping algorithms, the fixed portion of an object was considered before the variable part. The permutation for the variable portion, which was always the

identity permutation, matches best with a sequential traversal. On the other hand, this type of static grouping is not particularly suited to nonsequential reference tendencies.

The success of fixed-field grouping schemes based upon dynamic information depends on the validity of two assumptions. In order for such a technique to improve performance, the usage of a field must be independent of the particular pointer present in the field as well as the particular instance of the class. If the reference patterns are predictable, repeatable, and independent of a particular instance of the class, then such a static grouping scheme will perform well. However, if the reference pattern is dependent upon both the specific instance of the class as well as the current contents of the instance, then no static scheme will be highly successful. Completely random inspection of fields is neither aided nor hindered by a fixed-field grouping strategy nor by any other realizable scheme.

The appropriate meaning of "usage" cannot be adequately expressed by a quasi-static or dynamic definition but instead depends on the virtual memory configuration, its policies, and the reference behavior of the computation. Let f be a field in object A capable of referencing any other object. Assume object A was fetched and purged s times and field f was touched at least once in t of these s times in core. Define the *usage* of f to be 100*t/s. This value is the probability that the field was referenced at least once during one fetch-purge cycle for the object. Our original questions may now be reformulated. Is the usage of a field a function of the contents of that field? If not, is the usage of a field a function of the state of the object?

Field usage is independent of field contents as long as the information contained in this field cannot be ascertained by inspecting other fields of the object. If objects are fetched on demand, an object is swapped into core only when it is faulted upon. A reasonable assumption is that little or no knowledge of the contents of the object exists in primary memory. In order to determine the contents of any field, the computation must explicitly touch that field. Except for contrived cases, field usage is independent of the contents of the field. Dynamic usage, on the other hand, has the potential to be extremely sensitive to the contents of the field.

The success of fixed-field grouping also depends on the independence of field usage with object state. Information is lost by the data compression technique that discards the particular instance referenced and records only its class. For many classes of objects, the state of the object is more important than its class in predicting the usage of its fields. Consider two possible implementations for a hash table. Instances of the first class are composed of an array of keys and a pointer to an array of values. The second implementation is just an array of pointers to key-value pairs. Usage of the fields in the key array in the first class and the only array in the second class is independent of the particular class (implementation) of the hash table object. It is highly dependent upon the nature of the add/delete/find requests and the current state of the hash table. The same dependency is even more obvious for the usage of the field containing a reference to the value array in the first implementation. Although the computation may not know the contents of the value array, the absence of the particular key in the key array implies that the desired value is also not

present. On the other hand, there are cases where the usage of a field is dependent only upon the class. Consider a hypothetical subclass of Array called SpecialArray. Instances of this class act as normal arrays but may take special actions. For example, during a debugging session, a programmer may wish to monitor the values stored into a specific field of a collection of arrays. Each of these arrays could be instances of SpecialArray. In addition to the usual fields of an array, there would also be a reference to an ordered sequence of values stored into the special location. Deciding whether to record a given store operation is independent of the state of the SpecialArray.

In summary, the usage of a fixed field is essentially independent of the contents of that field. The same conclusion, however, does not apply to the other fields of the object. There are cases where the state of the object is the dominant factor. Other examples indicate that the usage of a field is more a function of the class of the object. Most classes span a portion of this spectrum. Where individual instances lie depends upon their state, the current state of their class, and global values and variables, as well as the particular virtual memory configuration, its policies, and the computation in progress.

## 4.7 Access Frequency

Each reference trace was analyzed in order to determine the frequency of access. The *access frequency* function maps a number *n* into the fraction of objects that were referenced n times. This distribution generalizes the notion of the average number of times that an object was accessed and is important for the design and evaluation of data compression schemes. The following analysis indicates the existence of a small set of key objects that are heavily utilized. However, there seems to be no need to extend preferential treatment to members of this set. A naive, demand-driven cache management scheme would suffice.

### *4.7.1 Discussion*

The access frequency functions for data objects were for the most part decreasing, while the code functions generally increased to a maximum and then fell with a similar slope (Figure 4-6). Code objects on the average were touched more frequently than data objects. Unlike code objects, most data objects were touched a small number of times in a computation. Except for quick methods and primitives handled by the microcode, if a code object were touched once, there was a high probability that it would be needed a large number of times. There is an inherent asymmetry between code and data contained in (but not peculiar to) the Smalltalk-80 virtual machine. Each bytecode executed requires at least one reference to the method. Extended bytecodes require two or three such accesses. However, many bytecodes and primitives do not touch object space at all, while a majority of the remaining operations touch one or two objects a small number of times. A comparison of the ranges defined by the means and the medians of the access frequency distribution highlights this difference between code and data. For code objects, the medians ranged from 24 to 94 while the means fell into the interval from 181 to 355. Corresponding extrema for data objects were 6 and 17 for the median and 61 to 183 for the mean, neglecting the 32K bitmap in the fifth trace.

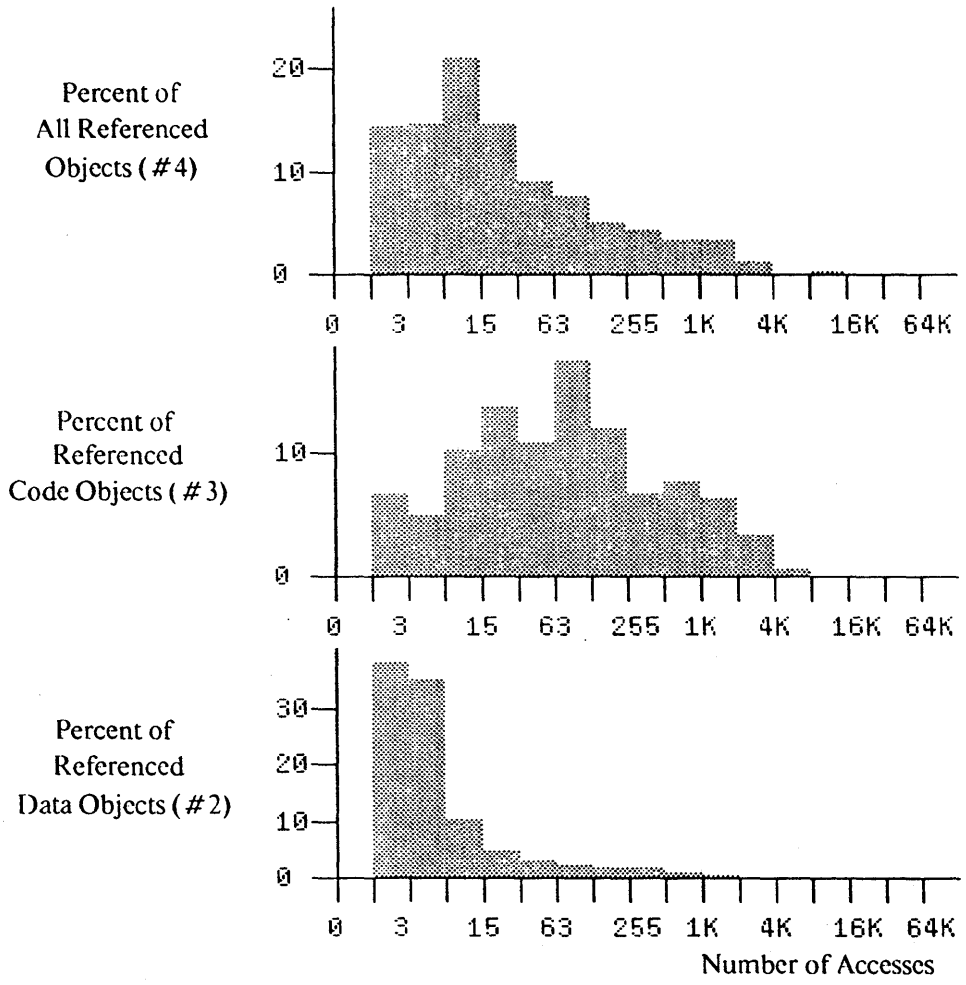


Figure 4-6 Access Frequency

Except for code objects in trace five, the average number of times an object was touched is far larger than the corresponding value at the third quartile. A small number of objects accounted for a large percentage of the accesses. This claim is supported by two other observations. In four of the five traces, 25% of the data objects were touched at most three times. Moreover, more than one-fourth of the data objects in the second trace were accessed at most once.

### *4.7.2 Effect of Object Size*

Consider a function that maps an object size into the average number of references made to objects of that particular size (Figure 4-7). Call this function the *importance* curve, since it exhibits the level of need for data in objects of a certain size. The component-wise multiplication of the importance curve with the quasi-static size function for the same trace yields the dynamic size distribution for that execution sequence. No new information is being provided by the importance function. It is simply a different view of the same data. However, this alternate view proved to be beneficial by indicating the absence of any consistent correlation between object size and access rate.

A flat importance function implies that the probability of accessing an object from a given set of recently used objects does not depend on the size of the object. If objects were not touched a great deal, then this shape of the importance curve is plausible. Small objects would have their fields touched many times.. Larger objects would have a number of accesses roughly equal to their size in fields. Very large objects would essentially be repositories for data; only a small fraction of their fields would be touched during a given computation. As the overall average number of accesses increases, however, such an importance curve becomes less likely.

A decreasing importance function is counterintuitive and would result if the access frequency were inversely related to object size. Data in larger objects would not be needed as much as data in smaller objects. One explanation is that large objects are often used as data repositories. Although circumstances exist where the mode of processing is to iterate over all members of a set, in many cases only the knowledge concerning the membership or absence of a particular value is required.

If the average number of accesses increases with size, then a rising importance function would appear. All data is equally important under these conditions, regardless of where it is found. Since the amount of data an object contains is typically proportional to its size, one expects the importance function to increase. The limiting case in which every field is equally likely to be accessed results in a linear importance function with a normalized slope of one.

Analysis of the importance function for data objects yields some hypotheses, but more data points are needed before they may be termed conclusions. Except for two data points, the curves for the first and third traces linearly increase with object size with a normalized slope much less than unity. The remaining three traces have a two-tier function. There are two approximate values, say *hi* and *lo*, that the importance curve attains. Define the lo set to be the collection of points at which the

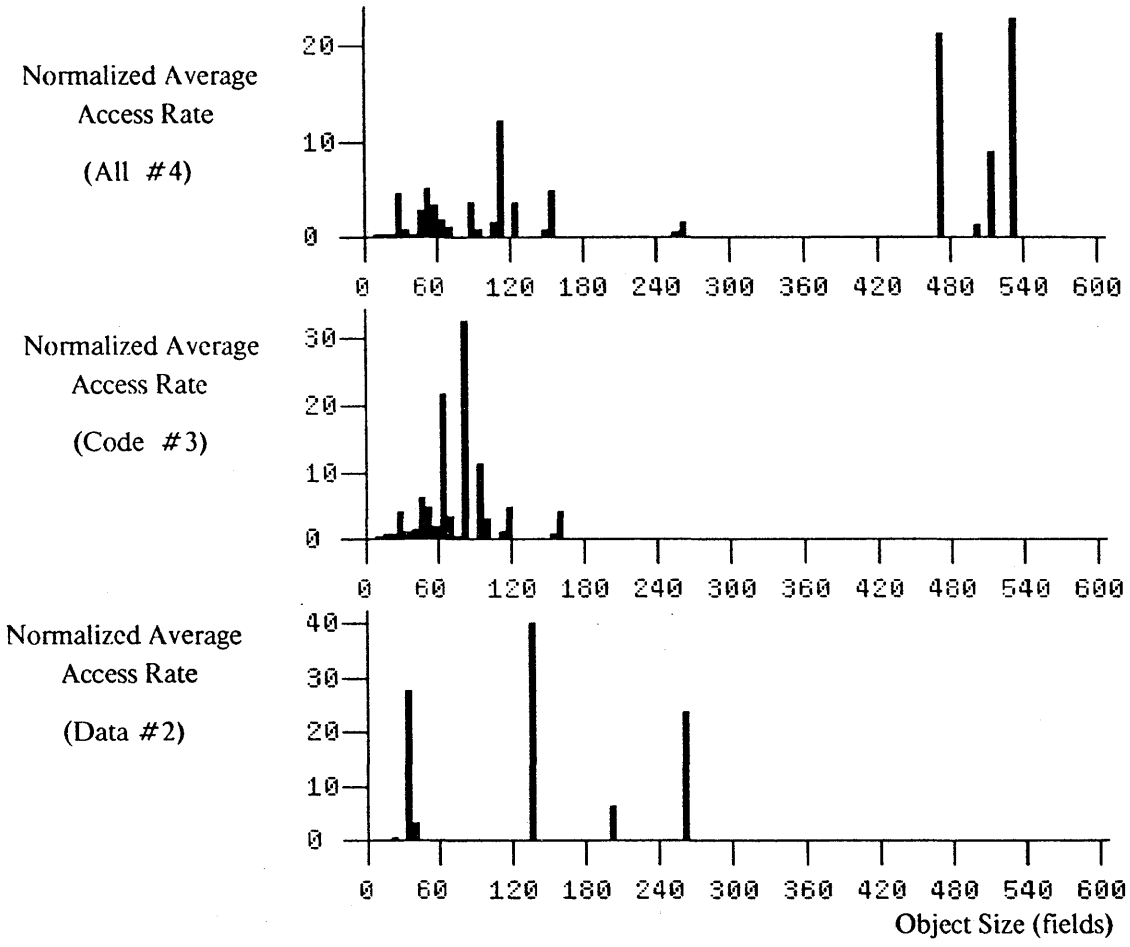


Figure 4-7 Importance Function
(note the differences in trace number)

function takes on the lo value. The hi set is defined similarly. Restriction of the importance function to either one of these sets results in a flat or slightly increasing linear fit.

Code importance functions were also superpositions of lo and hi functions. Rarely used message dictionaries, quick methods, and microcoded primitives are all accessed a small number of times and account for some of the lo activity. Hi values only occurred for large objects with at least 60 fields. In addition to size, hi values were caused by the repeated execution of methods and the frequent evaluation of blocks of code that served as the body of loops. However, there are not enough data points to warrant any conclusions regarding the overall slope of the importance function for code objects.

### *4.7.3 Implications for Caches*

The importance function effectively partitions objects into two sets. Objects in one set were, on the average, accessed repeatedly during the computation, while elements of the other group were not. Size is not a valuable predictor of membership in these sets. The lo values may be clustered near small (trace 5, code objects) or large (trace 4, code objects) sizes. Within either set, the importance curve may be approximated by a linear function with a normalized slope varying between zero and a positive value less than one. Given that an object is used during a computation, its importance cannot be predicted solely from the knowledge of its size. However, size can be used to estimate the relative importance of the object within either set.

The access frequency distributions and importance functions reveal that a small number of both code and data objects are heavily referenced by the Smalltalk virtual machine. These two sets are at times correlated in that the repeated evaluation of a block context comprising a loop corresponds to the processing of elements of an array, entries in a hash table, or bit sequences in a bitmap. Other frequently used code objects correspond to class dictionaries and methods that are accessed to perform messages sent to a large number of distinct, short-lived receivers.

Code-data distinctions and size observations cannot be profitably used to gauge the reference importance of a given object. Instance to class compression has indicated that while computations have some classes and objects in common, there will be little overlap on the size scale of small hardware caches. Even if frequently used objects and/or classes of objects can be identified and given preferential treatment, little gain would be realized beyond what may be achieved by a naive, demand-fetching, cache management scheme.

## 4.8 Reference Counts

One important consideration in any grouping scheme is the distribution of the reference counts of objects in the system. As long as a page is the physical unit of swapping from the disk to core, it is important to maximize the weighted sum of pointers that are within a page.

Weighting factors may be gained from a static, quasi-static, or dynamic analysis. Section 4.6 discussed a number of quasi-static and dynamic weights. A static analysis, however, is the simplest

to obtain. Without requiring any knowledge of the system beyond indistinguishable objects and pointers, static weighting schemes based only on reference counts can be employed. For example, all incoming pointers to an object could be considered equivalent. Then the weight of a pointer may be defined to be the multiplicative inverse of the true reference count of the object to which it refers. All weights would therefore lie in the half-open interval (0, 1]. These weights could then be input to a graph-theoretic algorithm that optimizes the intrapage pointers. Objects in interesting topological regions, such as cliques, cycles, or strongly connected regions, could also be identified and grouped together.

Roughly 75% of all objects in the Smalltalk-80 system had a reference count of one. The distribution of reference counts then fell off rapidly, with a small number of objects having the overflow value of 127. Even in the presence of high overflow counts, the static mean was still only two.

Two lengthy execution sequences were monitored in order to secure dynamic and quasi-static reference count information. The reported values represent the reference counts of all touched objects at the conclusion of the computation. Permanent objects and dynamically created objects whose reference count dropped to zero during the computation were not included. Compared to the static case, there was a slight decline in the fraction of objects with a reference count of one in both computations. For reference counts larger than one, the dynamic and quasi-static curves were similar in shape to the static distribution. Quasi-static reference counts were only slightly larger than their static counterparts. Although the median was one in both cases, the third quartile had risen to 4 and 5 while the mean had climbed to 9. Dynamic reference counts were comparable to the quasi-static results.

Reference counts alone yield limited information when ordering the fields of an object according to the reference counts of the objects named in the fields. Three-fourths of all objects cannot be distinguished on the basis of their reference counts. Most grouping permutations for objects derived from reference counts will default to the identity permutation. The initial placement for the reference count grouping algorithm will therefore be similar to the arrangement derived from the default grouping strategy. While reference counts do not provide enough information to distinguish most objects, the high fraction of objects with a reference count of one substantially reduces the number of decisions made by a grouping algorithm based on static connectivity.

One disturbing result from the standpoint of static grouping was the moderate use of objects with reference counts substantially larger than one. The ability of any static grouping algorithm to handle such an object depends on the nature of the references to that object. If a single object causes most of the dynamic accesses to this object, or if the set of objects that dynamically refer to this object may be easily grouped on a single page without adverse performance implications, the presence of these types of objects does not pose serious problems. On the other hand, if neither of these two conditions are met, then regardless of the initial placement, there is a high probability that a fault on this object will not be satisfied by the in-core buffer and hence will cause a page fault. If

the static importance of a high reference count has a counterpart in the dynamic domain, then after the first fault on this object, the object should tend to remain in core. Subsequent references to this object will probably not cause an object fault, since the object should be accessed often enough to survive many purge attempts. In this case, the initial placement of the object is not critical in an object-swapping scheme. In a page-swapping environment in which objects are not dynamically moved, the location of such an object in the virtual memory can have a nontrivial impact upon paging performance because the use of that object requires that the entire page be kept in core. This example highlights one of the features expected to be present in an object-swapping system. Different initial placements will tend to have less of an effect on performance in object-swapping systems than in paged virtual memories.

## 4.9 Selectors as a Percentage of Literals

A literal is a compile-time, manifest constant. CompiledMethods may contain one or more of these constants in what is called a literal frame. Some literals, known as *selectors*, are used to designate the names of messages sent to objects and correspond to the names of procedures in conventional programming languages. While the message name is known at compile time, the binding of this name to the code that performs the appropriate actions is not accomplished until run time. This binding, delayed because the true context in which the selector name is resolved is not available at compile time, is dependent upon the class (and possibly the superclasses) of the receiver and the contents of the message dictionaries for these classes.

This form of indirect linkage and delayed binding is not limited to selectors and CompiledMethods. One common example outside of Smalltalk-80 is the resolution of file names in a multiprogramming environment. A symbolic file name supplied by the user or a program is mapped into an actual file according to the user, the directory to which he is connected, the search rules, and the contents of the file directory.

A single selector may have a static and dynamic correspondence with any number of CompiledMethods. The inability to easily map a selector into a unique CompiledMethod prevented any serious efforts at grouping methods. However, all methods occur in exactly one message dictionary. Methods within a particular dictionary belong to the same class and are certainly related. Hence the system automatically partitions the methods into related subsets in a manner that may be utilized by grouping strategies.

The chief goal of the this analysis was to determine the relative importance of selectors compared to all literals. Static data was obtained by calculating the selector-literal ratio for all CompiledMethods. Quasi-static and dynamic results considered only the accessed fields in the literal frame of a method and neglected untouched literals. Each CompiledMethod used during the computation gave rise to a quasi-static and a dynamic fraction. Quasi-static counts of selectors and literals distinguished between instances of the same literal appearing in distinct CompiledMethods.

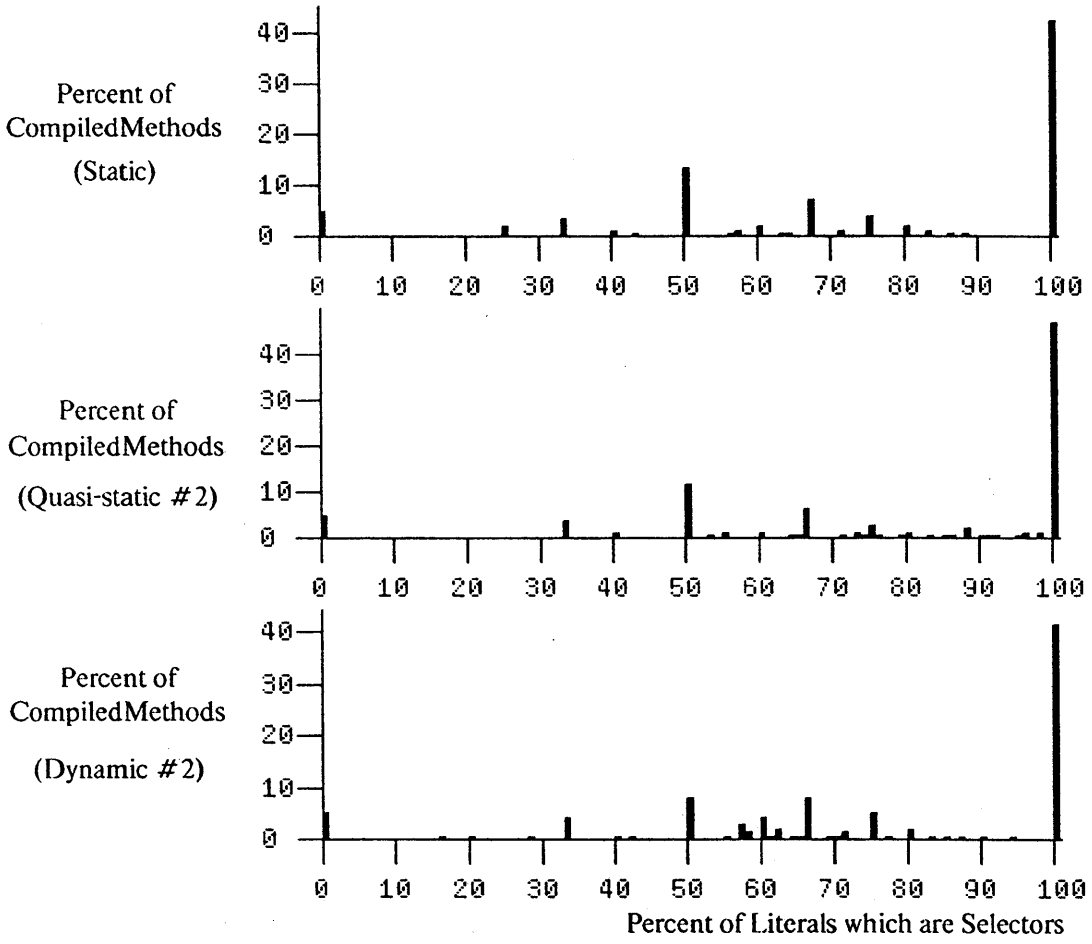


Figure 4-8 Selectors as a Percentage of Literals

In any given method, the expected value for the selector-literal ratio was three-fourths. Given a literal from any method's literal frame, however, the probability that it was a selector was about two-thirds. This discrepancy arose because the fractions from all CompiledMethods were weighted equally and were not assigned a value proportional to the length of the literal frame.

Most quasi-static and dynamic selector-literal figures slightly exceeded their static counterparts (Figure 4-8). One exception was the dynamic median for the first execution sequence, which was much larger than the static median. If selectors and non-selectors were evenly distributed within a method, then the presence of conditional jumps, early returns, and iteration statements would not affect the selector-literal ratios. However, there was a pronounced tendency for this ratio to rise in the quasi-static case and especially in the dynamic case.

Message selectors comprised the bulk of the literals in both the static and dynamic measurements. If literals are to be considered, static grouping algorithms ought to concentrate on selectors. Although code relocation in conventional programming languages has proved to be an effective technique in improving virtual memory performance, no serious attempt was made to group CompiledMethods for the following reasons:

> In the spirit of blurring the code-data distinction, CompiledMethods were considered to be ordinary objects and were treated as such, even by static grouping algorithms.
>
> Motion of code segments had been previously researched and has shown its worth. A similar investigation into grouping of data objects needed to be done.
>
> Delayed binding of selectors to code and the lack of a type (-inference) mechanism would have unduly hindered efforts to determine the important intermethod links. Inferring the types of receivers of messages by monitoring lengthy computation sequences would have provided only a probabilistic mapping from an instance of a selector to a set of CompiledMethods and not a deterministic function.

Research needs to be done in the area of code motion in object-oriented systems containing type inheritance. At the very least, its utility would derive from a simple comparison between the improvements in paging performance due to code relocation and the gains realized by grouping data. An investigation into the validity and ease of translating dynamic behavior into type information may remove a tremendous documentation burden from the programmer or a static type-inference mechanism. Such an effort could also determine the distribution for the number of classes in the hierarchy searched before finding the appropriate CompiledMethod.

### 4.10 Summary and Conclusions

A number of important points may be drawn from the preceding set of analyses. First and foremost is the high level of consistency between traces based upon radically different computational sequences. A random comparison between any statistic from one trace and the same statistic from another trace usually yields a close agreement. The measurements report inherent properties of the Smalltalk-80 programming environment rather than the characteristics of the monitored computations. Secondly, this close agreement lends support to the conclusion that the traces were long enough to filter most of the transient behavior. Incremental measurements of

fractional usage, the statistic most susceptible to this type of problem, support this claim. Finally, most of the abnormalities, unexpected data spikes, and overall trends in the distributions were easily explained by a quick analysis of the portions of the reference trace that gave rise to these findings. The importance of class objects (size ten) and message dictionaries (size roughly equal to a power of two) was highlighted by the appearance of relevant numbers in the statistics and spikes in the bar charts.

While the preceding analysis presents an in-depth picture of the reference behavior of Smalltalk-80, the generality of the results must be questioned at this time. As was evident from the discussion in section 4.6, the entire system was not analyzed. Only those portions needed to support the five computations were thoroughly monitored and studied. The total number of objects touched in each reference trace was very small. Since the measured system was designed, written, and implemented by only a handful of researchers, the programming styles and techniques evident in Smalltalk represent at most a few points in a wide spectrum. Compounding this problem is the fact that a close collaboration between these individuals for extended periods has undoubtedly influenced and merged their respective styles. A second important limitation is the nature of Smalltalk itself. This programming environment was designed to be a system undergoing continous change, modification, and evolution. All measurements, however, were made with a single snapshot. That is, all traces were taken from a single version that could be started from a particular state of the virtual machine any number of times. On the other hand, this instantaneous snapshot prevented any time-dependent tendencies from corrupting the data. Finally, it must also be noted that only one particular system was analyzed. Similar research needs to be undertaken on other object-oriented systems before the generality and relevance of these results may be ascertained.

# 5. Static Grouping Algorithms

Application of a particular grouping scheme to a running Smalltalk system produces an initial placement that represents the position of all objects in the virtual memory. Nine different initial placements were used in performance studies for the two types of virtual memories. Before any dynamic tests were done, the initial placements were statically analyzed to determine their similarities and differences. As Chapter 4 predicted, information derived from actual reference traces or from static reference counts had only a limited impact on grouping schemes. Performance predictions for both LOOM and a paged virtual memory were made using this knowledge. These static measurements are evaluated in Chapters 7 and 8 as to their predictive or non-predictive capabilities. Information regarding initial placements was also available after the fact and was used to explain the behavior of the virtual memory simulators under various configurations and purging policies. Numerical results cited in this chapter are presented in Appendix B.

## 5.1 Nine Algorithms

Consider a total ordering on all objects in a given Smalltalk environment. This ordering, together with the disk size of the objects, yields a unique virtual memory position for each object. Define the mapping from the set of objects to their disk addresses resulting from some grouping scheme G as the *initial placement* generated by G. Hereafter, *configuration*, *grouping*, and *arrangement* will be used as synonyms for initial placement.

Nine distinct grouping strategies were applied to the entire set of objects in the programming environment. These nine algorithms and initial placements fell into five *categories*. Groupings within a category produced similar measurements in both static and dynamic environments. Initial placements in different categories were not alike in a static sense. Their dynamic behavior, however, was not consistently different.

Two groups of three algorithms are graph-theoretic and use the directed graph where Smalltalk objects are represented by nodes and pointers are represented by directed arcs. One category of grouping schemes is based upon a depth-first traversal of this graph and attempts to put high-probability paths on the same page. The second collection uses a breadth-first approach and endeavors to locate all offspring of an object on a single page. When a new node is encountered in either type of traversal, a decision must be made as to the order in which the descendants of the node are to be investigated. This ordering may be succinctly described by a permutation. The default case is the identity permutation, which corresponds to an examination of an object's fields in the order assigned by the compiler when the class of that object was defined. Another permutation arose from reference counts. Pointers referring to objects with the lowest reference counts are followed first, while ties are resolved by defaulting to the ordering decreed by the identity

permutation. Finally, information garnered from traces of actual execution sequences similar to the ones analyzed in the preceding chapter provided the third permutation. Under this grouping strategy, fields accessed the most are investigated first. The identity permutation is again used when the dynamic information cannot distinguish between two fields. Because of the inherent inefficiency of the virtual machine emulator, only a limited amount of dynamic information could be obtained. Nevertheless, the dynamic permutation differed from the identity permutation 23% of the time. Each of the three permutations was used in both types of traversals. The foregoing discussion defines six of the nine initial placements.

Since the Object-Oriented Zoned Environment (OOZE) virtual memory [KAEH81] for Smalltalk-76 grouped objects by class, the initial placement derived by packing together all instances of a class was called the *OOZE* configuration. A one-to-one hash function, which simply permutes the bits of the unique identifier (*UID*) of objects, was used to scramble the OOZE ordering and derive the *hash* or *random* initial placement. This random arrangement was used to establish a benchmark against which improvements in paging performance could be classified as trivial or substantial. Of critical importance was the existence of an inverse of this hash function. This inverse function simply applies the inverse permutation to the bits of a number in order to compute the UID of an object. This inversion capability allowed the LOOM simulator to use a close approximation of the purging policy found in the original OOZE virtual memory implementation.

The ordering of the objects in the ninth grouping was derived from the compressed traces that are discussed in Chapter 6. For any reference trace T, consider the ordering on objects defined by the first object reference to each object mentioned by T. Assume all objects not referenced by T are ordered in some fashion such that all unneeded objects are ordered after all referenced objects. Call the initial placement defined by this ordering an *optimal* grouping with respect to T. For any page-swapping virtual memory that can process T without purging, this arrangement is optimal in the sense that the number of page faults is minimized. There may be many initial placements that cause this number of page faults while processing T. However, there is in general no initial placement for a given trace that is optimal across all types of virtual memory configurations, values of parameters, and kinds of policies.

OPT1, OPT2, and OPT3 are the names of the groupings derived from the three compressed reference traces. To distinguish between the optimal and the other initial placements, call the strategies generating the remaining eight arrangements the *realizable* grouping algorithms. The optimal and random groupings bound the region of reasonable static arrangements and act as benchmarks from which the relative as well as the absolute improvements in paging performance for a given initial placement can be ascertained.

Snyder [SNYDr] defines two levels of grouping, *internal* and *external*, and three techniques: *a priori*, *a posteriori*, and *dynamic*. Internal restructuring relocates fields within an object while external restructuring rearranges atomic objects. All nine groupings are forms of external restructuring. No groupings are dynamic because the placement of objects is always fixed in

advance. The two initial placements derived from dynamic data, as well as the optimal groupings, are a posteriori arrangements in that they utilized information gained from performing computations in the programming environment. The six other initial placements were derived with a priori techniques because only statically available information was utilized.

## 5.2 Static Pointer Distance

Define a pointer to be *immediate* if the particular object referenced by this pointer may be identified from an examination of only the numeric value of the pointer. Nil, true, false, and SmallIntegers are examples of immediate pointers. All references not having this special property are defined to be *non-immediate.* Objects referenced by non-immediate pointers are true entities requiring space in core, a home on the disk, or perhaps both. Let p be a non-immediate pointer contained in object A that refers to object B. Assume $d_A$ ($d_B$) is the disk address of the first word of A (B). Then the *static distance* associated with p is $|d_A - d_B|$. Notice that the distance of p is not a function of its relative position in A.

Figure 5-1 indicates that surprisingly few pointers had a static distance less than one disk page (256 16-bit words). Only 1 pointer in 7 for the depth-first initial placements satisfied this criterion, while the fraction of *close* pointers for the optimal grouping varied between 1/7th and 1/10th. Compounding this problem was the existence of page boundaries. Of all pointers with a distance of less than one page, one may expect only half to lie on a single page. This *on-page pointer ratio* dropped to 1 in 33 for the OOZE initial placement, 1 in 300 for the random initial placement, and only 1 in 500 for the breadth-first cases.

On-page pointer ratios do not provide information upon which a valid judgment of a breadth-first restructuring can be made. These grouping strategies depend on the fact that a single page fault for any one offspring of an object moves many of its siblings at least as far as the in-core disk buffer.

Distributions of pointer distances partitioned the set of initial placements into five categories. The numbers for the three depth-first configurations were similar. So were the figures for the three breadth-first initial placements. The OOZE, random, and optimal groupings, on the other hand, were each markedly different from the others. Similarities within a category and striking differences between categories appeared not only in this and other static analyses but at times in the dynamic measurements reported in Chapters 7 and 8.

Except for noise, the distance function for the random initial placement was monotonically decreasing and appeared to be a dying exponential. A truly "random" initial placement would have a linear, decreasing distribution. Since the median, the mean, and first quartile of the theoretical random configuration were larger than the measured values, the random initial placement was not truly random. Because it was drastically different from all other initial placements according to the neighbor relation (section 5.3.2), the random configuration was retained as a benchmark.

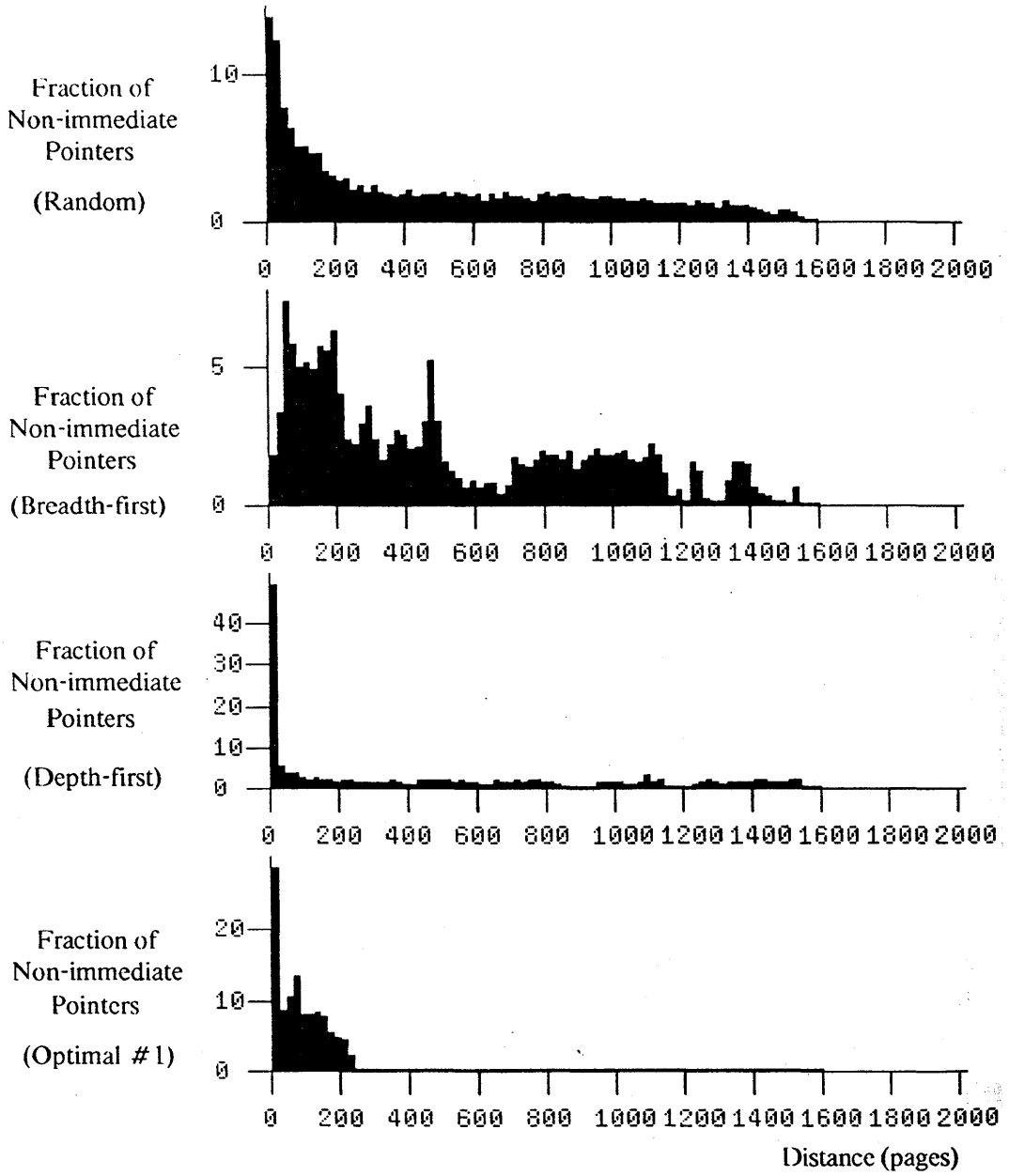


Figure 5-1 Static Pointer Distance (note scale change)

There are many reasons why only a small percentage of pointers had short distances. First, every object contains a reference to the class to which it belongs. While many classes had only a few instances, other classes contained well over a hundred instances. A second cause of the low on-page pointer ratios is the size of objects. In LISP systems, for example, list cells contain only two fields and the CAR of such an element is typically immediate. Hence the CDR will normally fall on the same page, once a linearizing of the list structure has been accomplished. Clark [CLAR76, CLAR77, CLAR78] has reported that approximately 85% of all list pointers do not cross a page boundary. After linearizing by a depth-first traversal in either the car or cdr direction, this percentage was increased to over 90%. Most Smalltalk objects have more than two fields. Since references may point to arbitrarily large structures, it is not possible to effect a similar linearizing of the structure. The third problem is the relatively high percentage of objects that have a reference count of one. Placing such objects as close as possible to their sole reference severely limited the number of possible initial placements.

The selected static grouping strategies had their greatest impact on the fraction of short pointers; medians, means, and quartiles of the distance distributions are comparable. From the standpoint of paging performance in an object-swapping scheme, the fraction of intra-page pointers is of prime importance. Less critical is the distribution of lengthy pointers. A pointer distance of 2 disk pages is equivalent to a distance of 2000 disk pages, unless secondary factors, such as disk arm movement, are under consideration.

One shortcoming of this distance measure is its inability to capture the notion of a *swapping set*. It is often the case that when an object is brought into core, there is a high probability that a small collection of other objects will subsequently be accessed. If these objects are not packed onto a single page, then the number of page faults required to transfer this swapping set into core will tend to decrease as the number of pages containing these objects is reduced. The distance of the inter-page pointers within a swapping set is inconsequential. Swapping set size in disk pages is what affects performance.

In addition to the static notion of pointer distance, a dynamic, time-varying interpretation exists. A pointer is *followed* by gaining an in-core reference to the object (or its leaf) specified by the pointer. If the pointer is a lambda, then it must first be resolved. The dynamic distance of a pointer is defined to be zero if the pointer can be followed without causing any page faults. Otherwise, that pointer is said to have an infinite dynamic distance.

If the current grouping of objects on the disk is not considered, then at any given moment in an object-oriented virtual memory, the only portion of the state of the environment due to the static grouping algorithm is the set of objects residing in the in-core disk buffer. A static grouping algorithm has only a limited effect on the dynamic distance, which is primarily a result of the purging policy as well as the dynamic grouping strategy used by the virtual memory. However, controlling this small portion of the state of the virtual memory can have a large cumulative effect on dynamic performance.

### 5.3 Neighbor Relation

The following measure of the degree of similarity between two initial placements investigates local grouping arrangements by examining the collections of objects on a single page. For any initial placement IP, define the *neighbor relation* for IP as the set of unordered pairs (A, B), such that $A \neq B$ and A and B are both on the same page under IP. This relation is the same as the equivalence relation defined by the partitioning of the set of objects according to page boundaries except that the neighbor relation is not reflexive. Including the pair (A, A) for all objects A would significantly reduce the differences between distinct groupings. The neighbor relation is symmetric but not quite transitive. However, if R is the set of pairs corresponding to the neighbor relation, then

$$(A, B) \in R \ \wedge \ (B, C) \in R \ \wedge \ A \neq C \ \Rightarrow \ (A, C) \in R.$$

Given two initial placements, $IP_1$ and $IP_2$, and their corresponding neighbor relations, $R_1$ and $R_2$, one measure of their similarity is the proportion of elements in $R_1$ found in $R_2$ (and vice versa). Define the *size* of a neighbor relation to be the number of unordered pairs in the relation and denote the size operator by two vertical bars: $|R_1|$. Let R be $R_1 \cap R_2$, which is the set of pairs of objects that are found together on the same page in both $IP_1$ and $IP_2$. The *fraction* of $R_1$ *retained* by $R_2$ is then defined to be:

$$\frac{|R|}{|R_1|}.$$

As an abbreviation, the value of the preceding formula will be referred to as the *retained fraction* or *retained value.*

#### *5.3.1 Effect of a Continuous Displacement*

The following example attempts to establish an intuitive correlation between retained fraction values and different initial placements. Consider any initial placement, $IP_1$, and its corresponding neighbor relation, $R_1$. Let A be an object not found in $IP_1$ that has a disk size of $A_s$, where $A_s$ is at most the size of one disk page. If f is the fraction of a disk page occupied by A, then $0 < f < 1$. Let $IP_2$ be the initial placement created by placing A at the beginning of $IP_1$. Then all objects in $IP_2$ will be offset from their position in $IP_1$ by $A_s$. If m is the expected number of objects on a page, the number of unordered pairs in the neighbor relation derived from a typical page will be

$$t = \frac{m*(m-1)}{2}.$$

In $IP_2$, the corresponding set of objects falls on two pages. The expected number of objects from this set on the first page will be $(1-f)*m$, while the remaining $f*m$ objects will be found on the second page. The number of unordered pairs due to this set of objects that are retained under $IP_2$ is

$$p = \frac{(fm)(fm - 1)}{2} + \frac{((1 - f)m)((1 - f)m - 1)}{2}.$$

Let r(m, f) be the ratio of preserved pairs to total pairs for this page, i.e. p/t. After a few arithmetic simplifications,

$$r(m, f) = 1 - \frac{2f(1 - f)}{1 - 1/m}.$$

Assuming there are m objects on each page, the preceding analysis is valid for all pages in $IP_L$. Hence r(m, f) is the retained fraction. Note that r is an increasing function of m. For a fixed f, as the average number of objects per page increases, the two initial placements become more similar. When m is fixed, r attains a maximum value of 1 if f=0 or f=1. Because of the symmetric nature of this problem, r attains its minimum when f=1/2. Since m≐6 for a page size of 128 words, the calculated minimum is 40%. This value is a bit smaller than the observed values of 49.6% for the depth-first traversal with the identity permutation and 49.2% for the hash initial placement.

One surprising outcome of this simple example is that two initial placements identical except for a half-page offset have a retained value of less than 50%. The maximum occurs only in the limit as m goes to infinity. This characteristic is due to the quadratic nature of the neighbor relation, since a page containing m objects contributes $O(m^2)$ entries to the neighbor relation. An important weakness of the retained fraction measure exposed by the preceding example is its inability to effectively capture the *intuitive* notion of the degree of similarity between two initial placements.

The retained fraction measure sharply distinguishes between intra-page pointers and inter-page pointers. Since the disk buffer is generally larger than a single page, it may be more appropriate to define the page size for this measure to be the number of words occupied by some fraction of the disk buffer. In this way, some pointers that are inter-page in the static sense are assumed to be intra-page (actually intra-buffer) in the dynamic sense. One problem with this definition is inability to statically predict the sequence of pages swapped between the disk and the disk buffer.

### *5.3.2 Discussion*

The retained value of each initial placement with respect to all other initial placements was calculated for three different page sizes. For a given page size, the aforementioned partitioning of the initial placements was evident. When any two breadth-first arrangements were compared, the retained value varied between 45% and 81%. For any two depth-first initial placements, the corresponding range was narrower, extending from 52% to 63%. On the other hand, when any breadth-first initial placement was compared with any depth-first grouping, the retained fraction ranged from only 17% to 24%. All retained values for the OOZE initial placement fell into the interval from 16% to 24%, while the corresponding numbers for the random configuration were between 0.2% and 3.3%. Retained fractions for the three optimal arrangements spanned a wide spectrum from 0.2% to 40.1%. Comparisons between optimal initial placements were not meaningful because of the limited overlap between any two. In light of the results of the preceding example, these figures indicate a strong similarity within each category and a great difference between categories.

### 5.3.3 *Effect of Page Size*

Some interesting tendencies of the neighbor relation were noticed as the page size was varied from 128 bytes to 256 bytes and finally to 512 bytes. The additional information derived from the following analysis also partitioned the grouping strategies into the same five categories.

The linear dependence of the size of the neighbor relation on page size is easily explained. If there are m objects on a page and p pages in the initial placement, there will be approximately

$$|R| = \frac{pm(m-1)}{2}$$

elements in the neighbor relation. Increasing the page size by a factor of s will mean there will be s*m objects on an average page but only p/s pages. Then the size of the relation will be

$$|R'| = \frac{(p/s)(sm)(sm-1)}{2} = \frac{pm(sm-1)}{2} = |R|(1 + \frac{m(s-1)}{m-1}).$$

Hence,

$$\frac{|R'|}{|R|} = 1 + \frac{m(s-1)}{m-1},$$

which is 1.1s - 0.1 for m=11. This formula yields close agreement with the empirical results for s=2 and s=4. Actual values, however, were slightly smaller than the preceding analysis would predict.

The retained fraction for the hash initial placement increased against all other configurations as the page size was increased. This may be explained by considering a truly random initial placement and a second, non-random initial placement. Assume these initial placements each occupy p pages. Given any two objects, the probability that both lie on the same page is 1/p, neglecting the portions of the pages occupied by the objects. For a given page in the non-random configuration, the expected fraction retained by the random initial placement will be

$$\frac{(1/p)(m(m-1)/2)}{m(m-1)/2} = \frac{1}{p}.$$

Assume the page size is increased by a factor of s. Then the expected fraction retained will be

$$\frac{(s/p)(sm(sm-1)/2)}{sm(sm-1)/2} = \frac{s}{p}.$$

Both of these retained fractions scale from a page to the entire initial placement. Therefore, the ratio of retained fractions for the random initial placement will roughly be equal to the ratio of the page sizes. A doubling occurred in the retained value of the hash initial placement with respect to all three depth-first initial placements in the 128-to-256 case as well as in the 256-to-512 case. Although the retained values of the random initial placement with respect to the three breadth-first configurations approximately doubled when the page size was increased from 128 words to 256 words, only a 30% increase occurred when the page size was increased from 256 to 512. While the OOZE-hash and optimal-hash retained fractions increased each time, the ratios were much less than two. Most of these discrepancies arose from the fact that the hash initial placement was not truly

random. A total of 1580 pages implies that the predicted retained value for the 256-word page size would be 0.06%. Empirical results, however, were in the 0.3% to 2.9% range.

Except for the interactions with the hash configuration, the retained value of two initial placements derived from different types of grouping schemes typically fell for larger page sizes. For example, the retained fraction of the OOZE neighbor relation with respect to the other six realizable initial placements fell as the page size was increased. In 34 of the 36 retained fractions between a depth-first initial placement and a breadth-first initial placement, the value fell. Very small increases caused the two anomalies. An intuitive explanation for these observations is that as the page size increases, larger and larger segments of the two initial placements are being compared. Differences between the two configurations should become more evident as long as there are still a moderate number of pages. (Note that if the entire initial placement fits onto a single page, then all initial placements are equivalent under this measure.) A more formal explanation for this behavior is the linear dependency between the size of the neighbor relation and the size of a disk page that arose from the squared term in the number of pairs of objects per page. In effect, more and more comparisons are being made. If the number of successful comparisons does not grow at least linearly with the page size, then the retained fraction will fall.

Conflicting changes were seen when retained fractions for two arrangements produced by similar grouping schemes were examined. For the three breadth-first arrangements, increasing the page size increased the retained value in 10 of 12 cases. The breadth-first initial placements became more alike as larger portions were compared. For the depth-first arrangements, only 5 of 12 retained fraction values increased. These results are explained by an inherent difference between a depth-first and a breadth-first traversal of a graph. In a breadth-first strategy, reordering the offspring has only a local effect as far as the offspring are concerned. These objects will be adjacent regardless of the particular permutation used. As the page size is increased, more and more siblings will be on the same page, differences in their order will become less significant, and the retained value will increase. In a depth-first traversal, however, reordering the offspring can have a more substantial impact on the resulting initial placement. Each offspring may be the root of an arbitrarily large subtree of the tree defined by the traversal. Siblings may be placed arbitrarily distant from each other and their common parent. The particular permutation employed can thus have a substantial effect on the final placement of the offspring. While increasing the page size potentially increases the number of siblings on the same page, the number of elements in the neighbor relation is also increased. Depending on the relative importance of these two conflicting tendencies, a larger page size may increase or decrease the retained fraction between any two arrangements produced by depth-first grouping strategies.

The value retained by $IP_1$ with respect to $IP_2$ is comparable with but usually not equal to the value retained by $IP_2$ with respect to $IP_1$. Different numbers of elements in the respective neighbor relations cause different neighbor relation sizes, prevent the retained fraction operator from being commutative, and thus mar the symmetry of the retained value tables. Neighbor relation size again differentiated between most initial placements but upheld the similarities within the breadth-first

and depth-first groupings. The OOZE configuration had by far the largest such relation. This may be attributed to the packing together of objects of similar size. Pages containing only small objects had a substantial impact on the size of the neighbor relation because of the quadratic effect. The obvious simple example indicates that the influence of the small objects is enough to overcome the opposing tendency of the pages containing only large objects.

### 5.4 Conclusions and Predictions

The retained fraction measure, which only considers the intra-page adjacency of objects, was used to compare and contrast the initial placements. One deficiency is the naivete of this measure. All neighboring objects were assumed to be equally important. But this is simply not the case. Many of the objects on a page were related; others were placed on the page only because the attention of the grouping algorithm had turned to a new yet unrelated portion of the graph. A more sophisticated measure might take these differences into account. For example, one could rank the importance of the elements of the neighbor relation by assigning weights to each such element. Such a weighting scheme may be based upon the number of direct pointers between the objects, the presence or absence of a cycle involving the two objects, or a more complicated function that depended on the number and nature of all "short" paths between the two objects. However, while this type of measure may potentially be more useful in predicting the relative paging performance of two groupings, it would not be as useful nor as quick as the neighbor relation for purposes of comparing initial placements.

A static analysis of the nine initial placements has highlighted the differences and similarities of the groupings. Initial placements derived from nearly similar grouping strategies were roughly the same; great differences existed between those generated from distinct grouping algorithms. One untested hypothesis suggested by the measurements is that all possible arrangements derived from one kind of search strategy are very similar. Low static connectivity meant that minor modifications to any grouping algorithm based primarily on connectivity would have only a limited effect on the final placement of objects. The particular set of permutations used during the traversal would not have a substantial impact on the final grouping as defined by the retained-fraction measure. Traversal type, on the other hand, would be the dominant factor. This theory holds for the breadth-first traversal of the Smalltalk environment, where in many cases all offspring may be packed onto a small number of pages. Although the three depth-first arrangements were very similar, this information does not constitute a proof that all possible depth-first traversals will yield similar groupings. The permutations used in the three depth-first traversals were not appreciably different in a sizable fraction of the cases. Other unexplored permutation sets may exist that yield vastly different depth-first initial placements.

Information gained from monitored execution sequences and static reference counts had only a slight effect on the groupings generated by a particular traversal strategy. For both the depth-first and breadth-first schemes, the initial placement derived from the dynamic data surprisingly had a higher on-page pointer ratio than did the other two arrangements. The difference between the

ratios for all elements within each category was small enough to render this distinction of dubious value. Since the dynamic information dealt with only a small number of classes and the reference count data could not distinguish between three-fourths of the objects, it was only natural for the initial placements generated by the same traversal type to be very similar.

Another hypothesis suggested by the data is that the static similarities would carry over into the dynamic domain. Paging performance and other monitored statistics should be very similar for the breadth-first as well as the depth-first arrangements. No solid conclusions were made regarding comparisons of different categories. Common sense indicates that the random and optimal arrangements should have the poorest and best performance, respectively. On-page pointer ratios would rank the depth-first groupings as the best, followed by the OOZE initial placement, the random configuration, and finally the breadth-first arrangements. The sibling-packing tendencies of the breadth-first strategies would undoubtedly improve their performance. Low fractional utilization values for data objects have indicated that many fields of an object are not touched during a computation. Since three fourths of all objects have a reference count of one, all siblings are not always needed. This information limits the performance improvement provided by sibling packing. There was no firm evidence to support a general consensus as to the "best" static grouping algorithm or even if there would be one. Rankings may depend upon the utility function used to evaluate their performance. Furthermore, this ranking may change when factors such as memory size, buffer size, fetching policies, and purging techniques are varied.

# 6. Reference String Compression

Actual reference traces provided the input data for the virtual memory simulators. Obtaining traces lengthy enough to cause moderate virtual memory interaction from the Smalltalk emulator required substantial computational power. Investing time and energy into data compression realized substantial benefits in the form of relatively efficient simulations as well as compact data files. Readers uninterested in the nature of the compression algorithm may skip immediately to section 6.2 without a loss of continuity. The only other portions of the thesis that require an understanding of the compression scheme are sections 8.3.7 and 9.1.

## 6.1 Developing the Algorithm

Much of the difficulty in presenting the compression algorithm will be avoided by applying a stepwise refinement approach to the main objective. The chief goal of the algorithm was to transform a complete reference trace derived from an emulation of the Smalltalk virtual machine into an equivalent, compact representation. Two reference traces are defined to be *equivalent* with respect to a virtual memory if, and only if, they cause the same sequence of page and/or object faults and the same sequence of page and/or object swaps.

Achieving this equivalence conflicted with an efficiency constraint. On the average, only a small amount of computation could be done for each reference generated by the emulator. Because of this restriction, the compression algorithm endeavored to preserve a weaker form of equivalence. Instead of guaranteeing the same sequence of object swaps both to and from primary memory, the compression algorithm concentrated on faulting at the expense of purging. Enough information was preserved to guarantee that the insertion of any reference deleted by the compression routine would not cause an object fault. However, information regarding the last use of an object was not explicitly included in the output of this algorithm. This lack of knowledge prevented an accurate recreation of the in-use bits on both pages and objects. Other minor difficulties arose from the fact that the compression scheme is object-oriented while the virtual memory simulation described in Chapter 7 is page-oriented.

### *6.1.1 A Simple Compression Scheme*

In order to increase the efficiency of the virtual machine emulation without requiring huge amounts of secondary storage, the compression scheme was designed to operate as a one-pass algorithm. Generation of the complete reference stream by the emulator, manipulation of this stream by the compression algorithm, and production of the final, compressed trace were done simultaneously. The algorithm may be viewed as a FIFO compression scheme that cached the set of most recently used objects filling a fixed size of primary memory. Define the *size* of such a cache to be the sum of the sizes of the objects contained in the cache.

References to any objects contained in this cache were simply discarded. A reference to an object not in the cache was appended to the output trace. This new object was then inserted in the cache. If the new size of the cache exceeded its limit, then some objects were purged using a FIFO object replacement policy. Since the cache was a window on the final segment of the output trace, the output trace defined the FIFO ordering on cache elements. An index into the output file called the *purge index* provided the information necessary for purging.

Cache size is the key to equivalence. Clearly, the size of the cache must be less than the smallest anticipated core size of any virtual memory simulation. Let this smallest memory size be M. Then the maximum cache size may be expressed as f*M, where $0<f<1$.

Consider a clock associated with a virtual memory that advances whenever an object is swapped into core or a new object is created. This clock advances one *tick* for each field in either type of object. Assume a virtual memory configuration with a core of size M and a particular purging policy can guarantee that an object will not be purged until at least f*M ticks have passed since it had been accessed. Then the original and compressed traces are equivalent (in the weak sense) with respect to this virtual memory.

This constraint guarantees that an object will remain in core for at least f*M ticks after the virtual memory processes a reference to it. All references deleted from the full trace by the compression algorithm are automatically satisfied, since any object specified by a deleted reference is guaranteed to have remained in core in the virtual memory simulation for at least f*M ticks.

Cache size, and therefore the efficiency of the compression algorithm, may be traded off against either the minimum anticipated core size or the complexity of the purging policies. Increasing f*M will lead to shorter compressed traces. At the same time, however, either the smallest possible core size will be increased or more constraints will be placed upon the purging schemes.

Unfortunately, there is no such f for some purging policies. Optimal, FIFO, random, and simple clock-based purging schemes may purge an object/page on the first reference after its last access. Although such occurrences are unlikely, since the guaranteed time before purging is zero, f must be zero. However, this concept is not applicable to optimal purging schemes. If an optimal scheme purges an object/page, then it will not appear in the reference trace for at least M ticks. Although no time guarantee exists for optimal replacement algorithms, the above compression scheme will preserve equivalence for these unrealizable algorithms for all f less than or equal to one.

Any purging policy that maintains a total ordering on objects/pages according to last access, such as LRU, can guarantee an f arbitrarily close to unity. More complicated clock algorithms, which monitor the ratio of the total size of all recently-touched objects to the total size of all untouched objects, can also guarantee any f. Demand paging with an LRU replacement algorithm was used in the simulation of the paged virtual memory. This simulation did not constrain the choice of f. Since an unsophisticated clock algorithm was used in the LOOM simulation, a conservative value

for f was used. M was 20K and f was one-half, which meant that the maximum cache size was 10K words. Section 9.1 will argue that this choice of f was valid.

### *6.1.2 A More Detailed Algorithm*

Distinguishing between read and write operations allowed the cleanliness of pages flushed from the LOOM disk buffer to be recorded and analyzed. A dirty bit in the cache entry for an object recorded the type of reference to that object in the portion of the output trace windowed by the cache. Read and write operations were distinguished in the compressed trace. A *read reference* was added to the output trace when the virtual machine emulator read a field from an object not contained in the cache. Whenever the emulator wrote on a clean object, a *write reference* was appended to the compressed trace. This action was taken regardless of the contents of the cache. All other references generated by the emulator were treated as before. An object may have appeared twice in some portion of the compressed trace delineated by the cache. If this event occurred, the first reference would necessarily be a read and the last reference would always be a write. Writing onto a clean object moved that object to the top of the ordering defined for the cache. Unless any other clean object in the cache was subsequently dirtied, this object would be purged after all the other objects currently in the cache. This promotion of the object in the cache ordering was effected by simply ignoring the read reference to the object in the compressed trace. When the purge index passed the read reference, the object would not be purged from the cache. The set dirty bit associated with the object's entry in the cache indicated that there would be an upcoming write reference to that object. This write reference would cause the object to be removed from the cache. In summary, dirty objects were purged only when the purge index swept across their unique write reference in the output trace. Clean objects were purged when the purge index swept cross their unique read reference. Because objects could appear more than once in f*M ticks of the compressed trace, the effective value of M was actually less than the chosen value.

Dynamically created objects could trivialize the importance of the initial placements. Computations that did not require many new objects could have been chosen. This approach was not taken because it would have seriously constrained the set of possible execution sequences. Completely neglecting new objects was another alternative. However, under that scenario, the LOOM simulation would not benefit from its late binding of objects to disk addresses. An acceptable compromise was the assumption that unique identifiers would not be reused. All objects with the same UID were assumed to have the same size. Therefore, at most one disk location was ever assigned to any UID. Garbage collection, reference counting, and/or compaction of secondary memory were not considered. Because the simulations did not implement a changing mapping from UIDs to object size, references to nonexistent fields were assumed to be references to the last field of the object.

The assumption concerning the invariance of the UID-size mapping was confirmed. Between 94% and 97% of all references agreed on the mapping from UIDs to in-core sizes. Much of the discrepancy arose from the *'become:'* operator and could not be easily accommodated by the

simulations. All in-core objects in Smalltalk are referenced indirectly through the object table (OT). *'Become:'* simply modifies the starting address of an object by changing its entry in the OT. This capability is important for variable-length objects that provide the illusion of an infinite capacity.

Two changes to the emulator reduced the amount of processing the compression algorithm was forced to perform. All references to the 32K display bitmap were ignored. By definition, this object would remain in core at all times in every simulation. The size of the display bitmap was not included in the amount of core used in a virtual memory simulation, since core memory was reserved for other objects. To translate the core size associated with a specific simulation to the actual size of core simulated, 32K sixteen-bit words must be added.

A substantial fraction of the references to CompiledMethods was not passed along to the compression routine. Code references were almost always consecutive, as section 4.5 has indicated. To preserve weak equivalence, the virtual machine emulator explicitly generated the first and last references to a CompiledMethod. An additional reference to the CompiledMethod was generated whenever a subcomputation terminated and control returned to the associated stack frame.

### *6.1.3 LOOM Requirements*

The preceding compression scheme is adequate for a paged virtual memory. However, the LOOM simulation requires additional run-time information, because of the use of compressed pointers, lambdas, and reference counts.

Let V be the contents of the field of an object. Define a value V to be *short* if, and only if, it is a sequence of bits, an immediate pointer, or a non-immediate pointer that refers to an in-core object. Whenever a field in an object is read in a LOOM-like system, its contents must be short in order for the computation to proceed. Whenever a field in an object is updated, both the new and the former values must be short. Since the new value was certainly used in the preceding computation and placed on the stack, it must be short. In either case, the compressor must provide the LOOM simulation with enough information to guarantee that the former contents of the field represent a short value. Instead of a simple sequence of accessed objects, the reference stream was implemented as a stream of ordered pairs of the form (A, B). The first element of the pair, referred to as a *left* or *object* reference, represents the accessed object, while the second element, referred to as a *right* or *leaf* reference, corresponds to the former value of the field. In LOOM, the ordered pair above would ensure that A was in core in its entirety; a leaf for B would be created only if the second pointer were not short.

The compression algorithm maintained additional information for each of its cached objects. Specifically, for each object in the cache, there was a single bit that told whether the object had appeared only on the right (as a leaf) or at least once on the left (as an object) in the recent past. For simplicity, the ordering of the leaves in the cache was defined to be the ranking of the set of objects with which they first entered the cache. Object references were treated as before, except that appearing as a leaf was equivalent to not appearing in the cache. If a leaf reference already

appeared in the cache as an object or as a leaf, then its status did not change when the ordered pair was appended to the output stream. Remaining details of the compression algorithm and the exact technique used to purge the cache are straightforward.

### *6.1.4 Equivalence*

There are a number of minor problems with this compression algorithm. Equivalence may be lost for a paged virtual memory because of two reasons. First, unless the static groupings are extremely successful and a large percentage of core is utilized, useful objects will not be tightly packed. The cache measured the sum of the in-core sizes of the objects. This number will differ from the total number of pages actually containing the objects because not all objects on these pages appeared in the cache simultaneously. Compounding this problem is the fact that all pointers in the paged virtual memory simulation occupied two words. Many objects required twice as much disk space as core space. However, the cache size in the compression algorithm was computed assuming all objects were in core.

Another drawback of the compression scheme is the fact that it was object-oriented instead of page-oriented. While this choice allowed the compressor to operate independently of any initial placement, it may have provided a slight benefit to the paged virtual memory. Consider an object straddling a page boundary. Although the emulator may have accessed portions of the object lying on different pages, the compressor would have discarded all but a couple of references to the object. In most cases, only one page would be required during a simulation of the paged virtual memory. The LOOM simulator, however, would be forced to swap in both pages in order to transfer the object into core. Two factors limited this advantage. First, there were eleven objects on an average page and only one of these could straddle a page boundary. Secondly, grouping was done in an attempt to concentrate references to a small set of pages. It may have been the case that the required but unfetched page contained objects referenced at about the same time the straddling object was accessed.

## 6.2 Three Execution Sequences

Three lengthy execution sequences were monitored to produce reference streams that were compressed by the algorithm described in the preceding section. Between 960K and 2280K bytecodes were executed in each trace, causing the emulator to generate three to eight million references. These traces were reduced to lengths between 40K and 80K reference pairs. The compression ratio was roughly 100:1.

Execution sequences were chosen by selecting actions the user or system would typically invoke. Many distinct types of computations were performed in a very short time span in order to produce a reference trace suitable for exercising a virtual memory.

The first trace involved the creation, initialization, and scheduling of a five-paned window called a Browser that was subsequently used to display the source code for a CompiledMethod. Message

dictionaries in various classes were queried in order to determine the presence or absence of a particular CompiledMethod. The entire display was cleared and all windows were cleared, framed, and titled. Finally, the decompiler was run on a CompiledMethod in order to produce source code.

In the second execution trace, source code from a particular protocol within a class was "filed out." This process entailed creating a new disk file, finding the source code for each CompiledMethod defined by this class-protocol pair, and writing the resulting text stream into the new file. A unique string was created and authenticated, the instance count of a class was calculated, and both the compiler and decompiler were invoked. For one class, the set of CompiledMethods that sent a particular selector was determined. Finally, an error message was sent to an object. The debugger was then entered by expanding the window created by the preceding error message.

The third compressed trace was twice as long as the other two and involved heavy user interaction. Operations for inserting, cutting, and pasting text were performed. Commands for undoing the last operation, canceling all operations, evaluating an expression, and invoking the compiler were input by selecting the appropriate sections of a pop-up menu. Scrolling and thumbing operations, which modified the portion of the text visible in a window, were also accomplished. In addition, a new class was defined and installed, a global variable was declared, and an existing disk file was opened. The contents of this file were read and compiled in an operation known as "filing in." Finally, the set of classes that responded to a particular message selector was determined.

## 6.3 Reference Spread

Two measures of the spread of the references to each object were obtained in order to analyze the compressed traces as well as to evaluate the compression algorithm. This algorithm successfully removed unneeded references and yielded compressed reference traces suitable for driving virtual memory simulations.

In these measurements, only object references were examined; leaf references were neglected. Consider an object A. Assume A was first referenced as an object on the $i^{th}$ reference pair and last referenced as an object on the $j^{th}$ reference pair. Define the *thin spread* of A to be j-i. Notice that if A were only referenced once, it would have a thin spread of zero. If A were referenced twice and the two pairs were consecutive, then A would have a thin spread of one. In the first two traces, at least one-fourth of all objects appeared only once. The median thin spread for all three traces was one. Leaf references caused many duplicate object references, which in turn gave rise to nonzero medians. Mean thin spreads were at least three orders of magnitude larger than the medians (Figure 6-1).

These facts indicate the size of the 10K cache for the compression scheme was adequate. The ratio of cache size to mean object size was 539. If the 10K cache size were too small, most objects would appear quite often and most thin spreads would be approximately some multiple of 539. This did not occur. Many objects were involved in only a particular portion of the entire computation and appeared in the compressed trace a small number of times. Each reference pair in the compressed

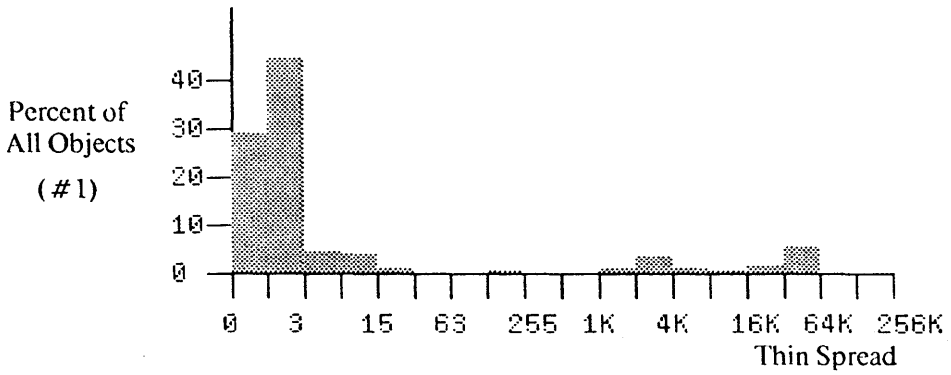


Figure 6-1 Thin Spread
(distance between first and last references)

traces would therefore be likely to cause faulting and possibly purging in a later virtual memory simulation. The compression algorithm was effective at eliminating redundant references and on this level was a clear success.

Large thin spreads, which raised the mean well above the median in all three traces, were caused by objects that spanned major portions of the execution sequence. Not all subcomputations were disjoint. Therefore, in addition to performing major context shifts, the virtual memory simulators were also compared as to their ability to predict and retain collections of objects used intermittently during the computation.

A second measure, *fat spread*, was also obtained. Let $s_k$ be the disk size of the object portion of the $k^{th}$ reference pair in the reference stream. Then the fat spread of the above object A would be

$$s_{i+1} + s_{i+2} + \cdots s_{j-1} + s_j.$$

Distances were rounded up to the nearest 256-word page. In all three reference traces, the median fat spread was one page. Average fat spreads were two to three orders of magnitude larger than the medians, falling into the 350 to 1400 page range. For the same reasons as above, low medians and much larger means are again indicative of a successful compression technique.

# 7. Grouping and a Paged Virtual Memory

This chapter begins with a concise description of the simulation of a paged virtual memory. The effects of static grouping on the dynamic performance of such a configuration are then presented and discussed. Finally, the predictions of paging behavior based upon the static analysis of the initial placements are evaluated. Appendix C presents a subset of the numerical information used in the following discussion.

## 7.1 Simulating a Paged Virtual Memory

In the simulation of a paged virtual memory, core was completely devoted to object space. No space was reserved for the table of in-core pages, purging information, or other memory management data. This was done so as not to constrain the comparison of actual paged virtual memories with LOOM. In Chapter 9, the requisite overhead as a function of core size is determined. The mapping between equivalent amounts of object space for the paged virtual memory and the object space for LOOM is then established.

The paged virtual memory simulator was a two-level, demand-paged hierarchy with no restrictions on the placement of disk pages in core. Core was assumed to be empty initially. Given a reference pair, the leaf reference was neglected while the object reference and a field offset were used to calculate the accessed page. Space for dynamically created objects with new UIDs was carved from the bottom of the free block at the high end of secondary memory. New objects with previously used UIDs were assigned to the location previously granted to that UID. Contents of the fields of objects were not maintained by the simulator. All that was necessary was a mapping from the set of UIDs to starting disk addresses. A strict LRU purging policy was employed because of its multiple simulation implications. Since LRU obeys the subset property [MATT], all core sizes could be simultaneously simulated in a one-pass sweep of the compressed trace. The one statistic obtained for this virtual memory was the mapping from core size to the number of page faults.

## 7.2 Results

The realizable grouping algorithms caused similar improvements in the performance of the paged virtual memory. Each of these strategies substantially reduced the number of page faults that occurred under the ungrouped, random arrangement. The particular grouping scheme was not critical; the fact that objects were grouped was the key factor in determining performance.

As previously noted in the static analysis of the groupings, initial placements generated from similar grouping strategies were similar. Initial placements in distinct categories were substantially different. A slight modification of this claim is applicable to their dynamic faulting behavior. Groupings generated by similar schemes were indistinguishable with respect to faulting rates. While categories

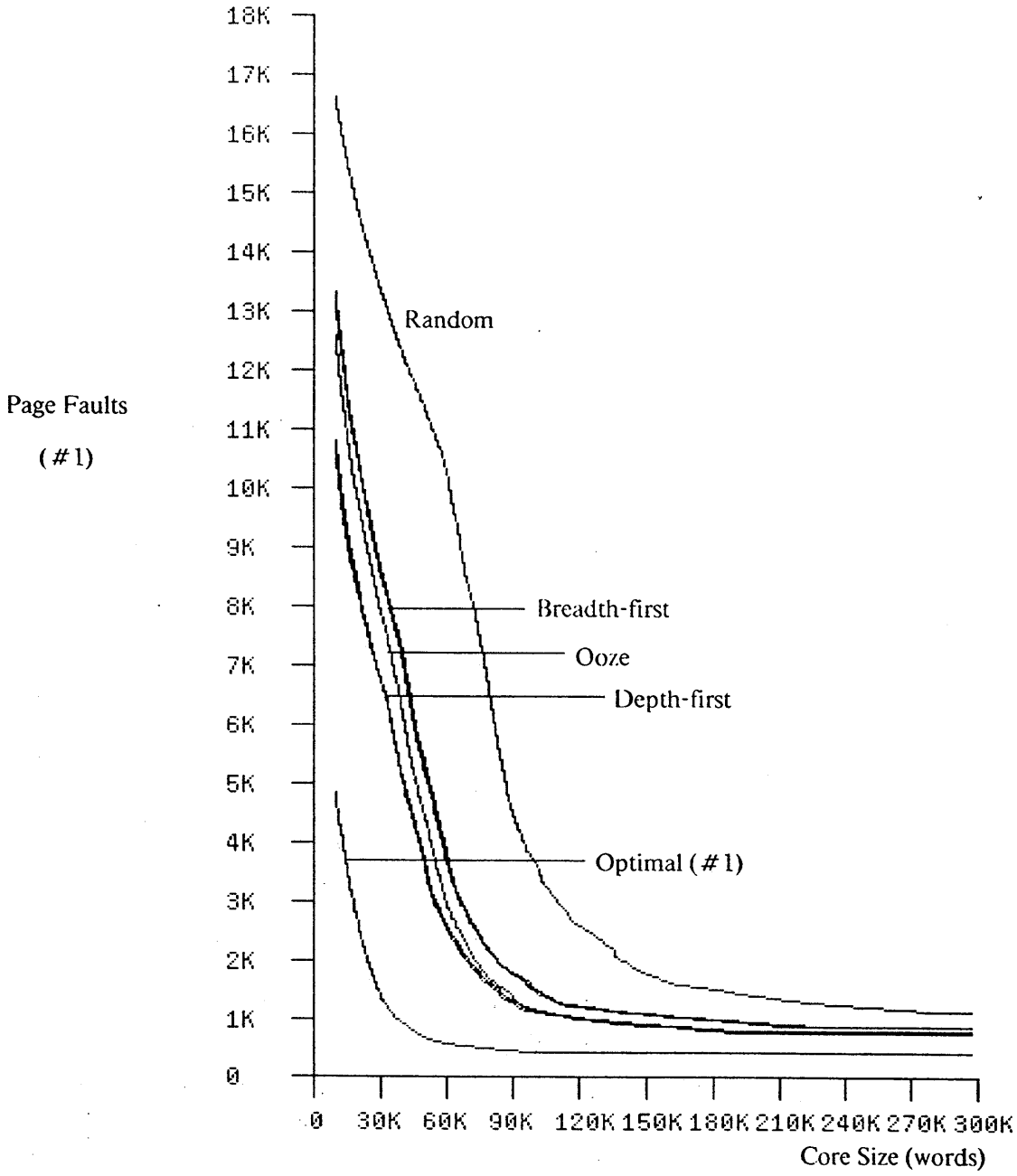


Figure 7-1 Parachor Curves for a Paged Virtual Memory

had distinct performance characteristics, their relative ordering and the degree to which they differed depended on core size.

Optimal groupings consistently outperformed all other initial placements (Figure 7-1). In all three traces, the best-to-worst ordering for small core sizes (20K) was optimal, depth-first, OOZE, breadth-first, and finally random. The seven middle groupings formed a narrow band in a much wider region bounded by the optimal and random initial placements. These groupings will hereafter be referred to as the *band groupings*.

Grouping algorithms may be evaluated by how closely their arrangement approximates the optimal initial placement. For a given core size, assume the optimal grouping causes $f_0$ fewer page faults than a random grouping. Assume another grouping causes $f_a$ fewer page faults than the same random initial placement. Define the *page fault reduction ratio* to be $f_a/f_0$. This number measures the degree to which a grouping algorithm has succeeded, assuming the optimal and random groupings are the best and worst possible arrangements, respectively. The page fault reduction ratio is essentially independent of the number and nature of the dynamically created objects. An unqualified version of this claim holds for tasks that do not cause the virtual memory to purge any information; the qualification must be added to cover smaller core sizes that force purging to occur. Minimum and maximum page fault reduction ratios for the band groupings are presented in Table 7.1. The success of the grouping schemes varied from moderate to good. Consistently coming closer than 75% of the optimal, however, will probably be extremely difficult to achieve.

| Core Size | 20K | 60K | 100K | 140K |
|---|---|---|---|---|
| Trace #1 -- Low | 35.0% | 65.9% | 67.0% | 56.0% |
| Trace #2 -- Low | 39.1% | 63.5% | 50.5% | 45.6% |
| Trace #3 -- Low | 37.7% | 61.9% | 70.5% | 61.4% |
| Trace #1 -- High | 54.6% | 80.0% | 79.3% | 69.7% |
| Trace #2 -- High | 57.7% | 78.2% | 77.8% | 79.1% |
| Trace #3 -- High | 58.7% | 75.8% | 81.8% | 74.4% |

Table 7.1 Extrema for the Page Fault Reduction Ratios

Within the narrow paging performance band, the ranking of the categories of grouping algorithms depended on the particular trace as well as the size of core. For small core sizes, the best-to-worst ordering was depth-first, OOZE, and then breadth-first. This advantage of the depth-first schemes declined for larger core memory sizes. In the first trace, the depth-first and OOZE groupings had similar faulting rates for core sizes above 90K. OOZE was actually better than any depth-first grouping in the second and third traces for all core sizes above 60K. Even though the depth-first arrangements outperformed the breadth-first groupings in the first trace, for large core sizes in the second and third traces, their performance was strikingly similar. These reversals precluded a definitive evaluation of the grouping techniques that was independent of core size.

Let *core utilization* be the percent of core containing objects that were touched. An underlying assumption is that if an object were touched, than all its fields were touched. While this hypothesis

is not true, the *actual* core utilization may be estimated by multiplying the core utilization by the mean fractional utilization of objects.

For large core sizes, core utilization and not purging policy was the important factor in determining the number of page faults. Dynamically created objects, which utilize 100% of the disk pages they occupy, can blur the utilization differences between initial placements. Since the total size of all new objects is known, their effect could be removed from the average utilization. Table 7.2 presents core utilization ratios for very large core sizes.

| | BFS | DFS | OOZE | Random | BFS | DFS | OOZE | Random |
|---|---|---|---|---|---|---|---|---|
| Trace 1 | 48.9% | 55.6% | 53.8% | 37.7% | 33.8% | 40.0% | 38.3% | 24.4% |
| Trace 2 | 53.8% | 53.2% | 65.3% | 47.0% | 44.4% | 43.8% | 56.3% | 37.8% |
| Trace 3 | 58.3% | 53.2% | 61.8% | 46.8% | 40.7% | 35.8% | 44.3% | 30.2% |

Table 7.2 Core Utilization Including (left) and Discounting (right) New Objects

Assuming eleven objects per page and a trace that referenced only a small fraction of all old objects, a truly random initial placement would utilize roughly one-eleventh (9.1%) of core. Observed values for the random grouping were higher because the grouping was not completely random and the trace referenced a sizable fraction of the objects. A number of pages contained more than one accessed object. Core utilization for the optimal initial placements was by definition 100%.

While a 100% utilization of core is theoretically possible, the maximum attainable value is probably much less. Empirical measurements of the fractional utilization of all objects indicate that between 39% and 51% of the fields of objects are touched during a computation 144K references in length. Since roughly three-fourths of all objects had a static reference count of one, any strategy that groups objects solely on the basis of connectivity and does not distinguish between heavily used and rarely used fields will have a utilization of core by old objects approximately equal to the fractional utilization of objects. This limit has been approached by the nonrandom groupings. Although these grouping schemes have had some effect, there may be room for improvement. Surpassing this bound requires the a priori knowledge of heavily accessed fields and the invariance of high-access tendencies. Even though utilization can be increased if this set of fields is on a per object basis, the grouping algorithm is much simpler if the high-access fields depend primarily upon the class and not the specific instance of the class. The dynamic statistics used to influence one depth-first and one breadth-first grouping were a first attempt in this direction. Further investigations into this area, however, are beyond the scope of this thesis.

## 7.3 Analysis of Predictions

The predictive capabilities of the static analysis of the initial placements may be evaluated in light of the performance of a paged virtual memory. Certainly the similarity of the paging rates for similar initial placements was upheld. However, except for the dismal performance of the random initial placement, the distinctiveness of the band initial placements was not strongly evident. Breadth-first groupings clearly outperformed the random initial placement and predictions limiting their degree of improvement were upheld. Low fractional utilization of objects prevented the

breadth-first initial placements from becoming the best arrangements. The static analysis did not predict reversals in the ordering of grouping schemes. These reversals were due to a performance degradation intrinsic to the depth-first groupings. Since the relative loss of performance of the depth-first algorithms for larger core sizes was evident in all three reference traces, this decline seems to be related to the traversal type (depth-first) and the amount of purging but not to the particular traces. However, the exact nature of this degradation was not investigated.

The retained fraction measure for the optimal groupings was able to accurately predict the paging performance for the paged virtual memory simulations. OPT1 predicted a best-to-worst ordering of depth-first, OOZE, breadth-first, and random, which in fact was the ordering for small core sizes in the first reference trace. As the page size was increased to 256 and 512 words, the depth-first and OOZE initial placements swapped positions. For simulations of the first trace with large core sizes, this reversal in performance actually occurred. Retained fractions with respect to OPT2 and OPT3 ordered the groupings as follows: OOZE, depth-first, breadth-first, and finally random. While the first two grouping categories were swapped in the ordering defined by paging performance with small core sizes, this ranking was confirmed for moderate and large core sizes for the second and third reference traces. As the page size was increased to 256 and 512 words, the retained fractions for the depth-first and breadth-first groupings became similar. This similarity caused the breadth-first groupings to catch and at times to outperform the depth-first initial placements with regard to paging rates for moderate and large memory sizes.

One explanation for the correlation between retained fraction values for larger page sizes and paging performance for larger primary memory sizes is the reduction in purging. When the size of core is increased, fewer pages are discarded. Individual pages and groups of pages tend to remain in core longer. Inter-page pointers become as important as intra-page pointers because their dynamic distances are not distinguishable. Enlarging the page size in the retained fraction measure has a similar effect. Not only are intra-page pointers considered in the analysis; short inter-page references also contribute to the final value.

# 8. Grouping and LOOM

This portion of the thesis describes the LOOM simulator and contrasts it with the actual system. The dynamic performance of the simulator under a number of policies and configurations is presented. Finally, performance predictions based upon the static analysis of the initial placements are examined in the light of actual results. Appendix C contains the collection of statistics and data on which the following discussion is based.

As in the case of the paged virtual memory, the realizable grouping strategies improved the performance of LOOM. However, there were two significant differences between the simulations of LOOM and the paged virtual memory. These two features arose because of the object-swapping characteristic of LOOM and the typically short stay of disk pages in the in-core buffer. First, the performance differential between the random initial placement and any realizable arrangement was not as great in LOOM as it was in the paged environment. LOOM was able to perform rather well in the face of the adverse conditions caused by the random initial placement. Secondly, the performance of LOOM was sensitive to the grouping strategy used. Different grouping strategies realized different performance benefits.

Another result gleaned from these simulations is the importance of a grouping strategy to LOOM-like systems. Other virtual memory policies and parameters play a secondary role and are essentially independent of the initial placement.

## 8.1 Simulating LOOM

For the reference pair (A, B), the LOOM simulator first determined if the object A were in core. If not, then an *object fault* would occur and the object A would be swapped into core. Any leaf for A would be discarded, since the leaf would have been expanded and replaced by the complete object A in the actual LOOM system. If B were already in core, either as an object or as a leaf, then nothing further would be done. Otherwise, a leaf for B would be created, since by definition the referenced field in A is assumed to have been a lambda. Call the resolution of such a pointer a *lambda fault.* Resolving such a pointer requires that the 32-bit pointer that references B be found in the disk representation for the object A. Note that only object A need be accessed. LOOM does not need to access an object in order to create a leaf for that object. While many lambda faults cause page faults, the necessary page may already be in the disk buffer. Depending on the nature of the reference and the previous state of the in-core object A, the clean-dirty bit for A would then be set appropriately.

Core memory and the disk buffer were initially empty. Both objects and pages were fetched on demand. The disk buffer employed a FIFO page replacement algorithm, while a clock purging scheme was used for primary memory.

Continuous compaction of core was assumed. If enough space existed for an object or a leaf, then it would be placed into core without causing any further disk interactions. Whenever the remaining free space could not accommodate a new object or leaf, the purging algorithm was invoked. A clock scheme was used for purging, which was done incrementally as needed. An index cycled through the table of in-core objects. There were three possible cases: a touched object, an untouched object, or a leaf. Touched objects were marked untouched, untouched objects were contracted to leaves, and leaves were simply discarded, since by assumption they were clean. Clean object contractions caused no disk references, but dirty contractions forced the object to be written to the disk.

This cleanliness remark deserves clarification. The one important piece of information contained in a leaf is the change in the disk reference count of the object that it represents. An assumption was made that a realizable algorithm could be found and employed that makes this *delta reference count* zero when the leaf is purged. When this number is zero, the actual reference count located on the disk is correct and need not be updated. Since it contains only a disk address, the leaf is clean and may be discarded without any disk interactions. One algorithm is very simple: always assume the final disk reference count of an object will be one. When a dirty object is contracted, the disk reference count may be set to one and the delta reference count adjusted appropriately. Since three fourths of all static reference counts are one, this algorithm is likely to succeed. Another possibility assumes that some constant fraction of all in-core references to the object eventually become disk references to the object. Other bases for predictions surely exist. However, the analysis of such algorithms is beyond the scope of this thesis. All these predictive schemes depend upon the contraction of dirty objects in order to set the disk reference count to the best guess of its final value, because the disk reference count of these objects may be set without incurring any further paging penalties. Clean objects present a problem, since their disk reference count cannot normally be adjusted when they are contracted without causing a page fault. Many of the clean objects are CompiledMethods, which typically have a stable reference count. These kinds of system objects do not pose serious problems. For other clean objects, the best policy is probably not to tamper with the disk reference count until the in-core reference count of the leaf changes to zero.

The LOOM simulator did not maintain the contents of the fields of all objects, nor did it maintain the starting address of all in-core objects. Instead, the simulator kept a table of in-core objects as well as a list of pages in the disk buffer. Six statistics were maintained for each LOOM simulation. Object faults, lambda faults, clean contractions, dirty contractions, disk buffer hits, and disk buffer misses were all counted. In addition, the simulator carefully monitored the utilization of pages in the disk buffer. For the purposes of this measurement, if an object were dirty, then all its fields were assumed to be dirty. If a portion of an object residing on a particular page were touched, then all the fields of that object on that same page were assumed to be touched.

## 8.2 Differences Between the Simulation and LOOM

The simulator was a fairly detailed model of the actual LOOM virtual memory. Pointers were compressed, lambdas and leaves were used, and new objects were considered immature and hence not immediately written to disk. On the other hand, a number of simplifications were made.

First, hints were not used to decide which pointers to resolve when an object was swapped into core. Instead, a simple dynamic decision was made. If the 32-bit pointer referenced an in-core object or a leaf for that object, then the pointer was resolved. Otherwise, the pointer became a lambda. In fact, the contents of all pointer fields were assumed to follow this pattern. A pointer was a lambda if, and only if, neither the referenced object nor its leaf were in core. Creating or destroying a leaf would automatically change all in-core references to this object. This effect was not important, however, since roughly three-fourths of all objects had a total static reference count of one. Moreover, not all objects were in core simultaneously. By definition, in-core reference counts were less than the total reference count.

In-core as well as disk reference counts were not kept by either simulator. The compression algorithm eliminated much of the information required to maintain these two sets of counts. Except for the reuse of a UID, neither simulation knew that an object had been destroyed. In the paged virtual memory, an inaccessible object would continue to occupy some fixed number of disk fields. In the LOOM simulation, such an object would eventually contract, possibly mature, and occupy a portion of the disk. Since the net result was similar in both simulators and UIDs were frequently reused, the inability to detect object destruction was not critical. Neglecting this aspect of memory management allowed enormous gains to be realized by a compression of the actual reference trace. A complete analysis of different types of memory allocation and deallocation schemes is beyond the scope of this thesis.

Another minor difference was the omission of forwarding markers in the LOOM simulator. An implicit assumption was that the disk name of the object always mapped into its in-core name. Since forwarding markers are used only when immature objects are forced to the disk, the effect arising from this omission may safely be neglected. Finally, contraction of objects referring to immature objects was not prohibited. LOOM would either prevent the contraction or mature the referenced object. Again, the effects of this simplification were not substantial.

## 8.3 Results

Each compressed reference trace drove a LOOM simulator with a buffer size of 8 disk pages and a fixed core size. The smallest core size was 20K words. Simulations were repeated with larger core sizes using increments of 20K words until no purging was required. Unlike the LRU policy utilized in the paged virtual memory simulation, the LOOM purging scheme did not obey the subset property and simulations of different sizes could not be run simultaneously.

### *8.3.1 Results Independent of Grouping*

Four of the six events counted during each LOOM simulation were independent of the grouping scheme used. For a given trace, the number of object and lambda faults, as well as the quantity of clean and dirty object contractions, depended only upon the size of primary memory. All four counts were monotonically nonincreasing for the set of core sizes simulated.

The number of object faults was always greater than the number of lambda faults. On the average, less than one lambda was resolved for each object that was swapped into core. Although this average is important, the general distribution of the number of resolved lambdas per object ought to be examined. Predictive schemes for deciding which pointers to resolve when an object is transferred into core can then take this information into account.

No such simple relation was evident between the numbers of clean and dirty contractions. For very small core sizes, the ratio of clean to dirty objects that were purged varied from two to six. When the size of core was large enough to cause only a small amount of purging, most of the contracted objects were dirty. In these large-core simulations, no purging equilibrium was established. The low clean-dirty purge ratio reflects a transient caused by the particular purging scheme since purged objects were a random sample of all in-core objects. In simulations of moderate and small core sizes, however, numerous purge cycles created a reasonable balance between marked and unmarked objects. The associated clean-dirty contraction ratios reflect not the clean-dirty ratio of in-core objects but the characteristics of the set of objects that had not been recently accessed.

Data supporting the success of the purging algorithm may be found by comparing the change in object faults with the change in object contractions as the size of core is decreased. If these two changes are comparable, then most object contractions later gave rise to an object fault. Such results would indicate that the purging algorithm was not performing well. On the other hand, if the number of new contractions is much larger than the change in the number of object faults, then most of the object contractions did not cause a subsequent object fault. The purging algorithm would be selecting the appropriate objects to discard. For example, when the core size is decremented by 20K from the smallest size for which no purging occurs, the ratio of the changes in object faults and object contractions was less than one-tenth for all three traces. Most of the contracted objects were not subsequently accessed. However, when the 40K to 20K transitions are considered, this range shifts upward. These ratios, which fall into the interval from two-thirds to six-sevenths, indicate the presence of thrashing. A majority of the contracted and purged objects were later needed.

Two of the six event counts, the number of hits and misses for pages in the disk buffer, clearly depended upon the grouping. Intuitively, for a given reference trace, the sum of the hits and misses for the disk buffer ought to be independent of the initial placement. However, this sum was not constant. Objects straddling pages caused a slightly different number of pages to be accessed for each arrangement.

### *8.3.2 Paging Performance*

Whenever the LOOM simulation interacted with the disk, a *page reference* was generated. Page references were caused by all object faults, lambda faults, and dirty contractions. Some of these page references named pages in the disk buffer and caused a *hit*. If the specified page were not in the buffer, then a *miss*, also called a *page fault*, occurred. Let the *hit rate* be the fraction of page references satisfied by the disk buffer. While this rate for the most part increased for larger primary sizes, a few minor anomalies occurred. Consider the third reference trace. In all three 40K to 60K transitions in the depth-first groupings, the hit rate and the number of page faults declined. Substantial reductions in the number of object and lambda faults caused fewer page references and allowed this set of conflicting events to occur.

One measurement obtained from the LOOM simulations was the number of page faults caused by each grouping (Figure 8-1). As in the paged virtual memory simulations, paging performance upheld the similarities between initial placements generated by grouping strategies of the same type. However, there were no reversals between initial placements derived from different grouping categories. The seven middle arrangements were again found in a band situated in the wide region delineated by the optimal and random initial placements. Within this band, the depth-first groupings were the best in two out of three traces, while the OOZE initial placement had the best performance in the other simulation. Breadth-first arrangements did poorly in all three traces, outperforming only the random initial placement.

In order to determine the relative effectiveness of the chosen grouping strategies in different types of virtual memories, the LOOM page fault reduction ratios were calculated. One caveat that must be mentioned is the fact that the optimal initial placements may not be optimal for the LOOM simulations. Contraction of dirty objects and resolution of lambdas may have caused the derived initial placements to be non-optimal. Therefore, the faulting rates of our "optimal" grouping must be considered to be close upper bounds on the paging rates for a true optimal arrangement. Table 8.1 presents the maximum and minimum value of the page fault reduction ratios for the seven band groupings.

| Core Size | 20K | 40K | 60K | 80K | 100K | 120K |
|---|---|---|---|---|---|---|
| Trace #1 -- Low | 22.7% | 22.2% | 23.4% | 21.9% | * | * |
| Trace #2 -- Low | 40.6% | 38.0% | 39.5% | 40.8% | 40.4% | 40.5% |
| Trace #3 -- Low | 24.8% | 21.8% | 19.1% | 18.5% | 19.0% | 18.1% |
| Trace #1 -- High | 50.3% | 48.2% | 49.6% | 48.1% | * | * |
| Trace #2 -- High | 62.7% | 58.9% | 61.3% | 61.7% | 62.7% | 63.6% |
| Trace #3 -- High | 58.0% | 55.7% | 50.6% | 46.9% | 47.1% | 46.0% |

Table 8.1 Extrema for the Page Fault Reduction Ratios

Two differences stand out between the two types of virtual memories. Both the minima and maxima of the LOOM band were smaller than the corresponding values for the paged virtual memory. Secondly, the relative width of the LOOM band was larger. In the paged virtual

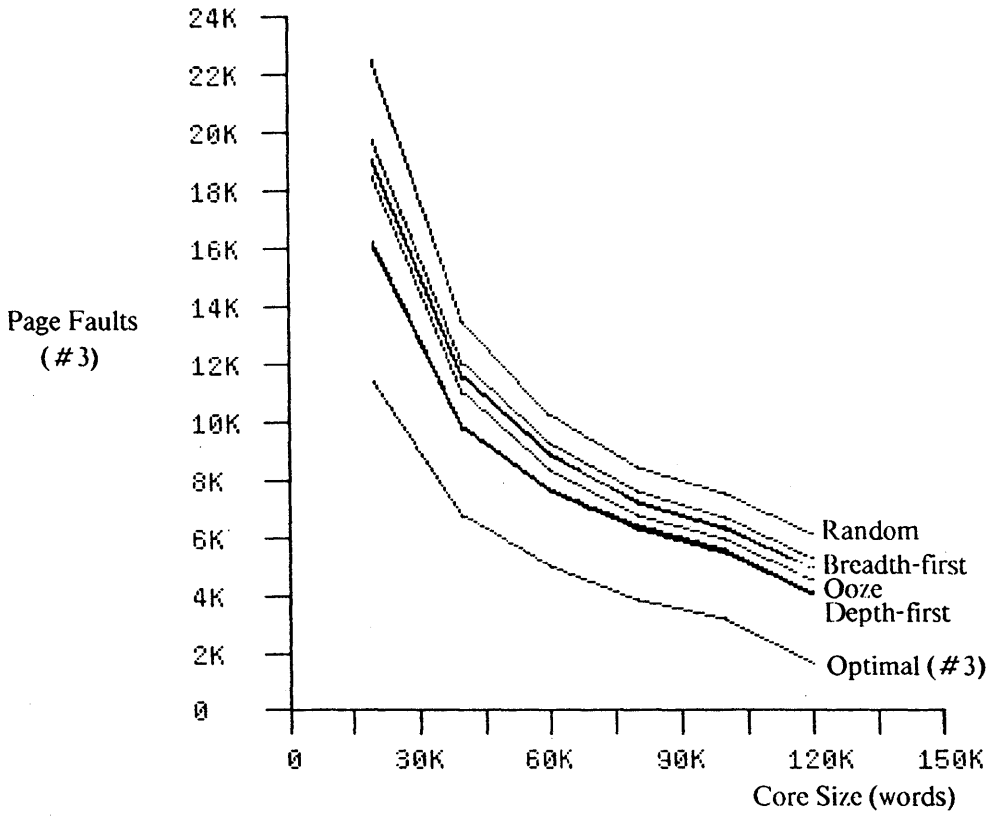


Figure 8-1 Parachor Curves for LOOM

memory, any reasonable grouping scheme substantially reduced the amount of paging from the level caused by the random initial placement. Differences between grouping schemes were not too significant. In LOOM, the amount of improvement in paging performance depended heavily on the type of grouping. Although LOOM was more sensitive to the type of the grouping scheme, grouping had less of an effect on LOOM than it did on the paged virtual memory. In the latter case, pages tended to remain in core for long periods. The critical measure was core utilization. Pages remained in the disk buffer in LOOM for only brief periods of time. The magnitude of the effect of a grouping scheme depended upon the short-term reference behavior of the Smalltalk virtual machine and the probability that the generated object and lambda faults would be successfully handled by the disk buffer.

### *8.3.3 Page Utilization and Cleanliness*

The LOOM simulation maintained the set of objects read from or written to a page in the disk buffer. When a page was flushed, its usage and cleanliness were tabulated. These fractions gave rise to the disk page *utilization* and *cleanliness* distributions.

Average page utilization values ordered the initial placements in the exact ordering defined by paging performance. Means for the realizable grouping schemes were bunched in the 15% to 25% range and increased only slightly for larger core sizes (Figure 8-2). Page utilization rose with memory size for two reasons. First, static grouping algorithms attempted to place on a page the set of objects that were likely to be needed at the same point of a computation. Little attention was paid to the dual problem of sets of objects likely to be purged at the same time. Larger core sizes increased the relative amount of faulting with respect to purging. Secondly, smaller amounts of purging meant that each disk page on the average remained in the buffer for a longer period of time. Longer residence times increased the probability that another object on the same page would be accessed before the page was flushed.

A substantial number of page transfers actually transferred only a single object. In the first reference trace with a depth-first grouping, for example, the median number of referenced objects per page was only one. The mean was slightly higher, reaching a maximum of three when no purging occurred.

Set apart from the realizable groupings was the performance of the optimal groupings, which utilized 30% to 50% of the pages swapped into the buffer when purging occurred. Unlike the realizable arrangements, the optimal initial placements drastically increased their mean utilization to the 60% to 70% range when there was no purging. Resolution of lambdas prevented a full 100% disk page utilization for these optimal initial placements. This low utilization value indicates that many lambda faults were not satisfied by the disk buffer and hence caused a page fault. Since the optimal arrangements had the highest mean page utilization values, a higher fraction of these lambda faults would undoubtedly have caused page faults in non-optimal groupings. As far as page utilization is concerned, lambda resolution was much more harmful than purging. Efficient

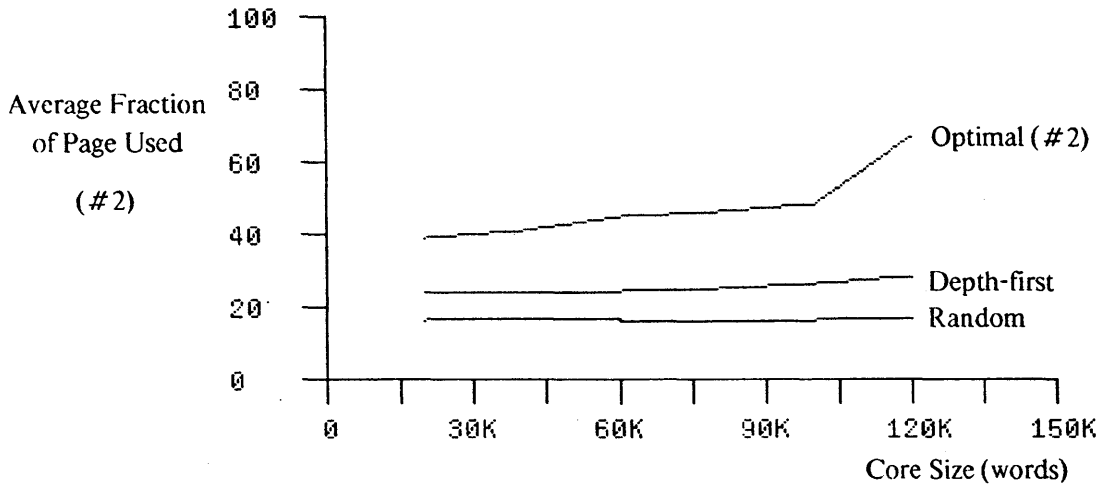


Figure 8-2 Mean Page Utilization

algorithms determining which pointers to resolve when an object is swapped into core can thus have an important impact on paging performance. While purging had little effect on the page utilizations achieved by realizable groupings, it had a substantial impact on the optimal utilizations. As better grouping schemes that accommodate faulting are found, the effect of purging will become more critical to the overall performance of the virtual memory.

These utilization figures indicate that page references caused by purging or lambda resolution ought to be treated differently than page references caused by faulting. One possible distinction is to keep any page used only for lambda resolution and purging at the lowest priority level in the disk buffer. Instead of elevating such a page to the top of the FIFO/LRU ordering, the page would remain at the bottom and would be the first to be discarded. The disk buffer working set would not be easily clobbered, because at all times at most one page in the disk buffer would have been used for purging or lambda resolution.

Between 60% and 85% of all pages were clean when flushed from the disk buffer in simulations that involved purging. These cleanliness fractions are much larger than the 10% to 60% interval reported for page-swapping virtual memories [KUCK]. Differences are due to the fact that Smalltalk permanent objects tend to be read-only. Dynamically created objects, which are by definition dirty, usually perish before being purged to the disk. For the most part, dirty pages in LOOM contained only a single dirty object. Exceptions occurred when dynamically created objects were matured and simultaneously written to the disk. The cleanliness of disk pages did not significantly depend on either the grouping involved or the size of core employed. This independence arose because both the initial groupings and the purging algorithm made no attempt to increase the number of dirty objects per flushed page.

Decreasing the number of dirty pages flushed from the disk buffer is important because it eliminates many double page faults and increases the lifetime of a write-once secondary storage medium. As long as a full disk page is the physical unit of transfer, these benefits arise in both page-swapping and object-swapping virtual memories. Simple attempts at rectifying this situation include grouping typically clean objects with similar objects, as well as grouping typically dirty objects with other normally dirty objects. A straightforward modification to the purge routine would be the determination of the in-core status of objects on a page in the disk buffer when a dirty object is contracted onto that page. If any of these objects are both unmarked and dirty, then purging them immediately would tend to reduce the number of dirty pages flushed from the disk buffer.

### *8.3.4 Effect of Core Purging Policy*

Many LOOM simulations were repeated for one or two initial placements with different virtual memory parameters and policies. These additional runs provided data that yielded the relative importance of grouping as well as the effect of grouping on a particular parameter or policy change. A complete factorial design was not attempted because of the prohibitive number of experiments it requires.

Since the grouping algorithms were geared towards faulting tendencies as opposed to purging requirements, the purging policy was modified to take into account the initial placement. This change caused mixed results in performance that were independent of the grouping strategy. Unless grouping schemes sensitive to purging issues are employed, the results indicate that the purging scheme may act independently of the initial placement without incurring a substantial performance penalty.

This change in the purging policy was effected by modifying the ordering of UIDs through which the purge index cycled. Since the randomizing function that produced the hash initial placement was invertible, the ordering of objects in secondary memory in this grouping could be easily determined at run time. This ordering, instead of the linear sequence (1, 2, 3, . . . ), was consistently used when the purge status of objects was checked. Call this scheme the *random purge policy* and the original scheme the *linear purge policy*. The random policy was chosen in an attempt to purge objects on the same page at the same time. In addition to repeating the hash simulations, one depth-first grouping was also rerun for all three compressed reference traces. Since the depth-first grouping was strongly correlated with the linear UID ordering utilized by the original purge policy, these additional simulations would determine if changing the purging policy had distinct effects on different static arrangements. If the new purging policy worked as intended, the faulting rates for the depth-first and random initial placements would increase and decrease, respectively. Page utilization and dirtiness were expected to change in the direction opposite to that undergone by the page fault total.

The random purging scheme was modeled after the policy utilized by the Object-Oriented Zoned Environment (OOZE) virtual memory for Smalltalk-76. OOZE also purged objects according to their ordering on the disk. One shortcoming of the simulation policy is that immature objects forced to the disk were appended to the set of objects already on on the disk without regard to their UIDs. The random purge operating on the random initial placement thus reflected the ordering of old objects but not the ordering of dynamically created objects.

The effect of the random purging policy was essentially independent of the initial placement and the compressed reference trace (Figure 8-3). For small core sizes, which caused a substantial amount of purging, there was a 3% to 10% increase in the number of page faults. This difference decreased for larger core sizes. In fact, for the second and third compressed reference traces, the random purge outperformed the original policy by 1% to 5% for moderate to large core sizes. This new policy had only negligible effects on page utilization and cleanliness for both initial placements.

Different numbers of dirty contractions, object faults, and lambda faults were the prime causes of differing faulting rates. Contraction of roughly equal numbers of dirty objects had a similar effect on the overall number of page faults for both initial placements. However, the particular sets of objects discarded by the purge routine had a nontrivial impact on paging performance and served to distinguish the linear purging policy from the random purging policy. The relative change in the number of page faults was larger in magnitude for the depth-first arrangement than for the random

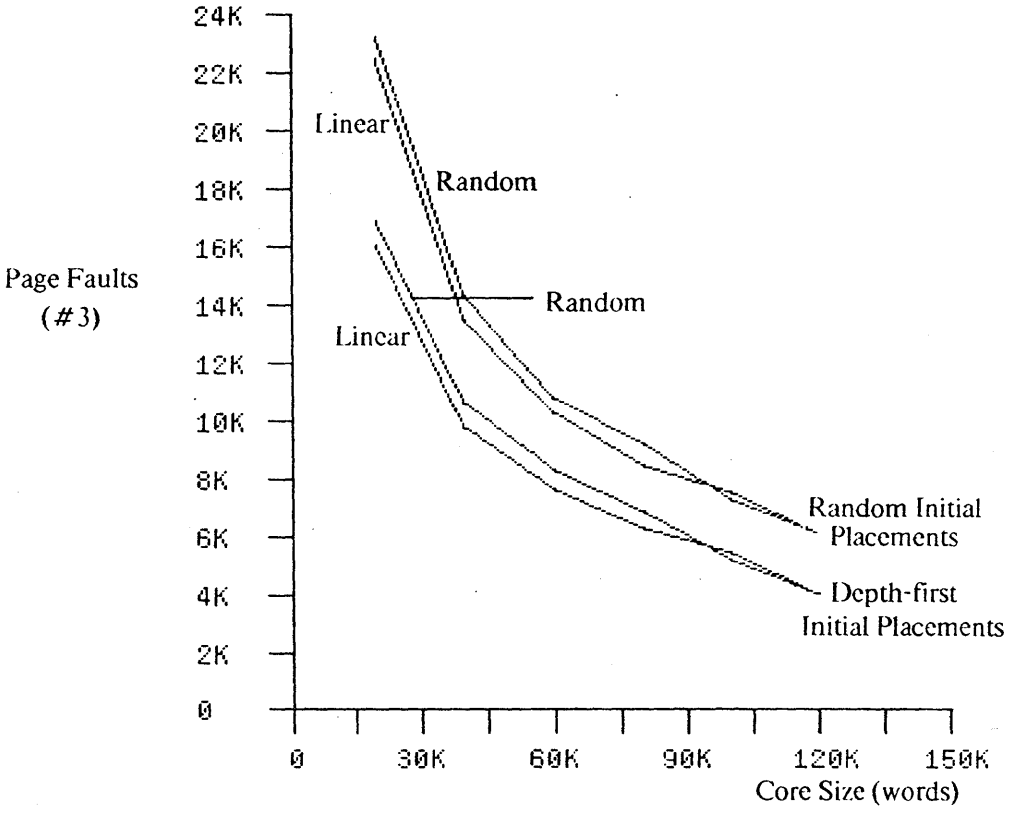


Figure 8-3 Effect of Core Purging Policy
on Parachor Curves for LOOM

initial placement. Absolute changes in faulting rates were roughly equivalent. One unconfirmed hypothesis drawn from these results is that random initial placements are not as sensitive to changes in the purging policy as are non-random initial placements. Because there was no true relationship between objects located on a single page in the random arrangement, modification of the purging policy tended to have a smaller relative effect.

The two implemented purging schemes are similar to a simple *bottoming* technique in which objects are purged from core according to the time at which they first appeared in core. If all objects are scanned in each purge cycle, then this technique differs from the previous two only in the permutation followed by the purge index. Analysis of enhancements to this type of purging algorithm, including special-casing new and old objects and partial instead of full core sweeps, is beyond the scope of this thesis.

Since the single attempt at grouping-directed purging did not realize consistent gains, these results do not rule out the possibility of such purging schemes. If more purging emphasis is placed on the grouping algorithm, it may be possible to improve performance by having the purging algorithm depend on the current mapping from objects to disk addresses. Otherwise, the purge algorithm should be free to choose objects to discard. For example, a purging policy could distinguish between clean and dirty objects. Contracting dirty objects could be delayed and perhaps entirely avoided. The important consideration is to purge the optimal set of objects regardless of the concomitant overhead. Flushing objects grouped on pages may reduce paging locally, but this strategy can have disastrous global effects. Since the optimal purging policy is unrealizable, efforts should be directed toward designing, simulating, implementing, and measuring realizable policies that investigate the local-global performance tradeoffs. Static groupings ought to be constructed to accommodate the purging policy in addition to the anticipated reference tendencies of the virtual machine.

### *8.3.5 Variable Buffer Size*

A second variation in the LOOM simulator was the size of the disk buffer. As the disk buffer increased in size, an equal amount of memory was removed from object space. Simulations for all three reference traces and all core memory sizes were repeated for 10 additional disk buffer sizes ranging from 2 to 32 disk pages (Figure 8-4). Each increase in buffer size corresponded to an equal decrease in the amount of primary memory reserved for objects.

The expected improvement in paging performance occurred for a range of larger buffer sizes. Performance declined when large buffers were coupled with small sizes of main memory. This effect was also anticipated and is due to the reduction in main memory reserved for objects. However, the computations were not lengthy enough to cause this phenomenon for any but the smallest memory sizes. The relationship between the optimum buffer size and the amount of main memory was not determined.

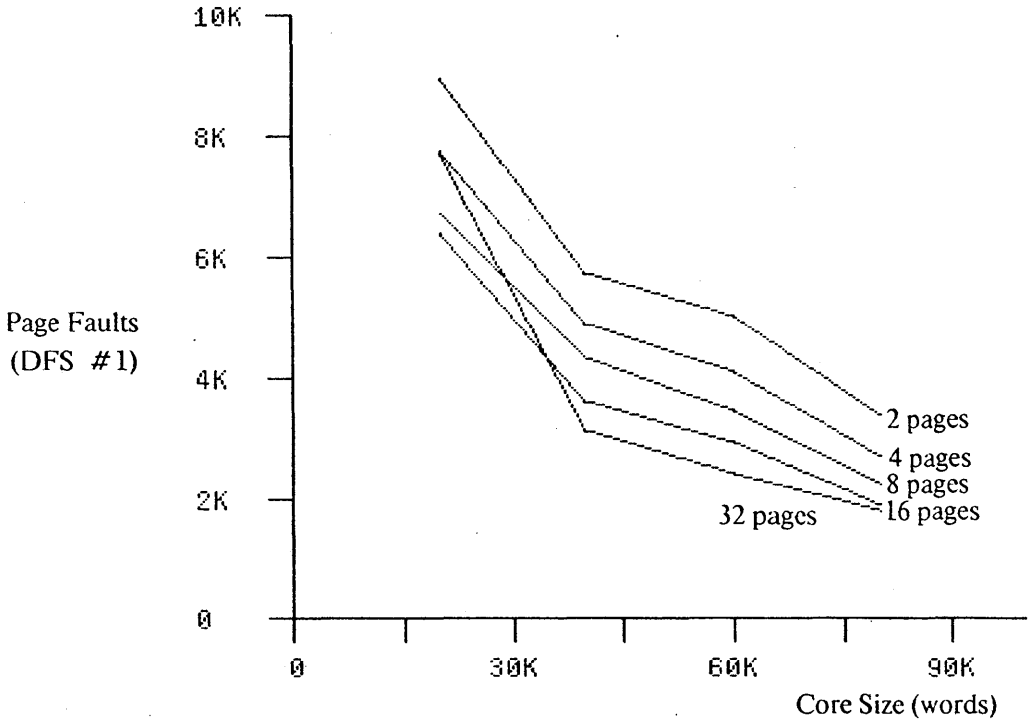


Figure 8-4 Effect of Disk Buffer Size
on Parachor Curves for LOOM

The number of page faults dropped in most of these additional simulations. Paging performance typically improved as the buffer size increased. Larger buffer sizes increased the average residence time for pages in the buffer and transformed many buffer misses into buffer hits. Smaller buffer sizes naturally implied the opposite. For all simulations involving buffers smaller than the initial 8-page buffer, paging performance declined. In the first trace, for example, there were 4% to 52% more page faults for buffers of size 2, 4, and 6. The magnitude of this decline increased slightly for larger core sizes. When purging was occurring frequently, smaller buffers moderately impeded the progress of the computation. However, for large memory sizes where purging was infrequent or nonexistent, small buffers acted as a bottleneck that severely limited the throughput of objects.

Very large buffer sizes coupled with tiny core sizes were the second cause of paging performance reduction. By reducing the amount of core reserved for objects, larger buffers caused more object purging, object faulting, and lambda resolution. All three actitivites in turn caused more page references. Large buffers increased both the hit rate and the number of page references. The net effect of these two conflicting tendencies depended upon the specific trace and the relative sizes of core and buffer. Consider a fixed core size of 20K words. In all three reference traces, the minimum number of page faults was achieved at buffer sizes larger than the original value of eight disk pages. As the buffer size increased, the number of page faults climbed and in two of the three reference traces eventually surpassed the original level.

Varying the buffer size had only slight effects on both the utilization and cleanliness of pages. The means, medians, and quartiles of these two distributions slowly climbed as the buffer size was increased. Consider the simulations of the first reference trace that had a buffer size of two pages. Average page utilization varied between 21% for a 20K core and 29% for an 80K core. When the buffer was extended to 32 pages, the average utilizations ranged from 28% to 34%.

Although the emphasis in this analysis has heretofore been placed solely on paging, there are two contributing factors that determine the performance characteristics of an object-swapping virtual memory. There is the obvious problem of page faults, which arise when a page reference is not satisfied by the disk buffer. Object faults, dirty contractions, and lambda resolutions also contribute to the total cost. At the very least, the computation at hand is momentarily suspended while the request made by the virtual machine is satisfied. The disk buffer is queried and data is eventually swapped between the buffer and core. Many of these requests also cause one or more page faults. The weighted sum of the costs for the disk-buffer transactions and the buffer-core interactions must be minimized subject to the constraint that the sum of the buffer and core sizes equals the available amount of primary memory. Tiny buffers are not optimal, since the buffer hit rate rapidly declines for small sizes. Choosing extremely large buffer sizes is also counterproductive, since it increases the number of contractions and object faults and may cause thrashing on two levels.

The size of the disk buffer is characterized by decreasing returns to scale. Doubling the buffer size from two to four pages had a substantial impact on paging performance. A similar increment of two pages from 10 to 12, or even a doubling from 16 to 32 pages, had a much smaller effect on the

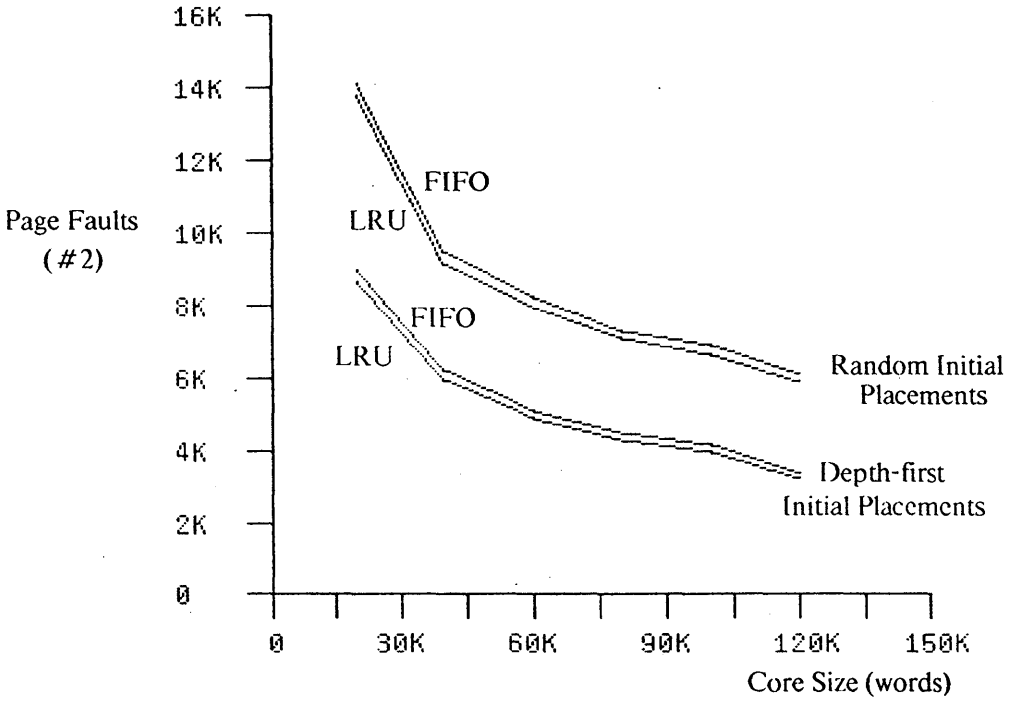


Figure 8-5 Effect of Disk Buffer Purging Policy
on Parchor Curves for LOOM

number of page faults. Because the size of the disk buffer trades off against the amount of core available for objects, larger buffers increase the number of object contractions, lambda resolutions, and perhaps page faults. For any given core memory size, cost criteria, initial placement, and expected set of reference tendencies, there is an optimum buffer size. The relationship between this optimum buffer size and the amount of available primary memory cannot be gleaned from available data, since the simulations represent only a handful of small core sizes. One expects the optimum buffer size to increase for larger primary memories. Whether this growth rate is linear, sublinear, nonexistent, or some combination of all three is currently an open question.

### *8.3.6 Effect of Disk Buffer Purging Policy*

Another policy changed in the LOOM simulator was the purging scheme for the disk buffer. Instead of a FIFO replacement scheme, an LRU purging algorithm was used. Simulations for the random arrangement and one of the depth-first initial placements were repeated for all core sizes and all three reference traces. The LRU scheme realized a small but consistent performance improvement.

The LRU scheme outperformed the FIFO scheme for both initial placements and all core sizes (Figure 8-5). For the depth-first grouping, this policy change realized a 3.8% to 6.1% reduction in the number of page faults. Slightly smaller relative reductions, in the 2.5% to 4.4% range, were realized when the random initial placement was used. Absolute reductions in the number of page faults were again comparable, with the depth-first reductions slightly smaller than those for the random configuration. Modifying the buffer purging policy had only slight effects on the page utilization and cleanliness distributions.

As in the simulations where the purging algorithm for object space was modified, the random initial placement was less sensitive to this policy change than was the depth-first arrangement. These results provide further evidence that as the grouping scheme becomes "better," the relative beneficial effects of changes that involve grouping become larger.

### *8.3.7 In-Core Residence Times*

Recall the clock associated with a virtual memory that was introduced during the presentation of the reference trace compression algorithm in section 6.1.4. Time advances one tick for each field in every object created dynamically or swapped into core. Define the *in-core residence time* (*in-core lifetime*) of an object to be the number of ticks that occur from the time an object first appears in core until it is purged. An object that is repeatedly faulted upon and subsequently flushed during a computation has a number of in-core residence times. Distributions of in-core lifetimes were graphed (Figure 8-6) in order to evaluate the validity of the purging assumptions made during the design of the compression algorithm. Section 9.1 argues that such assumptions were justified. Scatter diagrams plotting core entry and exit times for objects were created (Figure 8-7) in order to determine the correlation between faulting and purging tendencies. These plots indicate the presence of sets of objects that are simultaneously faulted upon. While many of these sets are

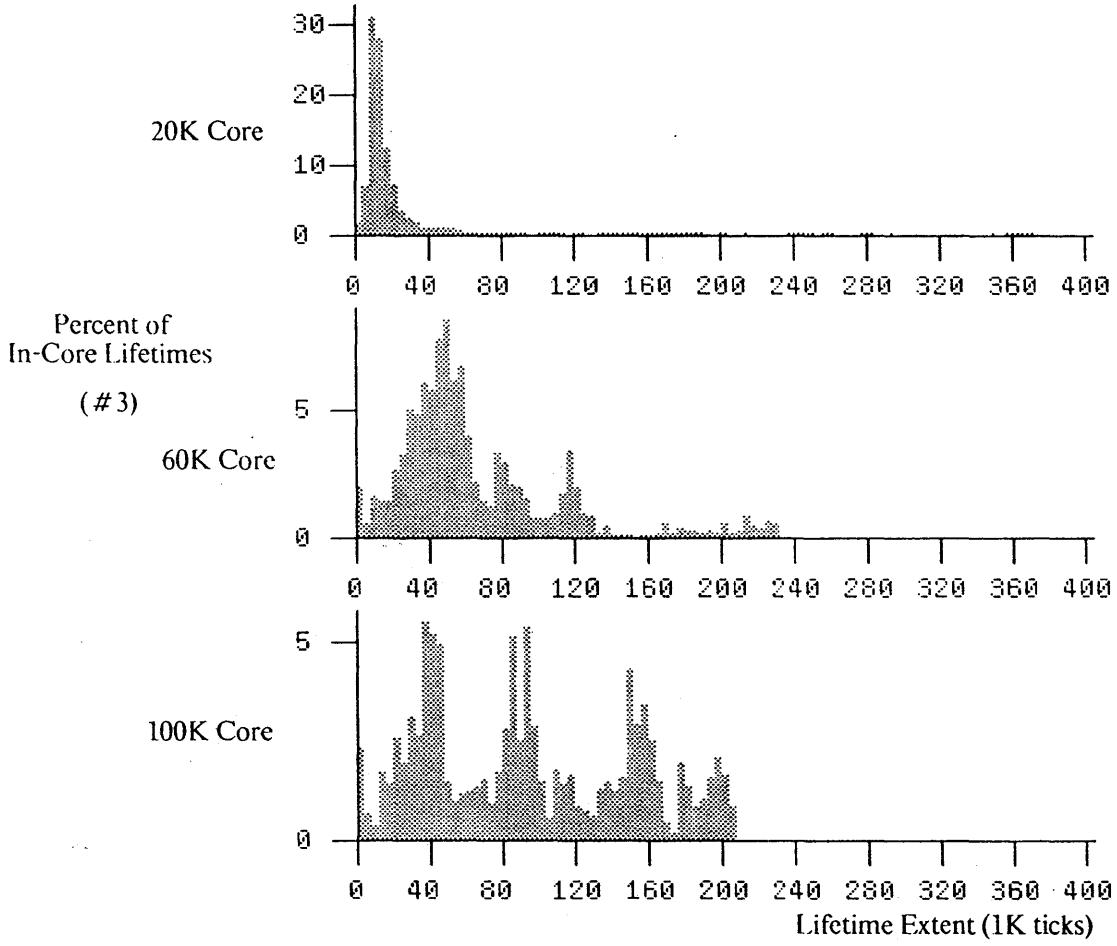


Figure 8-6 In-Core Lifetimes

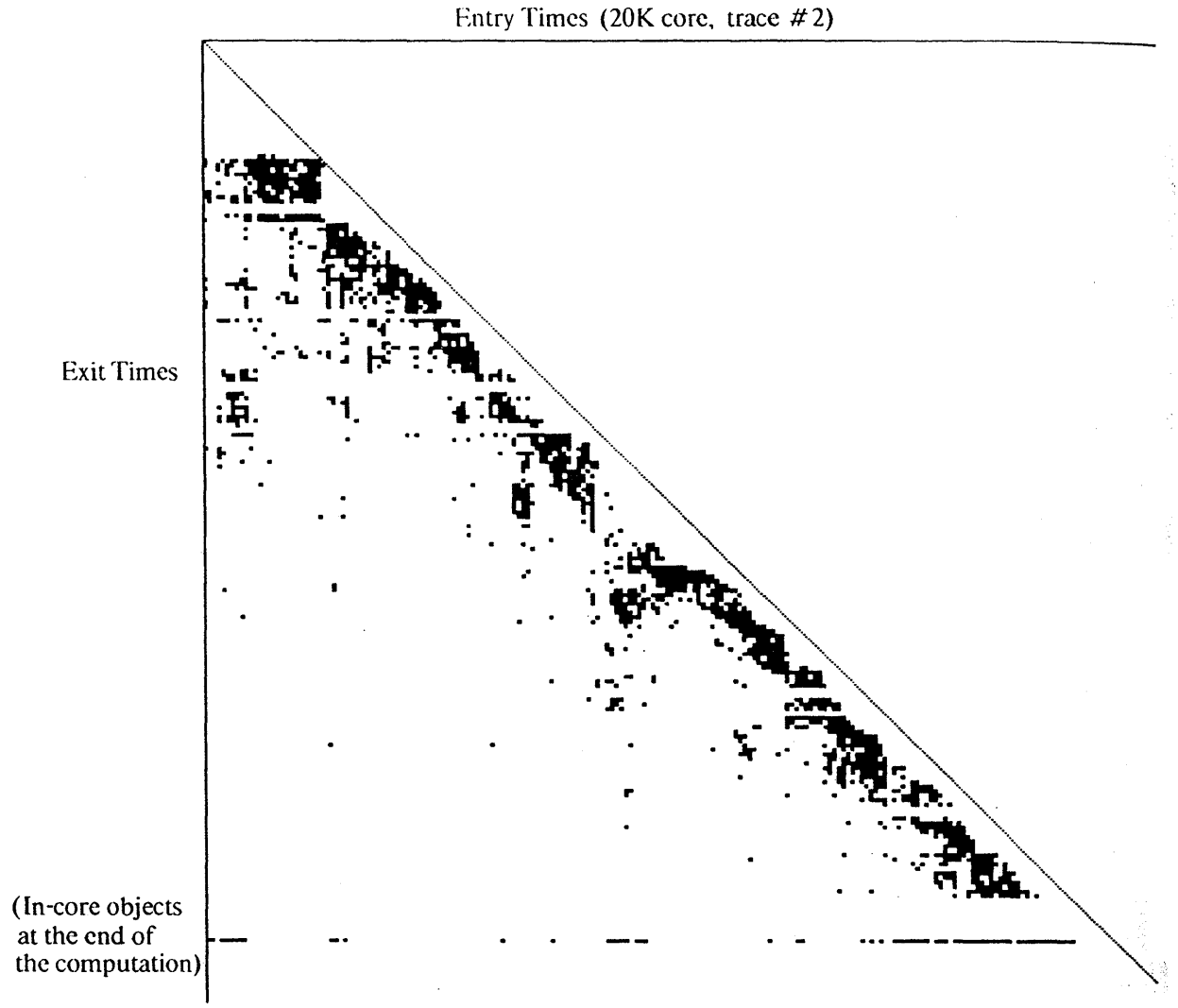


Figure 8-7 Object Entry and Exit Times

purged at roughly the same time, there are sets whose members are purged at different times. For these sets, static schemes cannot effectively group objects to accommodate both faulting and purging tendencies.

The means, medians, and quartiles of the residence distributions generally increased as the size of core grew larger. Residence data confirms the assumption that average objects remain in core for longer periods of time in larger memories. In almost all cases, the larger standard deviation value for the residence distribution increased with core size. For small memories in which thrashing was a problem, most residence times were similar because objects did not have a chance to distinguish themselves. Larger core sizes provided enough time for frequently used objects to establish themselves as such, which resulted in a spreading of the in-core residence time distribution. Less purging and faulting occurred in larger memories, which in turn decreased the total number of ticks that occurred during a given computation. With a smaller possible maximum, the observed maximum of the residence distribution typically declined for larger core sizes.

Because the set of residence distributions was not weighted with respect to the core size of objects, it was possible for the average residence time to exceed the size of primary memory. Such anomalies were due to the fact that some small objects had large lifetimes and some large objects had small lifetimes.

Scatter diagrams of object entry and exit times indicate that the faulting and purging tendencies were not completely correlated. Part of this exit time differential may be attributed to the particular purging scheme used. A substantial portion of the discrepancy is due to an inherent asymmetry between faulting and purging (i.e., between the first-use and last-use of objects). These results indicate that objects may be statically grouped for either faulting or purging but not always for both. Some *swapping sets* undoubtedly exist, in that many collections of objects are characterized by similar times for first-use and last-use in most computations. On the other hand, there are sets of objects that are simultaneously faulted upon yet have very different in-core residence time profiles. Static grouping cannot achieve satisfactory paging performance for both faulting and purging for these kinds of object collections. At most, one transfer direction may be effectively accommodated in both object-swapping and page-swapping virtual memories. If these first-use and last-use relationships are not time invariant, then static groupings may not be able to efficiently accommodate either direction of data transfer. One clear advantage LOOM has over a paged virtual memory for these collections is that the static placement constraints are not necessarily the dynamic constraints. Core utilization may be retained at an arbitrarily high level in LOOM. In a paged virtual memory without copying, there are no dynamic choices to be made short of deciding which full page to discard during a page fault. Another LOOM advantage is the delayed binding of dynamically created objects to disk homes. If most of the last-use discrepancy is due to the first time new objects are purged, then LOOM has solved the problem by consecutively packing together on the disk newly matured objects. Problems still remain if this packing is not suited for future purging and/or faulting tendencies. While not proving the nonexistence of grouping techniques

that efficiently handle the bi-directional transfer of objects, the scatter diagrams of core entry and exit times indicate that the search for such algorithms is likely to be difficult.

### 8.4 Analysis of Predictions

The paging performance predictions based on static analysis of the initial placements may now be compared with the results for LOOM. Since the LOOM results were similar to the paged virtual memory data, the same predictions were upheld. Similar initial placements had similar paging performance, while the random initial placement consistently underperformed all other static arrangements. Retained fraction values for the optimal groupings predicted either the OOZE or the depth-first initial placements to be the best realizable grouping strategy, depending on the reference trace. Breadth-first groupings were granted third place followed by the random initial placement. These predictions were upheld in their entirety, except for the reversal of the first two categories of grouping strategies in the third trace. The breadth-first groupings again improved their performance *vis-a-vis* the on-page pointer ratio predictions, but their upward performance climb was strongly curtailed. Unlike the results for the paged virtual memory, there were no reversals in the LOOM simulations. In a given trace, the distance between any two of the three band categories was essentially independent of the core size.

The location of the realizable groupings in the performance range defined by the optimal and random initial placements indicate some success with static grouping. However, there is still room for improvement. Unlike the paged virtual memory, in which the relative difference between band groupings was dwarfed by the improvement over the random initial placement, the LOOM simulation was much more sensitive to the particular grouping. Better groupings will have a much larger performance impact on LOOM than on a paged virtual memory.

Since there was no body of empirical data for LOOM-like systems, a number of parameters and policies were modified in order to determine the relative importance of grouping as well as its effects on these changes. Page fault reduction ratios indicate that grouping reduced the number of excess page faults by 18.1% to 63.6%, depending on the initial placement and the trace. Further absolute reductions of up to 30% could be achieved by expanding the disk buffer at the expense of having the buffer handle more page references. Changing the buffer purging policy netted about a 5% reduction in the number of page faults, while a modification of the core purging policy produced mixed results.

Data from these simulations may be summarized by a few key points. First, a good grouping algorithm can substantially affect the performance of any LOOM-like system. Other virtual memory parameters and policies are of secondary importance. Secondly, modifications to these parameters and policies will have larger relative improvements for better grouping strategies. Third, these simulations have identified lambda resolution as a key problem deserving study. In order to eliminate many page references and page faults, a good predictive algorithm needs to be found for determining the set of pointers to resolve immediately when an object is faulted upon. Finally, these results have underscored the importance of endeavoring to establish a synergistic relationship

between static grouping algorithms and the dynamic purging routine. If no accommodating static grouping can be found, then the purge algorithm ought to have the *option* of ignoring the initial placement as it performs its task. Related to this interaction between purging and grouping is the dynamic allocation and deallocation of new objects as well as their purging and placement. A brief introduction to such topics in the context of initial placement stability is presented in Chapter 10.

## 9. LOOM Versus a Paged Virtual Memory

In addition to measuring the effect of static grouping in LOOM and a paged virtual memory, the simulations and compression algorithm were designed to support a direct empirical comparison between their paging performances. Appendix C contains the relevant numerical details supporting the following analysis.

Adjustments to the core size of the LOOM simulation were made in order to account for the 2K disk buffer and the object table (OT), which contains the starting addresses and other information of all in-core objects. The appropriate OT size was derived from the maximum number of in-core objects when purging began in the LOOM simulations. A safety margin of 10% was added to the table length, which ranged from 2.5K words (20K core) to 15.5K words (100K core). No adjustments were made for the paged virtual memory, since the memory requirements for page tables and other associated information were small enough to be neglected.

A number of simplifications to the simulations avoided areas beyond the scope of this thesis and in doing so may have given either type of virtual memory a slight advantage. For example, all pointers in the paged virtual memory simulation were assumed to be direct, so that no indirection table was required. However, the problem of handling the *'become:'* operator was not solved. No real management of secondary memory was done in either simulation. One disadvantage for the paged virtual memory was the unavailability of dynamic copying schemes to compact working sets. A fixed static allocation of new objects and the assumption of UID reuse may have scattered the working set over many pages. Leaf references, on the other hand, provided the LOOM simulation with enough information to effectively manage primary memory. Since these right references were at times neglected in the paged memory simulation, it gained a nontrivial advantage over LOOM. Another simplification was the omission of the run-time stack. Because of pointer compression, the LOOM simulation would have derived a small advantage from the inclusion of the stack. This benefit would not have been significant, since it would have been proportional only to stack depth.

A direct comparison between the two types of virtual memories indicates that LOOM outperforms a paged virtual memory for a range of small memory sizes. Such results were invariably obtained for warm starts, cold starts, simulations that did reference counting, and those that did not. In addition to the particular computation, the initial placement played an important role in determining the memory size interval for which LOOM outperformed the paged virtual memory. Since the length of this interval varied inversely with the quality of the initial placement, the desirability of a LOOM-like virtual memory will depend on the existence and stability of quality initial placements.

The relative performance of LOOM is explained by examining its costs and benefits. LOOM increases the apparent size of main memory by compressing pointers and swapping objects. Pointer compression reduced memory requirements by one-third, while object swapping guaranteed a complete utilization of primary memory. Since page fault curves typically contain a "knee," the importance of this increase in the apparent size of memory decreases for core sizes large relative to the computation.

LOOM's method of extending the apparent memory size incurs costs. Resolution of a lambda requires an access to the disk representation of the object containing the lambda. Although lambda resolution becomes less costly for larger core sizes (see section 9.3), it causes a small but not insignificant fraction of the page faults. The other cost arises from the policy of swapping objects instead of pages. When an object is faulted upon, the page containing the object is swapped into the disk buffer if it is not already there. Only the required object is copied from the disk buffer into core. References to objects on one page can therefore cause more than one LOOM page fault. Purging objects on the same page can also cause more than one page fault. Page-swapping virtual memories, on the other hand, prefetch information by bringing all objects on a page into core whenever any object on that page is accessed. All objects on a single page are also purged simultaneously. This advantage of page swapping is synergistic with good groupings of objects on pages. Because the performance of LOOM is not as strongly dependent on the quality of the grouping scheme, grouping schemes have a limited effect on the number of page faults in LOOM. Although pages are physically swapped in both types of virtual memory, these two swapping schemes are inherently different. LOOM's finite disk buffer size causes some page faults that are not encountered by the paged virtual memory. If primary memory is large enough to accommodate a computation without purging information, the paged virtual memory is optimal in that it causes the smallest number of page faults. LOOM too can achieve this performance level for all disk buffers larger than a certain size. Without a large enough buffer, however, the performance of LOOM can be substantially worse, since references to objects on a single page can cause multiple page faults.

In short, LOOM's benefits are important for primary memory sizes that are small relative to the size of the computation. The overhead caused by a finite buffer becomes critical when the buffer is small in absolute terms or in terms of the size of main memory. Since a small, fixed-size buffer was employed in these experiments, the overhead substantially degraded LOOM's performance for large primary memory sizes.

### 9.1 Equivalence and the Compression Algorithm

In order for the compression scheme to preserve weak equivalence between the full and compressed reference traces, the virtual memory simulations had to guarantee that the minimum residence time was 10K ticks. In the LOOM simulations for a 20K core, the first quartile for the in-core residence time distributions varied between 10K and 12K ticks. For at least three-fourths of the times at which an object appeared in core, this equivalence preservation criterion was met. In LOOM

simulations involving 40K core memories, the first quartile varied between 24K and 30K ticks, which indicates that the number of violations was not significant for any memory size above 20K words.

One insight that eliminates most of these possible violations is the fact that clocks for larger core sizes run slower than clocks corresponding to smaller core sizes. As long as no faulting anomalies occur, larger core sizes purge and fault less often and thus their associated clocks tick slower. The minimum requirement of 10K ticks referred to a clock associated with a 10K main memory. However, the in-core residence times in question were obtained for a 20K core. For very small core sizes in which a substantial amount of purging occurs, changing the core size has a nontrivial impact on the virtual memory clock. Table 9.1 estimates the ratio of clock speeds by comparing maximum in-core residence times. The top portion presents clock speed ratios for differences in core sizes of 20K, while the bottom half of the table presents ratios when the core size is halved. Conservative extrapolations for the 20K to 10K clock rate ratio fall into the 1.3 to 1.5 range. Since the in-core residence time distribution falls off rapidly as the abscissa approaches zero, even small clock rate ratios will substantially reduce the number of possible violations. Most of the few remaining small in-core residence times occurred during the first purge cycle.

| Core sizes | | | Trace #1 | Trace #2 | Trace #3 |
|---|---|---|---|---|---|
| 120K | v. | 100K | * | 1.00 | 1.01 |
| 100K | v. | 80K | * | 1.05 | 1.03 |
| 80K | v. | 60K | 1.01 | 1.06 | 1.08 |
| 60K | v. | 40K | 1.04 | 1.17 | 1.15 |
| 40K | v. | 20K | 1.30 | 1.36 | 1.39 |
| 120K | v. | 60K | * | 1.11 | 1.13 |
| 80K | v. | 40K | 1.05 | 1.16 | 1.25 |
| 40K | v. | 20K | 1.30 | 1.36 | 1.39 |

Table 9.1 Estimated Clock Speed Ratios

The paged virtual memory, which used an LRU purging scheme, guaranteed a minimum in-core residence time equal to the size of core. However, the clock associated with a paged virtual memory ran faster than a clock in an object-oriented virtual memory for two reasons. Exclusively using 32-bit pointers caused a 50% expansion in the size of the computation. Secondly, every page swapped into core contributed a number of ticks equal to the page size (256 words), regardless of the size of the object that caused the page fault. Since core utilization ranged from 2/5 to 3/5, there was another clock speed increase of 67% to 150%. In the worst case, the paged virtual memory for a 10K core size ran nearly 4 times as fast as the corresponding clock in the compression algorithm.

Page fault reductions for the 10K to 20K transitions for the paged virtual memory ranged from 9% to 12% for the random initial placement and from 19% to 23% for the band arrangements. Optimal groupings had changes in the interval from 37% to 45%. A clock slowdown factor of nearly two occurred for the optimal arrangement in the 10K to 20K transition. Also offsetting the clock rate increase was the fact that any reference to a page moved the page (and therefore all objects on that

page) to the top of the LRU ordering. These actions, which would increase the duration of the in-core lifetime for all objects on the page, made the core size a strict lower bound on all in-core lifetimes.

Although equivalence *may* have been lost for simulations of very small core sizes, comparisons between all LOOM simulations and all paged virtual memory simulations are still valid. Each computation was represented by a fixed, compressed reference trace. The only warning that must be issued is that very small core sizes in both simulations may not accurately reflect the true behavior of the Smalltalk-80 programming environment. However, since the quantity of possible violations was low, the actual behavior was closely approximated by the simulations. Two independent factors confirm this assumption. First, the paging behavior for small core sizes fits naturally with the faulting rates for large core sizes, which unquestionably represent the actual behavior of Smalltalk-80. Secondly, most of the serious violations for LOOM occurred during the first purge cycle, before an equilibrium between marked and unmarked objects had been achieved. Warm-start simulations are described in section 9.4 and show that cold starts accurately described paging behavior. Any violations that did occur had only a negligible impact on the overall level of performance.

## 9.2 A Naive Comparison

The same partitioning of grouping strategies into distinct categories was seen in the paging performance of both types of virtual memory. Except for the lack of reversals in LOOM and an occasional swap of two adjacent categories, the performance rankings of static groupings were independent of the virtual memory type. As was previously mentioned, the paged virtual memory band was narrow, while the LOOM band was wider and closer to the performance of the random initial placement.

The virtual memory type with the better paging performance depended upon the reference trace, the core size, and the specific grouping scheme (Figure 9-1). For all realizable grouping strategies, LOOM dominated for some range of small core sizes. Pointer compression and a high utilization of core allowed LOOM to perform rather well, while the low utilization of core and uncompressed pointers caused the paging scheme to thrash. At a certain core size, depending on the reference trace and the initial placement, the paging performances became similar. Let the *crossover point* be the core size at which the function representing the faulting rate for LOOM intersects from below the corresponding function for the paged virtual memory. At all points less than the crossover, the LOOM simulation had a better paging performance than the paged virtual memory. For all larger core sizes, the paged virtual memory dominated as its faulting rate declined further. LOOM's faulting rate decreased slowly and became constant at the relatively small core size for which no purging occurred.

One cause of LOOM's poor performance for moderate and large core sizes was the fixed size of the buffer. Many pages were faulted upon more than once in LOOM in order to satisfy faults on different objects. In the paged virtual memory, when a page was brought into core for one object,

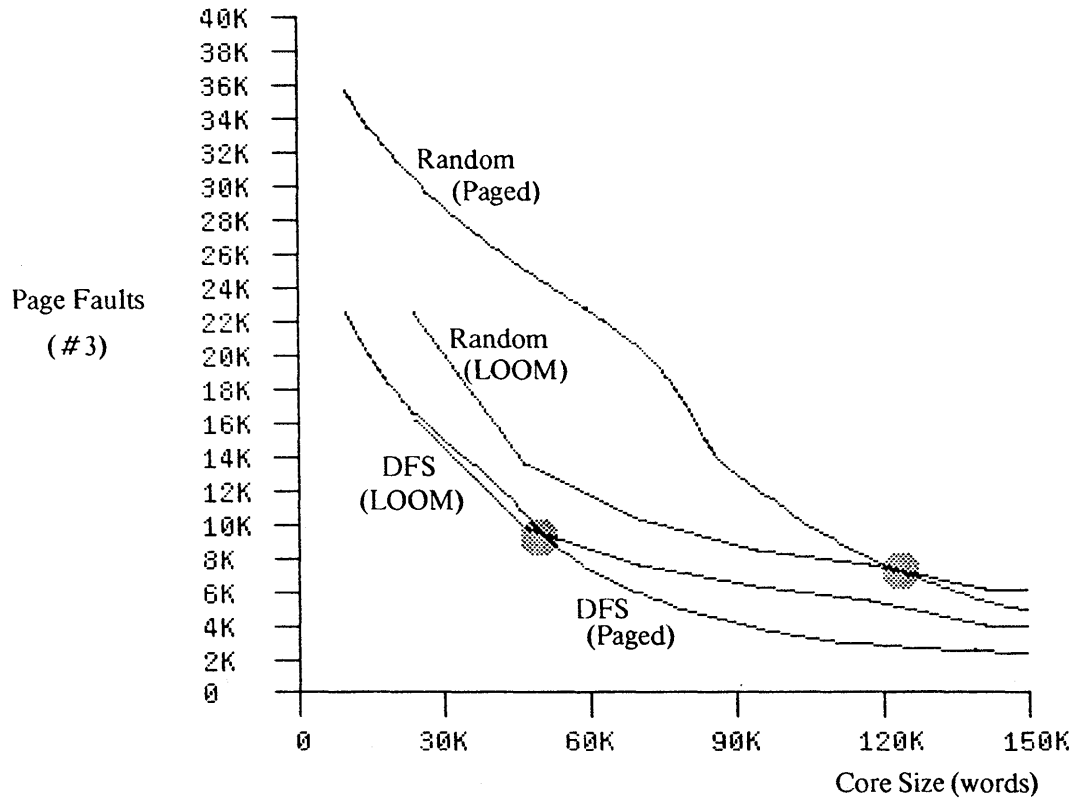


Figure 9-1 A Naive Comparison

all other objects on the page were automatically transferred into core. This set of objects remained in core until the page was purged. LOOM also had to contend with the resolution of lambda pointers, which caused additional references to the same page. The paged virtual memory naturally outperformed LOOM when little or no purging occurred. Swapping a large working set into core was a simple task for the paged virtual memory, since it was only a sequence of page transfers. In dealing with object faults and lambda faults, however, the LOOM buffer became a bottleneck that severely limited the throughput of needed objects and data.

Because the optimal initial placements caused little purging for even the smallest core sizes in the paged virtual memory simulations, the LOOM simulation with the optimal grouping was outperformed for all memory sizes. No crossover point existed. On the other hand, the LOOM simulation with the random initial placement performed well for a wide spectrum of primary memory sizes. The extremely low utilization of core by the paged virtual memory caused thrashing for a large range of core sizes.

These uninterpreted faulting rates are potentially misleading for a number of reasons. First, the simulations represent a cold start. Core memory is initially empty and is filled as the computation proceeds. In the Smalltalk programming environment, there is never a cold start. The image of core is loaded from a disk file (by swapping pages!) when the user session begins. When the user leaves the Smalltalk environment, the core image is saved by swapping pages to the disk. All intervening user actions are supported by a warm system that is normally full of objects. Warm-start simulations were performed in order to determine the effect of cold and warm starts in both types of virtual memories. The analysis of these runs is presented in section 9.4.

Another note of caution in interpreting these paging results concerns storage management. Object references in the compressed reference traces, which were inherently related to the computation, were independent of the particular memory management scheme used. Leaf references, on the other hand, represented the requirements of a reference counting policy used by any type of memory. This factoring of requirements into computational needs and memory management needs allowed comparisons to be made on simulations that included or neglected the leaf references. The LOOM simulation was originally at a disadvantage, because it was required to process the leaf references in order to perform the computation. Except for violations of the assumption concerning clean leaves, storage management would have caused no additional page faults. By ignoring the leaf references, the paged virtual memory simulation effectively did no memory management. Additional simulations, in which the paged virtual memory treated leaf references as object references *or* the LOOM simulation neglected leaves and leaf references, were performed. Section 9.3 contains the analysis of these runs.

Comparisons between LOOM and a paged virtual memory indicate that any type of grouping scheme realized tremendous performance benefits for the latter. These gains were largely due to the increase in core utilization. LOOM, which selectively allows objects into core, can perform reasonably well in the face of a poor initial placement. If the quality of an initial placement is

time-variant and extremely volatile, then LOOM is successful. On the other hand, if initial placements tend to behave consistently for all kinds of computations and do not decay after long periods of use and modification, then a definitive evaluation between the two types of virtual memories requires a more extensive analysis and also depends on the existence or nonexistence of better grouping schemes. Let the *stability* of an initial placement be the rate at which it deteriorates and becomes obsolete. The relative success of a naive, paged virtual memory then depends on the existence and stability of quality initial placements.

### 9.3 Leaf/No-Leaf Comparisons

In order to determine the effect of leaf references on the two types of virtual memory, simulations for two different initial placements, a random and a depth-first, were repeated for all three reference traces. Call the two original simulations the *initial* simulations. The original LOOM simulation will also be referred to as the *leaf* simulation, while the original paged virtual memory simulation will at times be called the *no-leaf* simulation. For these extra runs, the *no-leaf* simulation for LOOM neglected all leaf references, except for object contraction created no leaves, and never resolved any lambdas. All pointers were assumed to be short and object faults were handled normally. On the other hand, for these repeated runs, the *leaf* simulation for the paged virtual memory treated leaf references as object references. This interpretation of the reference trace corresponds to a reference counting scheme in which the reference count is kept with the object.

These additional sets of comparisons substantially improved the performance of LOOM when compared to the paged virtual memory. By equalizing the memory management duties, a more just picture emerges. The main memory size intervals for which LOOM dominates were extended. LOOM's relative improvements over the paged virtual memory for small core sizes also increased.

Requiring the paged virtual memory to treat leaf references as object references consistently led to an increase in the number of page faults. For small core sizes (30K), the increases fell into the 22% to 35% range. These increases peaked for moderate memory sizes and substantially declined for larger core sizes. Maxima were attained for core sizes at the "knee" of the page fault function for the no-leaf paged virtual memory. Larger core sizes handled the leaf references with only a small performance degradation. For very large core sizes in which purging did not occur, leaf references caused less than a 5% increase in the number of page faults. Most of the leaf references did not cause additional page faults, because the objects to which they referred were already in core or would soon be swapped into core.

Paging rates for the no-leaf LOOM simulation represent a lower bound on any lambda resolution scheme that does not prefetch objects because this "policy" was better than the optimal lambda resolution scheme. Page fault reductions of 20% to 34% and 15% to 32% occurred for the depth-first and random initial placements, respectively. Relative decreases in the number of page faults were again larger for the depth-first initial placement, while absolute reductions were slightly larger for the random initial placement.

Both relative and absolute page fault reductions typically decreased for larger core sizes. The apparent size of core was larger because no leaves were created for leaf references. Because the slope of the page fault function is not constant but decreasing, this apparent change in memory size was more important for smaller core sizes in which thrashing occurred. Slightly larger memory sizes caused substantially fewer page faults.

The second link between core size and performance improvement arose from the hit rate of page references caused by lambda resolution. Call these values the *lambda hit rates*. As an approximation to this value, consider the excess number of hits and misses generated by the leaf simulation of LOOM as compared to the no-leaf simulation of LOOM. Table 9.2 presents the hit rate of these additional page references.

| Core Size | 20K | 40K | 60K | 80K | 100K | 120K |
|---|---|---|---|---|---|---|
| DFSID #1 | 64.8% | 64.9% | 70.4% | 78.0% | * | * |
| DFSID #2 | 67.5% | 67.4% | 72.6% | 71.4% | 70.5% | 76.4% |
| DFSID #3 | 67.8% | 70.5% | 71.0% | 72.1% | 72.3% | 77.7% |
| Hash #1 | 57.6% | 58.6% | 62.5% | 72.6% | * | * |
| Hash #2 | 60.9% | 61.1% | 65.6% | 63.7% | 62.1% | 68.2% |
| Hash #3 | 57.3% | 61.5% | 59.1% | 62.7% | 62.3% | 66.7% |

Table 9.2 Lambda Hit Rates

Between half and three-fourths of all references caused by lambda resolution were satisfied by the disk buffer and did not cause page faults. This lambda hit rate was much higher than the hit rate for the no-leaf simulation or the composite hit rate for the initial simulation. Depth-first lambda hit rates were larger than those for the random initial placement. A higher utilization of the disk buffer, longer residence times for disk pages, and a larger on-page pointer ratio accounted for these differences. Larger core sizes, which caused less purging and faulting, meant that fewer page references were generated. This reduction caused pages to remain in the disk buffer longer and increased the lambda hit rate. Hence the page fault reductions in the no-leaf simulations were generally smaller for larger core sizes.

Neglecting leaf references had a small effect on both page utilization and page cleanliness in the LOOM simulations. Because no lambdas were resolved, a higher fraction of page references were due to the contraction of dirty objects. Page utilization declined in the no-leaf simulation for two reasons. First, the page utilization was smaller for purging than for faulting, and the lack of lambda resolutions increased the relative amount of purging versus faulting. Secondly, some resolved lambdas in the initial simulation referenced an object on the same page. If this page were missing from the buffer, then resolving the lambda would have referenced two objects on the same page. In the no-leaf simulation, however, only the referenced object, and not the object containing the lambda pointer, would be accessed. Since for even the depth-first groupings the on-page pointer ratio was exceedingly low, this effect was negligible.

Comparisons between the two leaf simulations or the two no-leaf simulations, as opposed to the two initial simulations, improved the performance of LOOM with respect to the paged virtual memory

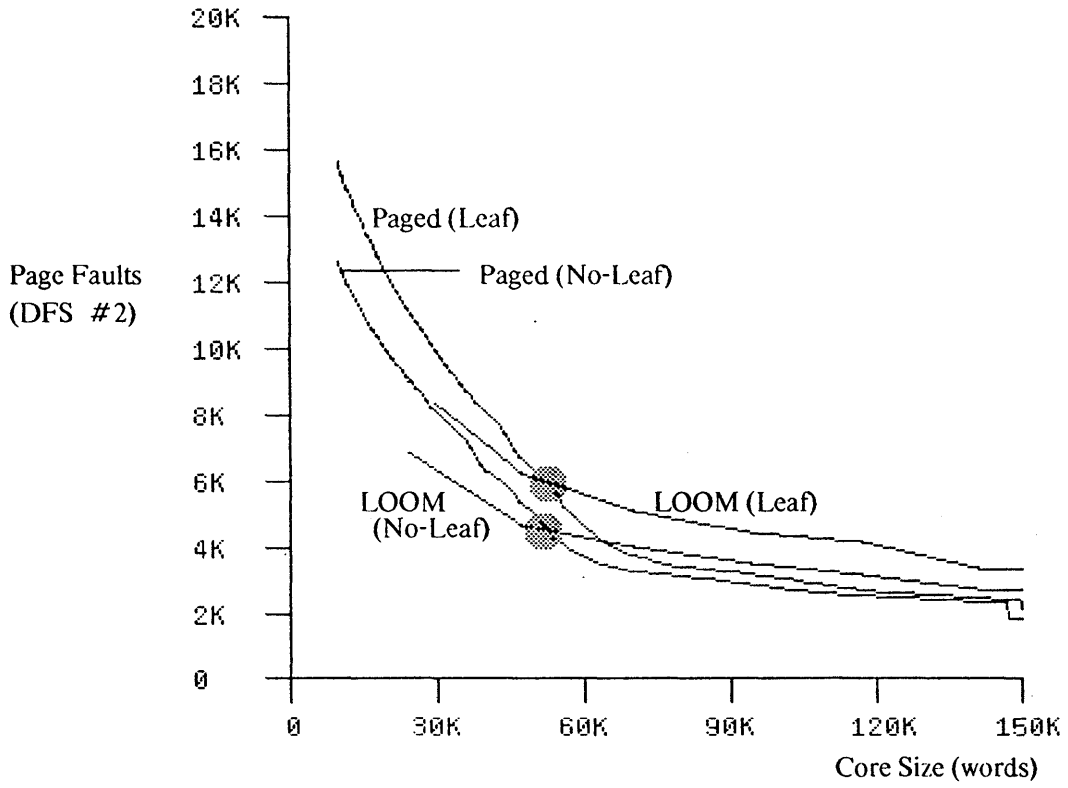


Figure 9-2 A Leaf/No-Leaf Comparison

(Figure 9-2). In general, the leaf and no-leaf crossover points were closer to each other than either was to the original crossover point. These results indicate that the original comparison severely penalized LOOM by not requiring the paged virtual memory to reclaim free storage. Substantial shifts in the crossover point occurred when the reference counting policy was uniformly included or neglected in the simulations.

## 9.4 Warm-Start Comparisons

There are three extremes to consider in comparing virtual memory performance: an empty memory, a memory full of irrelevant objects, and a memory full of useful objects. Except for purging, a memory full of irrelevant objects is similar to an empty one. Cold starts provide information on the first scenario, while warm starts with very large core sizes yield data for the third. Warm starts with tiny core sizes represent a mixture of both useful and useless objects.

Warm-start simulations were done for the random configuration and one depth-first initial placement for all three reference traces. Memory was filled by running a cold-start simulation on the last half the reference trace. All event counts were set to zero and the virtual memory simulators then processed the entire reference trace in the normal direction.

Because our definition of page faults only counts the number of pages swapped from disk to core and not vice versa, the warm starts for the paged virtual memory always outperformed the corresponding cold starts. For extremely small core sizes, however, the LOOM warm starts caused more page faults than the corresponding cold starts. Nevertheless, warm starts for LOOM outperformed cold starts for moderate and large sizes of primary memory. Warm starts did not shift the crossover point consistently in either direction.

All warm-start simulations of the paged virtual memory outperformed the corresponding cold-start. The definition of a page fault for the paged virtual memory guaranteed that the warm start would be no worse. Whenever a page reference was not satisfied by the set of in-core pages, the simulator would discard the least recently used page and fetch the required page. There was exactly one page fault, regardless of whether the discarded page was clean, dirty, or unused. The set of in-core pages remaining from the warm start could therefore be viewed as empty or valid, if a page needed to be discarded or a reference needed to be satisfied, respectively. Typical page fault reduction ranges were 3.4% to 6.9% (30K), 11.8% to 41.0% (120K), and 30.0% to 70.2% (240K). Relative performance improvements generally increased with core size for the obvious reasons. While the relative performance improvements were typically larger for the depth-first initial placement, the absolute improvements in paging performance were larger for the random grouping. In fact, for very large core sizes, the warm-start random configuration outperformed the warm-start depth-first arrangement. This anomaly was due to the much larger set of pages of the random initial placement that was accessed during the warming phase. Many untouched objects located on these pages were later used in the simulation of the full reference trace.

In the warm-start LOOM simulations, the change in paging performance depended on core size and the particular reference trace. Consider the first and third compressed reference traces. For small core sizes, warm starts caused more page faults than cold starts. This performance degradation, which was less than 6%, was due to additional purging. The contraction of dirty objects that were swapped into primary memory during the warming phase caused additional page references. In all three reference traces, the relative paging performance improved as the size of core was increased, because a larger fraction of the in-core objects was not purged. For a core size of 60K, the reduction in page faults was 4.1% to 15.9%. If the core size were large enough to prevent purging, then the interval of performance improvements jumped to 48.9% to 84.9%. Unlike the paged virtual memory simulation, no warm start for the random initial placement ever outperformed the corresponding depth-first warm start. Except for the disk buffer, the initial contents of primary memory were independent of the particular grouping involved. Neither of the two groupings dominated in the relative performance improvement category, although the random initial placement for the most part had the larger absolute reduction in the number of page faults. The important factor in deciding the level of performance improvement was core size, which determined the amount of useful data already in hand.

In three of the six comparisons between warm and cold crossover points, the warm start was preferred by LOOM. Two of the remaining comparisons yielded close crossover points and indicated that a cold start bettered the relative performance of LOOM. The warm-start simulation of the second reference trace involving the depth-first initial placement had no crossover point. This small data set indicates that the preferential start for LOOM is not known a priori. Instead, the particular reference trace, range of core sizes, and initial contents of the two core memories determine whether LOOM benefits more from a cold or warm start.

Warm-start faulting rates for the LOOM simulation declined so rapidly that a second crossover point occurred in the graphs for the second and third reference traces for the depth-first initial placement (Figure 9-3). Another such point nearly occurred for the first reference trace. These crossover points are the duals of the points that correspond to small core sizes, because at these points the page fault function for LOOM intersects the other curve from above. There is a range of smaller core sizes for which the paged virtual memory outperformed LOOM and a range of larger core sizes for which the opposite occurred. LOOM was able to outperform the paged virtual memory because of its roughly 100% utilization of core. No unneeded objects were ever swapped into core by LOOM. Unless a perfect static grouping can be found that is not time-variant, the paged virtual memory will always swap unnecessary objects into core. Let u be the apparent usage of core by the paged virtual memory, where *usage* is defined to be the ratio of the total size of all referenced, in-core objects to the size of primary memory. Then the actual utilization of core (as measured by short pointers) will be u/c, where c is the average compression ratio of disk size to core size. Therefore, for any grouping strategy, the utilization of core by LOOM will always be greater than the corresponding utilization by the paged virtual memory.

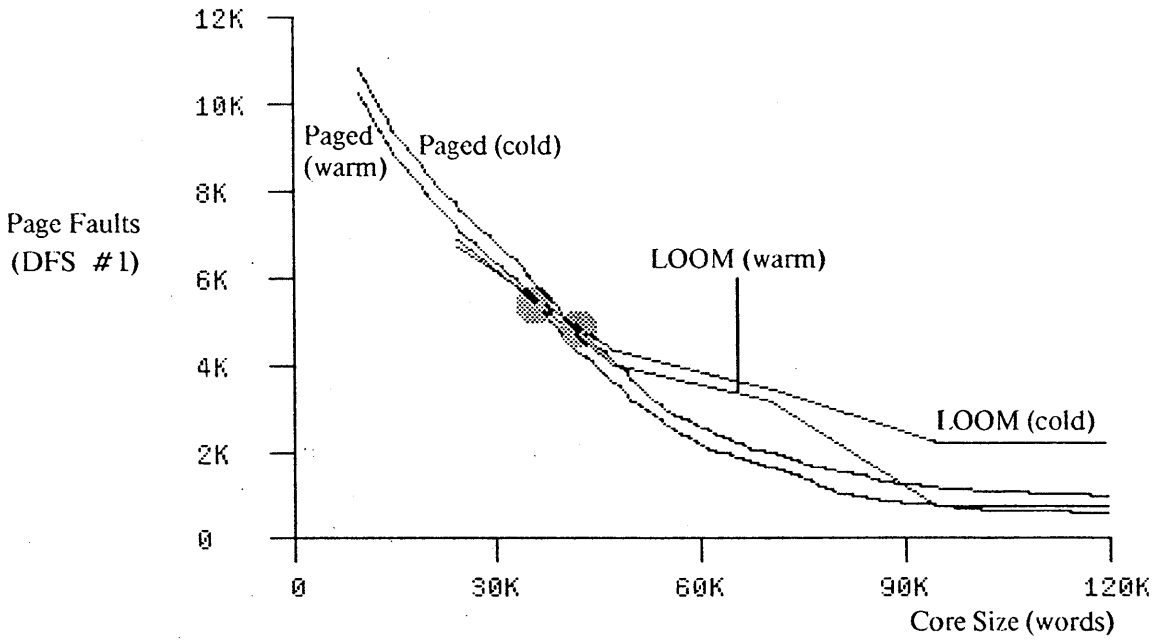


Figure 9-3 A Warm/Cold Comparison

Between the two crossover points was a wide range of core sizes for which the benefits of a warm start did not allow LOOM to overtake the performance of the paged virtual memory. However, LOOM was processing leaf references while the paged virtual memory was not.

Warm starts had opposite effects on the average page utilization and cleanliness values. Partially full memories meant that either fewer objects were faulted on, more objects were purged, or both. These tendencies increased the relative importance of purging over faulting and accounted for the increase in dirty pages as well as the general decrease in disk page utilization. Page utilization increases were seen, however, in 4 of the 6 simulations in which no purging occurred.

### 9.5 A Note Concerning Page Faults

Although a consistent definition of a page fault applied to both types of virtual memory simulations, the LOOM scheme encountered an inherent penalty. A page fault was defined as the physical swapping of a disk page from disk to core. Regardless of whether the displaced page was clean, dirty, or empty, only one fault occurred. Therefore, every reference not satisfied by the contents of core in the paged virtual memory caused exactly one page fault. In LOOM, on the other hand, an object fault may have caused more than one page fault. If core were full, then purging had to occur. Dirty object contractions caused page faults if the generated page references were not satisfied by the disk buffer. After enough purging had been accomplished, if the page containing the object were not in the disk buffer, then another page fault occurred.

### 9.6 Extending These Results

One shortcoming of the analysis between the two types of virtual memory is the restriction on computation length and the size of the set of accessed objects enforced by a virtual machine emulator that executed bytecodes 1000 times as slow as the actual Smalltalk-80 interpreter. The important question is whether these results will scale to lengthy computations involving more objects in larger systems.

Each crossover point may be associated with a measure relating the size of the computation to the available amount of core memory. For a given computation c, let M be the smallest size of core memory for which a LOOM system would never purge an object. Let m be the crossover point. Define the *crossover size ratio* to be M/m. Tables 9.3 and 9.4 present the crossover size ratios for selected simulations involving a depth-first grouping, assuming a linear interpolation between empirically measured page fault rates for the LOOM virtual memory.

These ratios highlight the importance of uniformly processing or neglecting leaf references. For example, when LOOM used leaf references but the paged virtual memory neglected them, the well-defined crossover ratios varied from 2.3 to 5.3. One such ratio did not exist. When the leaf references were treated uniformly by both types of simulations, the crossover size ratios were all defined and fell into the 1.6 to 2.9 range.

| Trace Number | LOOM Leaf | LOOM Warm Start | Paged Virtual Memory Leaf | Paged Virtual Memory Warm Start | Initial Placement | Crossover Size Ratio |
|---|---|---|---|---|---|---|
| #1 | Yes | No | Yes | No | DFSID | 1.59 |
| #2 | Yes | No | Yes | No | DFSID | 2.71 |
| #3 | Yes | No | Yes | No | DFSID | 2.11 |
| #1 | No | No | No | No | DFSID | 1.64 |
| #2 | No | No | No | No | DFSID | 2.71 |
| #3 | No | No | No | No | DFSID | 1.99 |
| #1 | No | Yes | No | Yes | DFSID | 1.74 |
| #2 | No | Yes | No | Yes | DFSID | 2.90 |
| #3 | No | Yes | No | Yes | DFSID | 2.16 |

Table 9.3 Leaf/No-Leaf Comparisons

| Trace Number | LOOM Leaf | LOOM Warm Start | Paged Virtual Memory Leaf | Paged Virtual Memory Warm Start | Initial Placement | Crossover Size Ratio |
|---|---|---|---|---|---|---|
| #1 | Yes | Yes | No | Yes | DFSID | 2.57 |
| #2 | Yes | Yes | No | Yes | DFSID | none |
| #3 | Yes | Yes | No | Yes | DFSID | 3.00 |
| #1 | Yes | No | No | No | DFSID | 2.31 |
| #2 | Yes | No | No | No | DFSID | 5.34 |
| #3 | Yes | No | No | No | DFSID | 2.80 |

Table 9.4 Initial Comparisons

These crossover size ratios correspond to one of the best realizable grouping schemes employed in this study. For other less successful arrangements, such as the random initial placement, the ratios would be much lower since the LOOM simulation outperformed the paged virtual memory for a wider spectrum of core sizes. As the initial placement becomes worse, the crossover point shifts to larger core sizes, the crossover size ratio falls, and LOOM becomes more attractive. This degradation in the arrangement may also arise from the effects of normal usage. References are copied and updated, new objects are created, and old objects eventually become inaccessible and disappear. Unless efficient and effective dynamic grouping strategies can be found that place new objects and move old objects, the attractiveness of any static grouping will decline. The rate of the decline, the costs of dynamic grouping, and the type of virtual memory will determine the appropriate average usage time before another static regrouping is necessary.

LOOM represents an object-oriented virtual memory with 16-bit in-core pointers. The simulation of the conventional virtual memory swaps pages and uses 32-bit references. A virtual memory containing aspects of both is an object-oriented virtual memory which does not compress pointers. While this virtual memory was not explicitly simulated, we can estimate its performance by noting that pointer compression reduced the total span of a computation by about one-third. This fact may be obtained by finding the ratio of the smallest main memory sizes for which the paged virtual memory with an optimal arrangement and the LOOM simulation with any arrangement did no purging. The utilization of core is 100% in both cases. Neglecting the presence of leaves, this ratio is the average compression of objects and was roughly 1.5 for all three compressed traces. Since all pointers are always resolved, there no page faults due to lambda resolution. If the quantity of

leaves is negligible, results from the LOOM simulation may be transformed into the results of a 32-bit, object-oriented virtual memory by multiplying the core size by 1.5 and removing the number of page faults due to lambda resolution. Comparison of the derived results for this hypothetical virtual memory indicate that pointer compression does not substantially affect the paging performance for LOOM. The validity of these results, which are based upon the estimate of the disk buffer hit rate for resolving lambdas derived from the no-leaf simulations, depend on the accuracy of two simplifications. First, the LOOM simulation did not explicitly maintain the contents of the fields of objects. It used a simple algorithm to decide whether a pointer was a lambda. The accuracy of this model is not known. Second, a naive lambda resolution scheme was employed in the simulator. Other algorithms that can substantially reduce the number of lambdas encountered by the virtual machine may exist. While our judgment of pointer compression remains inconclusive, these results indicate the need for serious study of the tradeoffs involved.

Let the *span* of a computation be the set of both existing and dynamically created objects used in a computation. Two important factors concerning spans need to be addressed before these empirical results can be applied to lengthy user sessions involving computations that use large portions of the set of objects comprising the programming environment. First, the distributions of span size, turnover rate, and degree of commonality must be determined. The effects of system evolution on these distributions must also be considered.

Rough estimates of the span size may be obtained from the initial set of monitored computations that provided data for the detailed analysis of reference behavior presented in Chapter 4. Neglecting the display bitmap and the run-time stack, the computations involved sets of objects that would fit into 20K words of object space. Rough estimates of similarities between computational spans may be garnered from comparisons between warm and cold-start LOOM simulations. Very small core sizes (20K) initially contained some or all of the span that was in core at the end of the warming phase. During the actual simulation following the warming process, some of the in-core objects were used before being purged. Other irrelevant objects were purged without being referenced. The ratio of the sizes of these two sets of objects, as well as the relative costs of fetching and purging objects, determined the differential in page faults between a warm and a cold start. Since warm starts caused both increases and decreases in the number of page faults, one immediate conclusion is that the degree of commonality between two spans depends upon the corresponding computations. Some context shifts will cause little purging and faulting while others will cause moderate amounts of both. Except for the compressed traces, which attempted to change contexts with unnatural haste, no information is currently available regarding span turnover rates in such interactive, display-oriented programming environments. Work needs to be done in this area before an informed evaluation of LOOM can be made.

Two modes of system evolution will also impact the reference behavior of such an object-oriented environment. First of all, Smalltalk is a programming environment that has undergone substantial modification since its inception as Smalltalk-72 [SHOC]. Changes will undoubtedly occur for the foreseeable future, not only by the implementors, but also by the users. Except for the virtual

machine, the entire Smalltalk programming environment is under the immediate control of the user. Both official releases and private systems will evolve with time, meaning that typical user sessions and computations will vary.

The reference behavior of any such system is also dictated by the performance constraints imposed by the hardware, microcode, and software implementing the underlying virtual machine and the top-level system. Performance considerations typically affect the data structures and algorithms chosen to implement a desired feature. Users of the system are constrained not only by the set of existing and constructible tools and properties but also by the feasibility of implementing and running new applications. While larger sizes of primary memory may initially favor a paging scheme by moving the operating region to the right beyond the crossover point, span sizes may drastically increase as larger and more ambitious subsystems evolve. Computations requiring a profusion of intermediate data structures, heavily-accessed personal databases, and large-scale, detailed simulations may become feasible and thus arise in such a memory-rich computing environment. Such applications would shift the operating point to the left and favor a LOOM-like system. Other computations, which were feasible in smaller memory configurations, will continue to be used and will restrict this movement. What is important is the composite effect of system evolution, new applications, and core memory enlargements. Knowledge concerning the dependence of span size on total system size and/or core memory size is required before this net effect can be estimated and the appropriate type of virtual memory ascertained.

## 9.7 An Evaluation of LOOM

Long-term changes in the size of the entire system and individual spans, as well as the short-term fluctuations in computation size and span turnover, will tend to shift the crossover size ratio of the programming environment. The "better" virtual memory depends upon the location and movement of this ratio in addition to the particular cost function of the client.

There are costs to a LOOM system besides paging performance that must be considered. LOOM is a complex system that is much more difficult to fully design, implement, debug, maintain, and modify than a conventional paging scheme. Secondly, much of the computational overhead of a paged virtual memory can be eliminated via hardware assists and/or caching techniques that are readily supported by current machine architectures. Many facets of LOOM, such as lambda resolution, the maintenance of up to three reference counts for each object, and the fact that variable-length sequences of information instead of fixed-length blocks are the units of swapping, require nontrivial algorithms. This complexity will be evident in the additional time, hardware, and/or software required to satisfy requests made to the virtual memory manager. Finally, the assumption that all leaves were clean is not true. Only experience with LOOM will provide the pertinent statistics.

Balancing these costs are the obvious LOOM benefits of a high core utilization, pointer compression, and a relative invariance to the degradation of spatial locality. There are also a number of possible improvements to LOOM and the static grouping algorithm in use.

Grouping strategies may be enhanced by treating code objects in a special manner. The current grouping schemes did not special-case CompiledMethods and were probably penalized. Once a Smalltalk type system becomes reality, such grouping methods will be feasible and should yield substantial improvements in paging performance for LOOM. Packing together code objects without large literals intervening (as was naively done in the OOZE and breadth-first initial placements) should realize gains. Intensive code-grouping efforts will also improve the performance of the paged virtual memory. However, incomplete utilization of CompiledMethods, in addition to primitives handled by the microcode, will limit the reduction in page faults for the paged virtual memory. Open questions include the worth of cleanliness grouping as well as hybrid graph-theoretic schemes that traverse the subtree corresponding to a node in a manner depending on the state of the node and its class.

There are a number of possible improvements to LOOM. For example, empirical results in section 8.3.6 have shown that an LRU ordering in the disk buffer outperformed a FIFO purging scheme. Various lambda-resolution and even object-prefetching schemes need to be implemented and evaluated. Purging policies, such as those that distinguish clean from dirty or young from old objects, also require investigation. Finally, a virtual memory environment such as LOOM, which maintains a strict disk-core separation, may also be used to explore dynamic grouping schemes that place and purge objects related by creation time and/or connectivity.

There is no doubt that LOOM will be an excellent testbed for virtual memory research. Empirical evidence has shown that LOOM will substantially outperform a conventional paged virtual memory for poorly organized initial placements and for memory sizes well below the knee of the page fault function. Small reference counts and similar object usage patterns in wildly different computations have indicated that useful static groupings are efficiently constructed by simple algorithms. In addition, the possibility of dynamic grouping, either by real-time copying [BAKE] or swapping mini-pages [BAER76], the relative ease of implementing a paged virtual memory, and the absence of huge performance gains for moderate and large memory sizes dictate against LOOM.

In addition to primary memory sizes that are small relative to the expected computations and unstable initial placements, two other scenarios strongly favor LOOM. First, some machine architectures "require" short pointers for acceptable performance. Memory and cache word size, register length, and bus and data path widths all constrain the maximum pointer size that the virtual machine can efficiently handle on every object reference. LOOM guarantees that all references seen by the virtual machine are short regardless of the total number of objects in the system. The problem of maintaining a *compact* representation for the set of name-location pairs for recently used objects is solved by LOOM's two separate name spaces. Short of paging an object table (OT) that encompasses the entire system, the corresponding requirement for the paged virtual memory has not been adequately addressed nor analyzed. The time and space overhead of any such scheme will enhance the relative performance of LOOM. This enhancement could be substantial if references to the OT show little spatial locality. Secondly, in relational data base systems and other applications where relationships are indirect and not by pointers, reference counts and pointers are meaningless

because objects can never become inaccessible and the boolean connectivity of the data does not constrain the motion of queries in data space. If high quality initial placements that are independent of queries cannot be found, then any static arrangement will be comparable with the random initial placement used in the simulations and a LOOM-like system will be preferable.

# 10. Dynamic Characteristics and Degradation of Initial Placements

## 10.1 Introduction

A final set of measurements were made using the Smalltalk-80 emulator in order to determine the relative amount of time computations spend updating the structure of objects grouped into some initial placement. Besides indicating the level of degradation of the initial placement in a short computation, this data may be used to evaluate and design memory management policies for dynamically created objects, since temporary structures cause most of the memory management workload. These studies concentrated on memory allocation and reclamation, object lifetimes, and dynamic pointer distances. Data from which the following discussion draws may be found in Appendix D.

Unlike all preceding monitored simulations, the run-time stack was included in this analysis. The contexts comprising the stack are ordinary Smalltalk objects whose behavior accounts for much of the dynamic characteristics of the system. Requests for new objects were explicitly caught and processed, while object destructions were noticed by checking the reference count of an object whenever a pointer to it was clobbered. Except for initializing the pointer fields of a new object to nil, all pointer field updates were tallied and processed. On the other hand, the act of reading and following a pointer value was neglected by the statistical package.

Five distinct computations were monitored, including sending an error message, displaying text in a window on the screen, determining the set of classes prepared to accept a specific message selector, compiling a method, and opening a browser. Since opening a browser required an enormous number of bytecodes, that sequence was arbitrarily halted. All other computations were run to completion.

Simple counts of events indicate that most of the effects of the computation were transient and did not substantially restructure the graph of existing objects. This fact highlights the futility of investigating initial placement stability via detailed simulations. In order to have even a small effect on the set of objects in secondary storage, an actual running system or a high-level simulator must be employed. Nevertheless, a number of interesting observations were made concerning detailed aspects of object lifetimes and characteristics of new pointers.

The total number of store operations was slightly larger than the number of bytecodes executed. On the average, one pointer was updated for every bytecode interpreted by the virtual machine. Although thousands of object instances were created, few remained at the end of the computation. A small number of old objects that had existed before the computation began also perished.

### 10.2 Stack-like Allocation and Deallocation

One view of the behavior of dynamically created objects is that their existence is stack-like. According to this model, given a set of objects used by some computation, the object in that set that is the most likely to perish is the youngest. While this view was not entirely validated, the lifetime behavior of most objects seemed to follow a stack-like discipline. A related question suggested by the analysis is the correlation between the deallocation of a stack frame and the deallocation of objects created in that stack frame.

A stack of existing dynamically created objects was maintained to determine the validity of this model. Newly created objects were pushed onto this stack while freed objects were removed. If the destroyed object was not at the top of the stack, then nothing else occurred. Otherwise, the stack was repeatedly popped until its top again referred to an existing object.

In general, the maximum stack length for a trace was only slightly longer than the stack size at the end of the computation. Most of the stack was empty, since the number of existing objects was much less than the stack length. A small number of objects violated the assumption of stack-like lifetimes and caused these conditions. Without compaction, the stack size grew as the computation proceeded and would tend to grow without bound for arbitrarily long computations.

To remedy this situation, compaction was assumed to be performed continuously. When an object was deallocated, its distance from the top of the stack was recorded. Call this offset the *long deallocation distance.* This object was removed from the stack and all younger objects were moved one slot to fill the gap. A perfect stack-like behavior would cause only zeros to be recorded and no movement of younger objects. Although some empirical distances were nonzero, the third quartiles and means of the long deallocation distance distribution were either zero or one. The durations of most objects followed a stack discipline.

All objects involved in the computation that were not created dynamically were assumed to be *permanent* or *old.* Whenever a permanent object is given a reference to a new object, there is a high probability that the death of the new object and all its descendants will seriously violate the stack ordering. To account for this likelihood, whenever a reference to a new object was given to an old object, the new object and any of its descendants that were new were removed from the stack and became old. Again, younger objects moved as many stack slots as necessary in order to eliminate the resulting empty spaces. The obvious parallel in an actual implementation would be the copying of the entire new structure from *new space* to *old space.* In this scenario, the stack only contains references to new objects that are not referenced directly nor indirectly by old objects. Pointers are unconstrained in either space and pointers may go from new space to old space but not vice versa. When an object in new space perishes, call its offset from the top of the stack its *short deallocation distance.* For each trace, the long and short deallocation distance distributions were quite similar. Removing these objects from the stack had only a negligible effect on the quartiles, means, and medians of the deallocation distance functions. In two cases, however, the maximum deallocation distance was drastically reduced.

At the end of each computation, less than 2% of all dynamically created objects still existed. However, at least 75% of the words copied from new space to old space still belonged to existing objects. Most of the copying was not done in vain. Except for the fifth trace, which represented an uncompleted computation, at least 90% of the remaining new objects were automatically transferred to old space by the aforementioned algorithm. This simple predictive mechanism had a high degree of accuracy and may thus be profitably used in memory management schemes that distinguish between new space and old space.

### 10.3 Object Lifetime

Consider a clock associated with the Smalltalk virtual machine that ticks once for each (extended) bytecode executed. Assume a new object A is created at time $t_c$ and becomes inaccessible at time $t_d$. Define the lifetime of A to be $t_d$-$t_c$. If a computation c spans at least the period from $t_c$ to $t_d$, then A is a *transient* object in computation c. Logging allocation requests and carefully monitoring reference counts allowed lifetimes for all transient objects to be determined. Lifetime distributions for the computation sequences were highly skewed and had medians in the range from 13 to 30 and means in the interval from 264 to 696 (Figure 10-1). Means much larger than even the third quartile were the result of both long-lived stack frames and objects, as well as local objects passed by result up the call stack. These results are not surprising since a majority of objects had stack-like lifetimes. Batson and Brundage [BATS77] reported similar findings of skewed lifetime distributions for arrays and contours (stack frames) in an empirical study of Algol programs.

### 10.4 Dynamic Pointer Distance

Associated with each new object is its time of creation. Let the creation time for all permanent objects be minus infinity. For two objects A and B with respective creation times $t_a$ and $t_b$, assume A contains a non-immediate reference to B. Hence B is not a SmallInteger nor the "nil" object. Let the *dynamic distance* of this pointer from A to B be $t_a$-$t_b$. Note that positive distances point backward in time while negative distances point forward in time. If B is old and A is not, then the distance is plus infinity. If A is old and B is not, the distance is minus infinity. If both are new, the distance is determined by ordinary subtraction. Otherwise, both objects are old and the distance is undefined.

Based upon the nature of the CONS operation and the infrequent use of RPLACA and RPLACD operators, researchers in Lisp systems have predicted that most dynamic pointer distances will be positive [LIEB]. That is, most references will be from a list cell to an older list cell. These assumptions are not directly applicable to Smalltalk because of the high use of primitives and CompiledMethods that update old pointer fields and the fact that stack frames are ordinary objects.

The dynamic distance of the new reference in each store operation was recorded for the five monitored computation sequences. Not surprisingly, the bulk of the store operations updated new objects. However, slightly more than half of these pointers that were non-immediate referred to permanent objects. Call these operations *new-old stores*. Every MethodContext frame on the stack

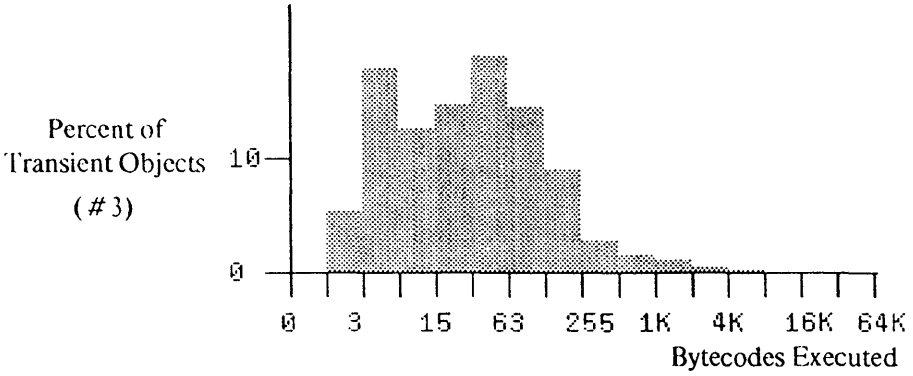


Figure 10-1 Transient Object Lifetimes

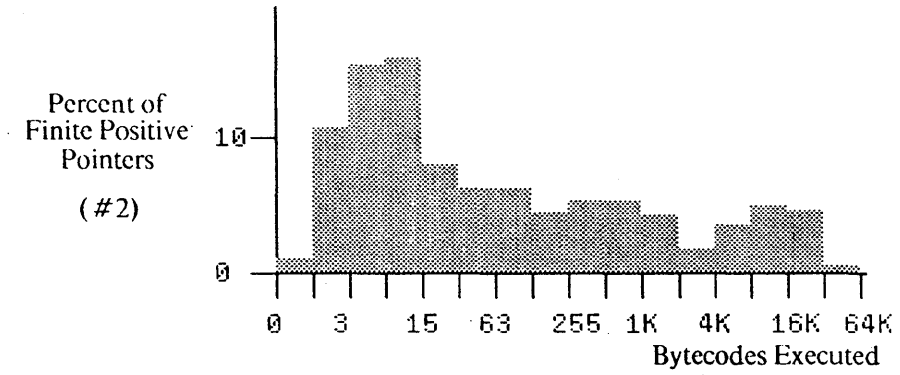


Figure 10-2 Dynamic Positive Pointer Distance

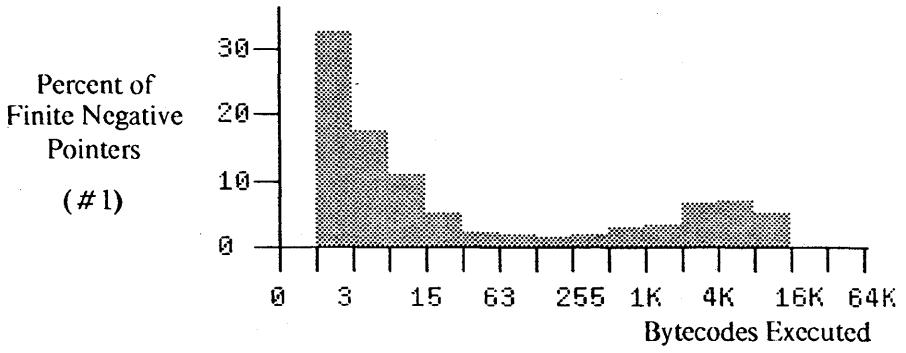


Figure 10-3 Dynamic Negative Pointer Distance

referred to an existing CompiledMethod and hence code references contributed a substantial number of new-old stores. All global variables, method literals, and classes were old and caused additional new-old references. These pointers were primarily used in a read-only mode, since very few old objects were ever updated or destroyed. The small number of store operations that created links from permanent objects to new structures severely limited the required amount of copying from new space to old space.

Self-reference, which arose when a pointer to the current stack frame was pushed on the stack, caused a few references to have a dynamic distance of zero. Since stack frames were necessarily created before local objects were defined, creating a new object and pushing it on the stack usually resulted in a reference with a small but negative dynamic distance. On the other hand, all message arguments were necessarily created before the corresponding stack frame was activated. Pushing an argument onto the stack usually resulted in a small but positive dynamic distance. This positive distance would tend to balance the small negative distance recorded when the same argument was first pushed on the stack in the calling environment. Unlike a Lisp-like system, where one expects almost all references to point backwards in time, references in new Smalltalk objects pointed forward in time quite often. For example, the ratio of finite positive pointers to finite negative pointers ranged from 1.59 to 6.85. Pointers from new space to old space have a positive infinite distance. When these infinite pointers are also considered, the interval ranged from 4.40 to 17.47. These ratios indicate that pointers forward in time may be common but in general are exceptions rather than the rule.

While most statistics for negative pointer distances were smaller than their positive counterparts, all pointer distance distributions were skewed (Figures 10-2 and 10-3). In 9 of 10 cases, for example, the mean distance was far larger than the median. Mean positive pointer distances were in the interval from 1832 to 10,417, while negative pointers had mean distances from -599 to -3897. Median pointer distances were much smaller in magnitude and fell into the ranges from 8 to 375 and from -2 to -12.

### 10.5 Degradation of an Initial Placement

The preceding analysis indicates that efforts at improving algorithms that manage primary memory ought to concentrate on the set of newly created objects and operations contained within this set. A simple transport rule, which copies objects from new space to old space, filters out a significant portion of the long-lived objects. A conventional mark and sweep garbage collector in new space would perform well at the end of a computation, because few accessible objects would remain in new space. One penalty, however, is the initial requirement of enough free space to allow the computation to run to completion without reclaiming core. A second consideration is the overhead of checking for old-new pointers.

While emphasizing the asymmetry in the update frequency of new and old objects, the speed limitations of the Smalltalk emulator prohibited a full investigation into the rate at which an initial placement degrades with usage. There are two key methods by which an initial placement will

evolve over time. First, old objects are updated and destroyed at a relatively slow rate. This process corresponds to a gradual modification of the underlying graph structure and the elimination of unconnected components. A second aspect of this modification is the introduction of new permanent objects that were dynamically created, matured, and forced to the disk.

Snapshots of an in-core version of Smalltalk were made once every four hours over a twelve-hour usage span. Sessions involving the implementation, debugging, and usage of the software package that tallied and processed the dynamic events reported in this chapter comprised the bulk of the usage. A depth-first initial placement was computed for the objects of the first snapshot. This placement was used to calculate static pointer distances and the fraction of these distances that were less than 256 words in all four snapshots. The four distance distributions were quite similar, while the on-page pointer ratio slowly declined from 16.9% to 16.1%. As measured by static pointer distance distributions, this initial placement degraded only slightly after a moderate amount of use.

Needless to say, these measures only considered objects that occurred in both the initial placement and the specific snapshot. Objects present in a snapshot but not in the initial placement were not included in the analysis. This single sequence of snapshots is not indicative of a user modifying more than a small collection of classes. Most of the computation was concentrated in a small part of the system, which may or may not be typical. Work is needed to determine the net effect of new objects matured to the disk and to design, implement, and test a variety of dynamic grouping algorithms that handle the transfer, placement, and movement of new permanent objects. Only then will the true degradation rates of initial placements and the need for periodic static regroupings be known.

# 11. Conclusions

The key finding of this thesis arose from a direct empirical comparison between object-oriented and page-swapping virtual memories. In terms of the number of page faults, LOOM outperformed a conventional virtual memory for a range of small memory sizes that depended on the quality of the initial placement generated by the grouping strategy. The better virtual memory design depends on the existence of quality initial placements and the rate at which they become obsolete.

In order to review the remaining major conclusions of this thesis, I have included a recapitulation of each chapter as well as warnings for areas not addressed. After discussing LOOM, this chapter provides suggestions for future research.

## 11.1 A Review

Chapter 2 provided a quick introduction to the LOOM virtual memory and the Smalltalk system. While LOOM logically swaps objects between memory levels, it always transfers physical pages between core and disk. Under certain circumstances, objects are the appropriate entities to be grouped when restructuring information in the secondary memory to enhance paging performance. On the average, objects should be smaller than a single disk page but not much more than an order of magnitude smaller.

The emulator for the Smalltalk-80 virtual machine that was constructed to produce actual execution traces was described in Chapter 3. While the emulator did not generate all references made by the virtual machine, the missing fraction was negligible. The traces discussed in this thesis are therefore representative and complete.

Chapter 4 investigated the basic reference tendencies of Smalltalk by examining event traces for five diverse operations that frequently occur in typical user sessions. Some statistics highlighted the differences between the access patterns of code objects from those of data objects. However, these distinctions did not warrant special treatment by cache management schemes, main memory managers, or grouping strategies. Object size distributions supported the claim that objects are the appropriate entity to group on disk pages. The observed locality of reference predicted a substantial reduction in the size of event traces by simple compression schemes.

The next chapter presented nine static grouping schemes. A static analysis of the initial placements generated by these grouping algorithms partitioned them into five categories. Under two static measures of similarity, initial placements within a category were remarkably alike, while those in different categories were not alike. This partitioning extended somewhat to the dynamic domain. Initial placements in the same category had similar paging performances. The converse of this statement, however, did not always hold.

Chapter 6 described a one-pass compression algorithm that reduced the length of event traces intended to be used as input to virtual memory simulations. If a few simple requirements are met by the simulation, this algorithm guarantees a weak equivalence between the initial and compressed reference traces. A reduction ratio of 100:1 was attained.

The effects of static grouping on the dynamic performance of a page-swapping virtual memory were presented in Chapter 7. Any of the realizable grouping strategies substantially reduced the number of page faults caused by an ungrouped initial placement. Performance differences between the grouping schemes were not relatively significant. The average utilization of main memory, as opposed to the particular grouping strategy, was the dominant factor.

Chapter 8 described the effects of static grouping on the object-oriented virtual memory LOOM. These grouping schemes realized limited performance gains, since LOOM was able to perform rather well in the face of an ungrouped initial placement. Other virtual memory policies and parameters played a secondary role in determining the number of page faults and were essentially independent of the initial placement.

The following chapter reported the direct empirical comparison between the object-oriented and page-swapping virtual memories. In terms of the number of page faults, LOOM outperformed the conventional virtual memory for a range of small memory sizes. Similar intervals were obtained for warm starts, cold starts, simulations that did reference counting, and those that did not. However, the length of this interval varied inversely with the quality of the initial placement.

Chapter 10 briefly examined the memory management problems of Smalltalk and pointed out the need to efficiently handle dynamically created objects, since most have extremely short lifetimes. Although most updated references in newly created objects pointed backwards in time, object lifetimes were not governed by a strict stack discipline.

While the Smalltalk-80 emulator provided these novel results, its nature prohibited the investigation of other related areas. Computation size, turnover rate, and composition could not be easily measured. The empirical comparisons between the two types of virtual memories could not be directly used to answer hard questions concerning real or imagined systems. Issues of initial placement stability were not thoroughly addressed, and the entire domain of dynamic grouping was avoided.

## 11.2 Recommendations

Inherent difficulties in implementing an object-swapping scheme must be considered before the choice of a virtual memory type is finalized. Potential performance benefits provided by LOOM must be carefully weighed against the costs. Such an analysis considers the interactions between the physical and virtual environments. Primary memory size is the one important physical parameter. The size and nature of the current and future computations, the existence of good grouping algorithms, and the stability of quality initial placements are the remaining inputs. These three

factors determine the nominal position of the crossover point, the short term fluctuations about this point, and the long-term trends that translate this point. A comparison between the distribution of the crossover point locations and the physical memory size will indicate the relative paging performance an object-oriented virtual memory would be expected to provide.

LOOM should not be viewed as a quick virtual memory implementation that will function as a panacea. Instead, such a scheme must be considered as part of a long-term research project whose success cannot be evaluated without a running prototype. An actual implementation of LOOM can be used to validate or invalidate the results presented in this thesis and explore the issues side-stepped by simplifications. Of critical importance to a page-swapping virtual memory are the questions of initial placement stability and the existence of efficient dynamic grouping schemes. A rapid degradation of the initial placement in the direction of a random grouping will substantially shift the crossover point. A real system will not be burdened by the speed constraints encountered in emulations and simulations and can measure this degradation rate. Experimentation can easily go beyond the bounds established by this study in attempting to solve these problems. If good arrangements are inherently stable or effective dynamic algorithms can be found, then the advantages of LOOM will be more than offset by the costs for memory sizes comparable to the expected span of computations.

## 11.3 Directions for Future Research

Simulations of LOOM have identified a number of important considerations that can substantially affect the paging performance of object-swapping virtual memories. Lambda resolution schemes, object-prefetching policies, and purging algorithms need to be closely examined in isolation, with respect to each other, and in conjunction with different types of initial placements.

The static grouping algorithms employed in this study emphasized the object-fetching behavior of the virtual machine using either a priori or a posteriori knowledge. Just as important is the minimization of disk accesses for purging policies and lambda resolution schemes. New restructuring techniques that are cognizant of these characteristics should be designed, evaluated, and compared with the set of simple algorithms used here.

The nature and extent of dynamic modification to the initial placement needs to be investigated. New objects are constantly matured and added to the set of permanent objects. Fields are continuously updated and old objects eventually perish. Dynamic grouping [BAER76], copying [BAKE, SVOB], and maturing techniques deserve an examination for both object-oriented and page-swapping virtual memories.

The effects of memory reclamation algorithms, such as reference counting, [real-time] garbage collection, and hybrid schemes, will certainly depend on the initial placement and virtual memory type. It may be the case that both static and dynamic grouping schemes should be tuned to the anticipated memory management policies.

Finally, the experiments and measurements performed on Smalltalk-80 ought to be duplicated on other object-oriented, interactive programming environments in order to determine whether the reported results apply to only one system or characterize general computational tendencies.

## Bibliography

[ACM] ACM. Proceedings of a Symposium on Storage Allocation. *Communications of the ACM*. 4(10): 1961 October.

[BAER72] Baer, J. L.; Caughy, R. Segmentation and Optimization of Programs from Cyclic Structure Analysis. *AFIPS Proceedings, Spring Joint Computer Conference*. 40: 23-36; 1972.

[BAER76] Baer, J. L.; Sager, G. R. Dynamic Improvement of Locality in Virtual Memory Systems. *IEEE Transactions on Software Engineering*. SE-2(1): 54-62; 1976 March.

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

[BATS70] Batson, A. P.; Ju, S.-M.; Wood, D. C. Measurements of Segment Size. *Communications of the ACM*. 13(3): 155-159; 1970 March.

[BATS76] Batson, A. P. Program Behavior at the Symbolic Level. *Computer*. 9(11): 21-26; 1976 November.

[BATS77] Batson, A. P.; Brundage, R. E. Segment Sizes and Lifetimes in Algol 60 Programs. *Communications of the ACM*. 20(1): 36-44; 1977 January.

[BELA66] Belady, L. A. A Study of Replacement Algorithms for a Virtual Storage Computer. *IBM Systems J.* 5(2): 78-101; 1966.

[BELA69] Belady, L. A.; Kuehner, C. J. Space Sharing in Computer Systems. *Communications of the ACM*. 12(5): 282-288; 1969 May.

[BISH] Bishop, P. B. *Computer Systems with a Very Large Address Space and Garbage Collection*. Cambridge, MA: M.I.T. Laboratory for Computer Science; 1977 May. MIT/LCS/TR-178.

[BRAW68] Brawn, B. S.; Gustavson, F. G. Program Behavior in a Paging Environment. *AFIPS Proceedings, Fall Joint Computer Conference*. 33(2): 1019-1032; 1968.

[BRAW70] Brawn, B. S.; Gustavson, F. G.; Mankin, E. S. Sorting in a Paging Environment. *Communications of the ACM*. 13(8): 483-494; 1970 August.

[CLAR76] Clark, D. W. *List Structure: Measurements, Algorithms, and Encodings*. Pittsburgh, PA: Carnegie-Mellon University, Dept. of Computer Science; 1976 August.

[CLAR77] Clark, D. W.; Green, C. C. An Empirical Study of List Structure in Lisp. *Communications of the ACM*. 20(2): 78-86; 1977 February.

[CLAR78] Clark, D. W.; Green, C. C. A Note on Shared List Structure in Lisp. *Information Processing Letters*. 7(6): 312-314; 1978 October, 1978.

[COFF] Coffman, E. G., Jr.; Denning, P. J. *Operating Systems Theory.* Englewood Cliffs, NJ: Prentice-Hall; 1973.

[COME] Comeau, L. W. A Study of the Effect of User Program Optimization in a Paging System. *ACM Proceedings, Symposium on Operating Systems Principles.* Gatlinberg, Tenn. 1967.

[DENN] Denning, P. J. Virtual Memory. *Computing Surveys.* 2(3): 153-189; 1970 September.

[FERR74] Ferrari, D. Improving Locality by Critical Working Sets. *Communications of the ACM.* 17(11): 614-620; 1974 November.

[FERR76] Ferrari, D. The Improvement of Program Behavior. *Computer.* 9(11): 39-47; 1976 November.

[FERR78] Ferrari, D. *Computer Systems Performance Evaluation.* Englewood Cliffs, NJ: Prentice-Hall; 1978.

[FINE] Fine, G. H.; Jackson, C. W.; McIsaac, P. V. Dynamic Program Behavior under Paging. *Proceedings of the 21st National ACM Conference.* P-66: 223-228; 1966.

[GOLD] Goldstein, I. P.; Bobrow, D. G. *A Layered Approach to Software Design.* Palo Alto, CA: Xerox PARC, Computer Science Laboratory; 1980 December. CSL-80-5.

[GUER] Guertin, R. L. Programming in a Paging Environment. *Datamation.* 18(2): 48-55; 1972 February.

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

[INFO] Informatics, Inc. *Experiments in Automatic Paging.* Griffiss AFB, NY: Rome Air Development Center, Air Force Systems Command; 1971 November. RADC-TR-71-231.

[INGA78] Ingalls, D. H. H. The Smalltalk-76 Programming System: Design and Implementation. *Conference Record, Fifth Annual ACM Symposium on Principles of Programming Languages*: 9-16; 1978 January.

[INGA81] Ingalls, D. H. H. The Smalltalk Graphics Kernel. *Byte.* 6(8): 168-194; 1981 August.

[JOHN] Johnson, J. W. *Program Restructuring for Virtual Memory Systems.* Cambridge, MA: M.I.T. Laboratory for Computer Science; 1975 March. MIT/LCS/TR-148.

[KAEH81] Kaehler, T. Virtual Memory for an Object-Oriented Language. *Byte* 6(8): 378-387; 1981 August.

[KAEH] Kaehler, T. Working paper on the Smalltalk-80 virtual memory system. To appear.

[KERN] Kernighan, B. W. Optimal Sequential Partitions of Graphs. *Journal of the ACM.* 18(1): 34-40; 1971 January.

[KUCK] Kuck, D. J.; Lawrie, D. H. The Use and Performance of Memory Hierarchies: A Survey. Tou, J. T., ed. *Conference Proceedings, Computer and Information Sciences III.* New York: Academic Press; 1970: 45-77.

[KUEH] Kuehner, C. J.; Randell, B. Demand Paging in Perspective. *AFIPS Proceedings, Fall Joint Computer Conference.* 33(2): 1011-1018; 1968.

[LEWI] Lewis, P. A. W.; Yue, P. C. Statistical Analysis of Program Reference Patterns in a Paging Environment. *Conference Digest of the 5th IEEE International Computer Society Conference.* 71C41-C: 133-134; 1971 September.

[LIEB] Lieberman, H.; Hewitt, C. *A Real Time Garbage Collector That Can Recover Temporary Storage Quickly.* Cambridge, MA: M.I.T. Artificial Intelligence Laboratory; 1980 April. Memo 569.

[LISK] Liskov, B. et al. *CLU Reference Manual.* Cambridge, MA: M.I.T. Laboratory for Computer Science; 1979 October. MIT/LCS/TR-225.

[LOWE] Lowe, T. C. Automatic Segmentation of Cyclic Program Structures Based on Connectivity and Processor Timing. *Communications of the ACM.* 13(1): 3-9; 1970 January.

[LUNI] Luniewski, A. W. *The Architecture of an Object Based Personal Computer.* Cambridge, MA: M.I.T. Laboratory for Computer Science; 1979 December. MIT/LCS/TR-232.

[MATT] Mattson, R. L. et al. Evaluation Techniques for Storage Hierarchies. *IBM Systems J.* 9(2): 78-117; 1970.

[McKEE] McKeeman, W. M. Language Directed Computer Design. *AFIPS Proceedings, Fall Joint Computer Conference.* 31: 413-417; 1967.

[McKEL] McKellar, A. C.; Coffman, E. G., Jr. Organizing Matrices and Matrix Operations for Paged Memory Systems. *Communications of the ACM.* 12(3): 153-164; 1969 March.

[METC] Metcalfe, R. M.; Boggs, D. R. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM.* 19(7): 395-404; 1976 July.

[MORR] Morrison, J. E. User Program Performance in Virtual Storage Systems. *IBM Systems J.* 12(3): 216-237; 1973.

[O'NEI] O'Neill, R. W. Experience Using a Time-Shared Multi-Programming System with Dynamic Address Relocation Hardware. *AFIPS Proceedings, Spring Joint Computer Conference.* 30: 611-621; 1967.

[PARM] Parmelee, R. P. et al. Virtual Storage and Virtual Machine Concepts. *IBM Systems J.* 11(2): 99-130; 1972.

[RAMA] Ramamoorthy, C. V. The Analytic Design of a Dynamic Look Ahead and Program Segmentation System for Multiprogrammed Computers. *Proceedings of the 21st National ACM Conference.* P-66: 229-239; 1966.

[RAND] Randell, B. A Note on Storage Fragmentation and Program Segmentation. *Communications of the ACM.* 12(7): 365-372; 1969 July.

[SALT] Saltzer, J. H. On the Modeling of Paging Algorithms. *Communications of the ACM.* 19(5): 307-308; 1976 May.

[SAYR] Sayre, D. Is Automatic "Folding" of Programs Efficient Enough to Displace Manual? *Communications of the ACM.* 12(12) 656-660; 1969 December.

[SHOC] Shoch, J. An Overview of the Programming Language Smalltalk-72. *SIGPLAN Notices.* 14(9): 64-73; 1979 September.

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

[SNYDr] Snyder, R. On A Priori Program Restructuring for Virtual Memory Systems. Lanciaux, D., ed. *Operating Systems: Theory and Practice.* New York: North-Holland; 1979: 207-224.

[SPIR] Spirn, J. R.; Denning, P. J. Experiments with Program Locality. *AFIPS Proceedings, Spring Joint Computer Conference.* 40: 611-621; 1972.

[SVOB] Svobodova, L. *Management of Object Histories in the SWALLOW Repository.* Cambridge, MA: M.I.T. Laboratory for Computer Science; 1980 July. MIT/LCS/TR-243.

[SWIN] Swinehart, D.; McDaniel, G.; Boggs, D. R. *WFS: A Simple File System for a Distributed Environment.* Palo Alto, CA: Xerox PARC, Computer Science Laboratory; 1979 October. CSL-79-13.

[TSAO] Tsao, R. F.; Comeau, L. W.; Margolin, B. H. A Multi-Factor Paging Experiment: I. The Experiment and the Conclusions. Freiberger, W., ed. *Statistical Computer Performance Evaluation.* New York: Academic Press; 1972: 103-134.

[VERH] Ver Hoef, E. W. Automatic Program Segmentation Based on Boolean Connectivity. *AFIPS Proceedings, Spring Joint Computer Conference.* 38: 491-495; 1971.

[XERO] Xerox Learning Research Group. The Smalltalk-80 System. *Byte.* 6(8): 36-48; 1981 August.

# Appendix A

## A-0. Basic Data

| Trace Number | All References | Code References | Data References | All Primitives | Primitives Caught | Primitive Percentage |
|---|---|---|---|---|---|---|
| #1 | 144K | 81,157 | 66,299 | 7543 | 6327 | 83.9% |
| #2 | 144K | 93,358 | 54,098 | 6242 | 5829 | 93.4% |
| #3 | 144K | 102,161 | 45,295 | 6791 | 5688 | 83.8% |
| #4 | 144K | 95,256 | 52,200 | 8446 | 7544 | 89.3% |
| #5 | 144K | 36,456 | 111,000 | 2139 | 1883 | 88.0% |

| Trace Number | ByteCodes Executed | All Objects | Size of All Objects | New Objects | Size of New Objects |
|---|---|---|---|---|---|
| #1 | 39,527 | 705 | 49,214* | 89 | 1442 |
| #2 | 32,556 | 1240 | 52,427* | 393 | 2381 |
| #3 | 37,042 | 878 | 18,079 | 186 | 1542 |
| #4 | 40,000 | 723 | 51,099* | 85 | 2625 |
| #5 | 14,044 | 422 | 46,761* | 100 | 899 |

*Includes the 32K bitmap

## A-1. Size Distribution

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 2 | 32K | 10 | 5 | 17 | 25 | 354 |
| All <2K | 2 | 1593 | 10 | 5 | 17 | 19 | 48 |
| All Code | 4 | 1201 | 17 | 12 | 34 | 31 | 47 |
| All Code <2K | 4 | 1201 | 17 | 12 | 34 | 31 | 47 |
| All Data | 2 | 32K | 8 | 4 | 14 | 22 | 414 |
| All Data <2K | 2 | 1593 | 8 | 4 | 13 | 14 | 47 |

Static Size Distribution

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All <2K | 2 | 502 | 10 | 4 | 20 | 24 | 50 |
| All Code <2K | 4 | 502 | 18 | 10 | 36 | 34 | 53 |
| All Data <2K | 2 | 485 | 5 | 4 | 10 | 14 | 45 |

Quasi-Static Size Distribution #1

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All <2K | 2 | 502 | 9 | 4 | 17 | 16 | 30 |
| All Code <2K | 4 | 502 | 18 | 10 | 36 | 32 | 47 |
| All Data <2K | 2 | 258 | 5 | 4 | 10 | 9 | 15 |

Quasi-Static Size Distribution #2

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All <2K | 2 | 515 | 13 | 5 | 20 | 21 | 36 |
| All Code <2K | 4 | 260 | 20 | 11 | 36 | 33 | 40 |
| All Data <2K | 2 | 515 | 10 | 5 | 20 | 15 | 33 |

Quasi-Static Size Distribution #3

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All <2K | 2 | 530 | 12 | 5 | 24 | 26 | 52 |
| All Code <2K | 4 | 473 | 19 | 12 | 36 | 33 | 43 |
| All Data <2K | 2 | 530 | 5 | 4 | 10 | 18 | 59 |

Quasi-Static Size Distribution #4

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All <2K | 2 | 502 | 10 | 4 | 21 | 25 | 48 |
| All Code <2K | 4 | 502 | 16 | 7 | 36 | 40 | 63 |
| All Data <2K | 2 | 258 | 4 | 4 | 10 | 11 | 21 |

Quasi-Static Size Distribution #5

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 2 | 32K | 41 | 14 | 182 | 5864 | 12,370 |
| All <2K | 2 | 502 | 27 | 11 | 68 | 67 | 98 |
| All Code | 4 | 502 | 41 | 25 | 116 | 74 | 89 |
| All Code <2K | 4 | 502 | 41 | 25 | 116 | 74 | 89 |
| All Data | 2 | 32K | 41 | 10 | 32K | 12,953 | 15,780 |
| All Data <2K | 2 | 485 | 10 | 5 | 34 | 52 | 113 |

Dynamic Size Distribution #1

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 2 | 32K | 34 | 10 | 68 | 2615 | 8711 |
| All <2K | 2 | 502 | 27 | 10 | 43 | 54 | 80 |
| All Code | 4 | 502 | 36 | 22 | 68 | 66 | 85 |
| All Code <2K | 4 | 502 | 36 | 22 | 68 | 66 | 85 |
| All Data | 2 | 32K | 10 | 10 | 258 | 7013 | 13,277 |
| All Data <2K | 2 | 260 | 10 | 10 | 10 | 29 | 60 |

Dynamic Size Distribution #2

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 2 | 515 | 27 | 10 | 65 | 51 | 66 |
| All <2K | 2 | 515 | 27 | 10 | 65 | 51 | 66 |
| All Code | 4 | 260 | 36 | 25 | 68 | 65 | 68 |
| All Code <2K | 4 | 260 | 36 | 25 | 68 | 65 | 68 |
| All Data | 2 | 515 | 10 | 10 | 10 | 19 | 46 |
| All Data <2K | 2 | 515 | 10 | 10 | 10 | 19 | 46 |

Dynamic Size Distribution #3

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 2 | 32K | 33 | 14 | 70 | 3175 | 9513 |
| All <2K | 2 | 530 | 27 | 12 | 68 | 64 | 99 |
| All Code | 4 | 473 | 34 | 25 | 68 | 64 | 80 |
| All Code <2K | 4 | 473 | 34 | 25 | 68 | 64 | 80 |
| All Data | 2 | 32K | 10 | 10 | 32K | 8852 | 14,343 |
| All Data <2K | 2 | 530 | 10 | 10 | 34 | 62 | 134 |

Dynamic Size Distribution #4

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 2 | 32K | 32K | 68 | 32K | 21,204 | 15,278 |
| All <2K | 2 | 502 | 30 | 10 | 68 | 80 | 127 |
| All Code | 4 | 502 | 40 | 20 | 116 | 103 | 141 |
| All Code <2K | 4 | 502 | 40 | 20 | 116 | 103 | 141 |
| All Data | 2 | 32K | 32K | 32K | 32K | 28,134 | 10,761 |
| All Data <2K | 2 | 258 | 10 | 10 | 20 | 19 | 32 |

Dynamic Size Distribution #5

### A-2. Fractional Utilization

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 0 | 100 | 50 | 20 | 68 | 45 | 28 |
| All Code | 3 | 92 | 54 | 25 | 68 | 48 | 24 |
| All Data | 0 | 100 | 33 | 20 | 71 | 43 | 30 |

Fractional Utilization #1

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 2 | 100 | 50 | 10 | 75 | 45 | 33 |
| All Code | 3 | 93 | 51 | 15 | 68 | 48 | 25 |
| All Data | 2 | 100 | 33 | 10 | 75 | 45 | 34 |

Fractional Utilization #2

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 0 | 100 | 29 | 7 | 70 | 39 | 32 |
| All Code | 3 | 91 | 57 | 40 | 73 | 53 | 25 |
| All Data | 0 | 100 | 25 | 5 | 67 | 35 | 32 |

Fractional Utilization #3

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 0 | 100 | 40 | 20 | 74 | 44 | 29 |
| All Code | 3 | 89 | 54 | 35 | 67 | 49 | 23 |
| All Data | 0 | 100 | 30 | 10 | 75 | 41 | 32 |

Fractional Utilization #4

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 1 | 100 | 59 | 20 | 75 | 51 | 30 |
| All Code | 3 | 94 | 45 | 14 | 60 | 42 | 26 |
| All Data | 1 | 100 | 75 | 25 | 75 | 56 | 31 |

Fractional Utilization #5

| Trace Length | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 1K | 5 | 77 | 24 | 14 | 57 | 33 | 24 |
| 2K | 5 | 100 | 25 | 13 | 57 | 36 | 26 |
| 4K | 5 | 100 | 37 | 15 | 68 | 40 | 26 |
| 8K | 5 | 100 | 37 | 15 | 68 | 40 | 26 |
| 16K | 5 | 100 | 40 | 20 | 74 | 42 | 27 |
| 32K | 0 | 100 | 40 | 20 | 75 | 44 | 27 |
| 64K | 0 | 100 | 47 | 20 | 75 | 45 | 28 |
| 144K | 0 | 100 | 50 | 20 | 68 | 45 | 28 |

Incremental Fractional Utilization #1 (All)

| Trace Length | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 1K | 0 | 100 | 20 | 14 | 45 | 31 | 23 |
| 2K | 0 | 100 | 25 | 20 | 55 | 34 | 24 |
| 4K | 0 | 100 | 30 | 20 | 64 | 38 | 25 |
| 8K | 0 | 100 | 30 | 20 | 73 | 41 | 27 |
| 16K | 0 | 100 | 40 | 17 | 73 | 42 | 27 |
| 32K | 2 | 100 | 44 | 20 | 75 | 45 | 28 |
| 64K | 2 | 100 | 40 | 13 | 75 | 45 | 33 |
| 144K | 2 | 100 | 50 | 10 | 75 | 45 | 33 |

Incremental Fractional Utilization #2 (All)

| Trace Length | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 1K | 0 | 80 | 25 | 14 | 45 | 31 | 20 |
| 2K | 2 | 80 | 25 | 14 | 50 | 33 | 21 |
| 4K | 2 | 86 | 25 | 14 | 60 | 36 | 25 |
| 8K | 3 | 100 | 30 | 14 | 67 | 39 | 28 |
| 16K | 0 | 100 | 37 | 14 | 68 | 42 | 31 |
| 32K | 0 | 100 | 39 | 14 | 76 | 43 | 32 |
| 64K | 0 | 100 | 40 | 13 | 75 | 44 | 31 |
| 144K | 0 | 100 | 29 | 7 | 70 | 39 | 32 |

Incremental Fractional Utilization #3 (All)

| Trace Length | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 1K | 0 | 75 | 20 | 14 | 43 | 29 | 20 |
| 2K | 5 | 77 | 30 | 15 | 60 | 37 | 24 |
| 4K | 5 | 86 | 30 | 15 | 65 | 38 | 25 |
| 8K | 5 | 89 | 33 | 15 | 68 | 41 | 26 |
| 16K | 5 | 95 | 46 | 20 | 75 | 44 | 27 |
| 32K | 3 | 100 | 50 | 20 | 75 | 45 | 27 |
| 64K | 0 | 100 | 48 | 20 | 75 | 46 | 30 |
| 144K | 0 | 100 | 40 | 20 | 74 | 44 | 29 |

Incremental Fractional Utilization #4 (All)

| Trace Length | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 1K | 0 | 77 | 20 | 10 | 29 | 23 | 20 |
| 2K | 1 | 77 | 20 | 10 | 29 | 23 | 20 |
| 4K | 1 | 86 | 20 | 10 | 40 | 29 | 22 |
| 8K | 1 | 100 | 25 | 14 | 57 | 36 | 26 |
| 16K | 1 | 100 | 30 | 14 | 74 | 41 | 29 |
| 32K | 1 | 100 | 50 | 17 | 75 | 47 | 30 |
| 64K | 1 | 100 | 54 | 20 | 75 | 49 | 30 |
| 144K | 1 | 100 | 59 | 20 | 75 | 51 | 30 |

Incremental Fractional Utilization #5 (All)

| Trace Length | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 1K | 5 | 77 | 37 | 14 | 57 | 35 | 23 |
| 2K | 5 | 77 | 37 | 14 | 57 | 35 | 23 |
| 4K | 5 | 85 | 46 | 14 | 60 | 42 | 25 |
| 8K | 5 | 85 | 46 | 14 | 60 | 42 | 25 |
| 16K | 5 | 89 | 50 | 15 | 67 | 46 | 25 |
| 32K | 2 | 89 | 54 | 33 | 65 | 47 | 24 |
| 64K | 2 | 92 | 54 | 36 | 67 | 50 | 23 |
| 144K | 3 | 92 | 54 | 25 | 68 | 48 | 24 |

Incremental Fractional Utilization #1 (Code)

| Trace Length | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 1K | 3 | 85 | 45 | 14 | 59 | 39 | 25 |
| 2K | 3 | 85 | 45 | 14 | 60 | 42 | 24 |
| 4K | 3 | 85 | 50 | 14 | 65 | 43 | 25 |
| 8K | 3 | 85 | 50 | 14 | 65 | 45 | 25 |
| 16K | 3 | 88 | 50 | 14 | 65 | 46 | 25 |
| 32K | 3 | 89 | 51 | 14 | 68 | 47 | 25 |
| 64K | 3 | 89 | 50 | 15 | 68 | 47 | 25 |
| 144K | 3 | 93 | 51 | 15 | 68 | 48 | 25 |

Incremental Fractional Utilization #2 (Code)

| Trace Length | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 1K | 5 | 71 | 39 | 14 | 57 | 37 | 21 |
| 2K | 5 | 71 | 45 | 14 | 56 | 37 | 21 |
| 4K | 5 | 86 | 50 | 14 | 64 | 44 | 24 |
| 8K | 3 | 91 | 54 | 14 | 68 | 48 | 26 |
| 16K | 3 | 91 | 52 | 19 | 65 | 47 | 26 |
| 32K | 3 | 91 | 54 | 25 | 70 | 49 | 26 |
| 64K | 3 | 91 | 54 | 39 | 73 | 51 | 25 |
| 144K | 3 | 91 | 57 | 40 | 73 | 53 | 25 |

Incremental Fractional Utilization #3 (Code)

| Trace Length | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 1K | 5 | 68 | 38 | 14 | 57 | 33 | 22 |
| 2K | 5 | 77 | 46 | 15 | 60 | 40 | 23 |
| 4K | 5 | 79 | 45 | 14 | 62 | 40 | 24 |
| 8K | 5 | 89 | 50 | 14 | 63 | 43 | 25 |
| 16K | 5 | 89 | 54 | 15 | 66 | 47 | 25 |
| 32K | 3 | 89 | 53 | 15 | 63 | 46 | 24 |
| 64K | 3 | 89 | 54 | 35 | 67 | 49 | 23 |
| 144K | 3 | 89 | 54 | 35 | 67 | 49 | 23 |

Incremental Fractional Utilization #4 (Code)

| Trace Length | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 1K | 11 | 77 | 44 | 13 | 56 | 39 | 22 |
| 2K | 11 | 77 | 44 | 13 | 56 | 39 | 22 |
| 4K | 5 | 86 | 40 | 14 | 63 | 41 | 25 |
| 8K | 3 | 88 | 40 | 14 | 59 | 40 | 25 |
| 16K | 3 | 88 | 40 | 14 | 59 | 40 | 25 |
| 32K | 3 | 89 | 43 | 14 | 60 | 41 | 26 |
| 64K | 3 | 94 | 45 | 14 | 63 | 42 | 26 |
| 144K | 3 | 94 | 45 | 14 | 60 | 42 | 26 |

Incremental Fractional Utilization #5 (Code)

| Trace Length | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 1K | 5 | 75 | 20 | 13 | 50 | 32 | 25 |
| 2K | 5 | 100 | 25 | 13 | 75 | 36 | 27 |
| 4K | 5 | 100 | 25 | 20 | 75 | 39 | 27 |
| 8K | 5 | 100 | 25 | 20 | 75 | 39 | 27 |
| 16K | 5 | 100 | 25 | 20 | 75 | 40 | 28 |
| 32K | 0 | 100 | 30 | 20 | 75 | 41 | 29 |
| 64K | 0 | 100 | 30 | 20 | 75 | 42 | 31 |
| 144K | 0 | 100 | 33 | 20 | 71 | 43 | 30 |

Incremental Fractional Utilization #1 (Data)

| Trace Length | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 1K | 0 | 100 | 20 | 20 | 25 | 26 | 20 |
| 2K | 0 | 100 | 25 | 20 | 30 | 30 | 22 |
| 4K | 0 | 100 | 25 | 20 | 64 | 35 | 25 |
| 8K | 0 | 100 | 25 | 20 | 75 | 38 | 28 |
| 16K | 0 | 100 | 25 | 20 | 75 | 39 | 29 |
| 32K | 2 | 100 | 29 | 20 | 75 | 42 | 30 |
| 64K | 2 | 100 | 25 | 10 | 75 | 43 | 35 |
| 144K | 2 | 100 | 33 | 10 | 75 | 45 | 34 |

Incremental Fractional Utilization #2 (Data)

| Trace Length | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 1K | 0 | 80 | 20 | 11 | 30 | 26 | 19 |
| 2K | 2 | 80 | 25 | 20 | 33 | 30 | 21 |
| 4K | 2 | 86 | 20 | 11 | 40 | 31 | 24 |
| 8K | 4 | 100 | 25 | 11 | 67 | 35 | 28 |
| 16K | 0 | 100 | 25 | 10 | 80 | 40 | 33 |
| 32K | 0 | 100 | 25 | 10 | 80 | 41 | 34 |
| 64K | 0 | 100 | 28 | 10 | 80 | 41 | 33 |
| 144K | 0 | 100 | 25 | 5 | 67 | 35 | 32 |

Incremental Fractional Utilization #3 (Data)

| Trace Length | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 1K | 0 | 75 | 20 | 20 | 30 | 27 | 19 |
| 2K | 5 | 75 | 25 | 20 | 75 | 35 | 25 |
| 4K | 5 | 86 | 25 | 20 | 75 | 38 | 26 |
| 8K | 5 | 86 | 30 | 20 | 75 | 40 | 27 |
| 16K | 5 | 95 | 30 | 20 | 75 | 43 | 28 |
| 32K | 5 | 100 | 30 | 20 | 75 | 44 | 29 |
| 64K | 0 | 100 | 30 | 10 | 75 | 44 | 34 |
| 144K | 0 | 100 | 30 | 10 | 75 | 41 | 32 |

Incremental Fractional Utilization #4 (Data)

| Trace Length | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 1K | 0 | 67 | 10 | 8 | 20 | 15 | 14 |
| 2K | 1 | 67 | 10 | 8 | 20 | 16 | 14 |
| 4K | 1 | 83 | 20 | 10 | 25 | 22 | 17 |
| 8K | 1 | 100 | 25 | 10 | 50 | 33 | 27 |
| 16K | 1 | 100 | 25 | 20 | 75 | 41 | 31 |
| 32K | 1 | 100 | 67 | 20 | 75 | 51 | 31 |
| 64K | 1 | 100 | 67 | 20 | 75 | 53 | 31 |
| 144K | 1 | 100 | 75 | 25 | 75 | 56 | 31 |

Incremental Fractional Utilization #5 (Data)

### A-3. Access Frequency

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 1 | 26K | 16 | 5 | 59 | 209 | 1221 |
| All Code | 1 | 8K | 31 | 8 | 116 | 236 | 915 |
| All Data | 1 | 26K | 10 | 3 | 24 | 183 | 1454 |

Access Frequency #1

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 1 | 12K | 6 | 1 | 24 | 118 | 554 |
| All Code | 1 | 7K | 34 | 10 | 152 | 256 | 665 |
| All Data | 1 | 12K | 6 | 1 | 7 | 61 | 473 |

Access Frequency #2

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 1 | 6K | 16 | 4 | 70 | 167 | 538 |
| All Code | 1 | 6K | 74 | 18 | 246 | 355 | 758 |
| All Data | 1 | 4978 | 9 | 2 | 28 | 76 | 355 |

Access Frequency #3

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 1 | 14K | 14 | 5 | 64 | 203 | 839 |
| All Code | 1 | 9K | 24 | 8 | 125 | 250 | 832 |
| All Data | 1 | 14K | 10 | 3 | 25 | 150 | 842 |

Access Frequency #4

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 1 | 95K | 32 | 13 | 140 | 349 | 4706 |
| All Code | 1 | 3370 | 94 | 30 | 246 | 181 | 321 |
| All Data | 1 | 95K | 17 | 8 | 64 | 502 | 6491 |

Access Frequency #5

**A-4. Interreference Headway**

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 1 | 108K | 2 | 1 | 23 | 180 | 2643 |
| All Code | 1 | 74K | 1 | 1 | 18 | 134 | 1960 |
| All Data | 1 | 34K | 1 | 1 | 6 | 50 | 778 |

Interreference Headway #1

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 1 | 128K | 2 | 1 | 56 | 307 | 4234 |
| All Code | 1 | 86K | 1 | 1 | 41 | 233 | 3115 |
| All Data | 1 | 31K | 2 | 1 | 16 | 78 | 1149 |

Interreference Headway #2

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 1 | 97K | 7 | 1 | 51 | 239 | 1876 |
| All Code | 1 | 65K | 1 | 1 | 32 | 139 | 1055 |
| All Data | 1 | 30K | 8 | 1 | 21 | 100 | 767 |

Interreference Headway #3

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 1 | 93K | 1 | 1 | 60 | 118 | 1513 |
| All Code | 1 | 67K | 1 | 1 | 45 | 74 | 1095 |
| All Data | 1 | 25K | 2 | 1 | 18 | 44 | 520 |

Interreference Headway #4

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 1 | 108K | 1 | 1 | 1 | 176 | 1518 |
| All Code | 1 | 20K | 1 | 1 | 24 | 128 | 491 |
| All Data | 1 | 91K | 1 | 1 | 1 | 57 | 950 |

Interreference Headway #5

### A-5. Instance to Class Compression

| Trace | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Number of Classes Above 90% | 11 | 11 | 13 | 7 | 9 |
| Percent of All Fixed Fields Touched | 11 | 11 | 6 | 8 | 9 |

### A-6. Reference Counts

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 1 | 127 | 1 | 1 | 2 | 3 | 8 |
| All < 127 | 1 | 110 | 1 | 1 | 1 | 2 | 5 |
| 1 < All < 127 | 2 | 110 | 5 | 3 | 7 | 7 | 9 |

Static Reference Count

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 1 | 127 | 1 | 1 | 4 | 9 | 24 |
| All < 127 | 1 | 112 | 1 | 1 | 4 | 5 | 11 |
| 1 < All < 127 | 2 | 112 | 5 | 3 | 9 | 10 | 16 |

Quasi-Static Reference Count #1

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 1 | 127 | 1 | 1 | 8 | 18 | 38 |
| All < 127 | 1 | 112 | 1 | 1 | 6 | 9 | 21 |
| 1 < All < 127 | 2 | 112 | 7 | 4 | 19 | 22 | 30 |

Dynamic Reference Count #1

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 1 | 127 | 1 | 1 | 5 | 9 | 24 |
| All < 127 | 1 | 104 | 1 | 1 | 5 | 5 | 11 |
| 1 < All < 127 | 2 | 104 | 6 | 4 | 10 | 12 | 15 |

Quasi-Static Reference Count #2

| | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| All | 1 | 127 | 1 | 1 | 7 | 16 | 36 |
| All < 127 | 1 | 104 | 1 | 1 | 5 | 6 | 16 |
| 1 < All < 127 | 2 | 104 | 7 | 3 | 10 | 15 | 23 |

Dynamic Reference Count #2

**A-7. Selectors as a Percentage of Literals**

| Type | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| Static | 0 | 100 | 78 | 50 | 100 | 73 | 29 |
| Quasi-Static #1 | 0 | 100 | 75 | 58 | 100 | 73 | 28 |
| Dynamic #1 | 0 | 100 | 95 | 60 | 100 | 78 | 28 |
| Quasi-Static #2 | 0 | 100 | 80 | 50 | 100 | 75 | 27 |
| Dynamic #2 | 0 | 100 | 87 | 50 | 100 | 77 | 28 |

| | SELECTORS | NON - SELECTORS | FRACTION |
|---|---|---|---|
| Static | 10,737 | 5357 | 66.7% |
| Quasi-Static #1 | 470 | 218 | 68.3% |
| Dynamic #1 | 8077 | 2048 | 79.8% |
| Quasi-Static #2 | 523 | 202 | 72.1% |
| Dynamic #2 | 7454 | 2976 | 71.5% |

ByteCodes executed:
96,256 in Trace #1
95,901 in Trace #2

# Appendix B

## B-0. Abbreviations

| | |
|---|---|
| DFS | -- depth-first search |
| BFS | -- breadth-first search |
| ID | -- using identity permutations |
| Dyn | -- permutations arising from dynamic information |
| Refct | -- permutations arising from reference counts |
| Ooze | -- based on the grouping for the Object-Oriented Zoned Environment virtual memory |
| Hash | -- a random grouping based on permuting the bits in the unique identifier of the object |
| OPTi | -- the optimal grouping based on the *ith* compressed trace |

## B-1. Static Pointer Distance*

| | CLOSE** | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|---|
| BFSID | 0.19% | 0 | 1538 | 386 | 155 | 854 | 508 | 409 |
| BFSRefct | 0.14% | 0 | 1535 | 408 | 149 | 879 | 520 | 424 |
| BFSDyn | 0.20% | 0 | 1539 | 386 | 155 | 854 | 507 | 409 |
| DFSID | 14.8% | 0 | 1541 | 296 | 7 | 864 | 478 | 506 |
| DFSRefct | 14.8% | 0 | 1537 | 298 | 6 | 825 | 472 | 500 |
| DFSDyn | 15.0% | 0 | 1536 | 311 | 7 | 860 | 476 | 501 |
| Ooze | 3.32% | 0 | 1559 | 569 | 128 | 1199 | 630 | 513 |
| Hash | 0.35% | 0 | 1576 | 353 | 85 | 840 | 488 | 443 |
| OPT1 | 13.1% | 0 | 244 | 69 | 17 | 134 | 80 | 67 |
| OPT2 | 12.4% | 1 | 475 | 191 | 36 | 340 | 192 | 155 |
| OPT3 | 10.0% | 1 | 352 | 81 | 35 | 167 | 103 | 84 |

For the realizable groupings:

| | |
|---|---|
| Number of Objects: | 17,268 |
| Space requirements: | 404,384 (1580 pages) |
| Total Pointers: | 88,729 |
| Immediate Pointers: | 31,027 |
| Non-Immediate Pointers: | 57,702 |

*All distances were rounded up to the nearest 256-word page. SmallIntegers and "nil" were defined to be immediate pointers. Only non-immediate pointers were considered in this analysis.

**CLOSE is the percentage of non-immediate pointers which are within one page of their referent.

**B-2. Neighbor Relation**

| | BFSID | BFSRefct | BFSDyn | DFSID | DFSRefct | DFSDyn | Ooze | Hash |
|---|---|---|---|---|---|---|---|---|
| BFSID | * | 46.3% | 73.1% | 22.4% | 23.1% | 22.5% | 18.9% | 0.5% |
| BFSRefct | 46.7% | * | 45.5% | 23.3% | 23.0% | 22.8% | 19.0% | 0.6% |
| BFSDyn | 73.5% | 45.4% | * | 21.9% | 22.7% | 22.5% | 19.3% | 0.6% |
| DFSID | 20.8% | 21.5% | 20.2% | * | 60.8% | 59.4% | 22.7% | 0.2% |
| DFSRefct | 21.5% | 21.1% | 20.9% | 60.8% | * | 58.9% | 21.2% | 0.3% |
| DFSDyn | 21.1% | 21.1% | 21.0% | 59.8% | 59.4% | * | 21.6% | 0.3% |
| Ooze | 20.3% | 20.2% | 20.7% | 26.3% | 24.5% | 24.8% | * | 2.2% |
| Hash | 0.5% | 0.6% | 0.5% | 0.2% | 0.3% | 0.3% | 1.9% | * |
| OPT1 | 4.6% | 3.5% | 4.6% | 9.1% | 8.6% | 8.9% | 8.1% | 0.2% |
| OPT2 | 14.5% | 13.4% | 14.4% | 21.2% | 20.1% | 20.3% | 36.1% | 1.4% |
| OPT3 | 6.3% | 3.6% | 6.6% | 9.3% | 8.4% | 8.9% | 10.5% | 0.4% |
| | | | | | | | | |
| BFSID | * | 46.2% | 75.0% | 21.7% | 22.7% | 21.9% | 17.7% | 1.0% |
| BFSRefct | 46.3% | * | 47.2% | 23.7% | 22.4% | 22.3% | 18.1% | 1.0% |
| BFSDyn | 75.1% | 47.2% | * | 22.0% | 22.5% | 21.9% | 18.4% | 1.0% |
| DFSID | 19.5% | 21.2% | 19.7% | * | 59.9% | 61.2% | 21.5% | 0.7% |
| DFSRefct | 20.5% | 20.3% | 20.4% | 60.4% | * | 59.1% | 20.2% | 0.7% |
| DFSDyn | 19.9% | 20.2% | 19.9% | 62.3% | 59.2% | * | 20.0% | 0.7% |
| Ooze | 19.2% | 19.5% | 19.9% | 25.9% | 24.2% | 23.8% | * | 2.9% |
| Hash | 0.9% | 0.9% | 0.9% | 0.7% | 0.7% | 0.7% | 2.4% | * |
| OPT1 | 4.4% | 3.8% | 4.6% | 8.3% | 8.0% | 8.0% | 8.3% | 0.3% |
| OPT2 | 11.6% | 11.5% | 11.8% | 18.5% | 17.3% | 17.6% | 40.1% | 1.7% |
| OPT3 | 6.9% | 4.3% | 6.9% | 7.9% | 7.3% | 7.3% | 9.6% | 0.4% |
| | | | | | | | | |
| BFSID | * | 46.9% | 80.4% | 21.3% | 21.4% | 19.8% | 16.7% | 1.3% |
| BFSRefct | 47.3% | * | 47.6% | 21.3% | 21.3% | 21.1% | 16.0% | 1.3% |
| BFSDyn | 80.7% | 47.3% | * | 20.9% | 20.8% | 20.3% | 16.4% | 1.3% |
| DFSID | 19.0% | 18.8% | 18.6% | * | 58.4% | 52.7% | 19.9% | 1.4% |
| DFSRefct | 19.1% | 18.9% | 18.6% | 58.5% | * | 59.7% | 18.0% | 1.4% |
| DFSDyn | 17.7% | 18.7% | 18.1% | 53.1% | 60.0% | * | 17.9% | 1.4% |
| Ooze | 18.1% | 17.5% | 18.0% | 24.6% | 22.3% | 22.0% | * | 3.3% |
| Hash | 1.1% | 1.1% | 1.1% | 1.4% | 1.3% | 1.3% | 2.6% | * |
| OPT1 | 4.1% | 4.0% | 4.3% | 7.6% | 7.9% | 7.3% | 8.3% | 0.4% |
| OPT2 | 9.3% | 9.3% | 9.4% | 15.4% | 13.9% | 14.6% | 37.2% | 1.8% |
| OPT3 | 6.9% | 4.4% | 6.8% | 6.9% | 6.6% | 6.6% | 9.3% | 0.6% |

Retained fraction of unordered pairs with page sizes of 128, 256, and 512 words

| (Page Size) | OPT1 128 | OPT2 128 | OPT3 128 | OPT1 256 | OPT2 256 | OPT3 256 | OPT1 512 | OPT2 512 | OPT3 512 |
|---|---|---|---|---|---|---|---|---|---|
| OPT1 | * | 5.66% | 8.40% | * | 5.06% | 7.41% | * | 4.83% | 6.52% |
| OPT2 | 14.0% | * | 7.50% | 13.8% | * | 6.17% | 14.2% | * | 5.66% |
| OPT3 | 17.4% | 6.28% | * | 16.8% | 5.23% | * | 15.0% | 4.56% | * |

Retained fraction of unordered pairs between optimal initial placements

| Page Size | BFSID | BFSRefct | BFSDyn | DFSID | DFSRefct | DFSDyn | Ooze | Hash |
|---|---|---|---|---|---|---|---|---|
| 128 | 72,970 | 73,582 | 73,360 | 67,795 | 67,765 | 68,326 | 78,429 | 67,225 |
| 256 | 145,359 | 145,799 | 145,651 | 130,596 | 131,809 | 132,153 | 157,686 | 129,983 |
| 512 | 283,048 | 285,659 | 283,840 | 252,167 | 252,751 | 253,868 | 311,895 | 243,030 |

| Page Size | OPT1 | OPT2 | OPT3 |
|---|---|---|---|
| 128 | 7386 | 19,513 | 15,580 |
| 256 | 14,304 | 38,547 | 30,754 |
| 512 | 27,123 | 74,612 | 59,878 |

Number of unordered pairs in the specified neighbor relation

**B-3. Neighbor Relation under Continuous Displacement**

| OFFSET* | DFSID | HASH |
|---|---|---|
| 0 | 100% | 100% |
| 4 | 94.2% | 94.1% |
| 8 | 88.4% | 88.1% |
| 12 | 83.3% | 82.4% |
| 16 | 78.3% | 77.4% |
| 20 | 73.6% | 72.7% |
| 24 | 69.5% | 68.8% |
| 28 | 65.8% | 64.9% |
| 32 | 62.6% | 61.7% |
| 36 | 59.6% | 58.7% |
| 40 | 56.8% | 56.3% |
| 44 | 54.8% | 54.2% |
| 48 | 53.0% | 52.4% |
| 52 | 51.6% | 51.0% |
| 56 | 50.5% | 50.0% |
| 60 | 49.9% | 49.4% |
| 64 | 49.6% | 49.2% |
| 68 | 49.8% | 49.4% |
| 72 | 50.3% | 50.0% |
| 76 | 51.3% | 51.0% |
| 80 | 52.5% | 52.4% |
| 84 | 54.3% | 54.1% |
| 88 | 56.4% | 56.2% |
| 92 | 58.9% | 58.8% |
| 96 | 61.8% | 61.6% |
| 100 | 65.1% | 64.9% |
| 104 | 68.9% | 68.4% |
| 108 | 73.2% | 72.7% |
| 112 | 77.8% | 77.3% |
| 116 | 82.6% | 82.2% |
| 120 | 88.0% | 87.8% |
| 124 | 93.6% | 93.6% |

*In 16-bit words with a page size of 128 words

# Appendix C

## C-0. Abbreviations and Trace Data

DFS -- depth-first search
BFS -- breadth-first search
ID -- using identity permutations
Dyn -- permutations arising from dynamic information
Refct -- permutations arising from reference counts
Ooze -- based on the grouping for the Object-Oriented Zoned Environment virtual memory
Hash -- a random grouping based on permuting the bits in the unique identifier of the object
OPTi -- the optimal grouping based on the *ith* compressed trace

| TRACE NUMBER | COMPRESSED REFERENCES | BYTECODES (ESTIMATE) | TOTAL REFERENCES (ESTIMATE) |
|---|---|---|---|
| #1 | 41,580 | 1110K | 3885K |
| #2 | 43,049 | 960K | 3360K |
| #3 | 80,443 | 2280K | 7980K |

| TRACE NUMBER | STATISTIC TYPE | NON-POINTER OBJECTS | COMPILED METHODS | POINTER OBJECTS |
|---|---|---|---|---|
| * | static | 47.9% | 19.6% | 32.5% |
| #1 | quasi-static | 23.1% | 33.9% | 43.0% |
| #2 | quasi-static | 12.3% | 16.4% | 71.3% |
| #3 | quasi-static | 35.3% | 20.9% | 43.8% |
| #1 | dynamic | 3.9% | 34.7% | 61.4% |
| #2 | dynamic | 3.7% | 28.2% | 68.1% |
| #3 | dynamic | 6.3% | 29.3% | 64.4% |

| TRACE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| #1 | 0 | 41K | 1 | 0 | 4 | 2938 | 9369 |
| #2 | 0 | 43K | 1 | 0 | 2579 | 4170 | 10,127 |
| #3 | 0 | 79K | 1 | 1 | 14 | 7865 | 18,676 |

Thin Spread

| TRACE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| #1 | 1 | 4887 | 1 | 1 | 1 | 350 | 1109 |
| #2 | 1 | 6145 | 1 | 1 | 191 | 469 | 1203 |
| #3 | 1 | 13,313 | 1 | 1 | 2 | 1391 | 3221 |

Fat Spread (rounded up to next 256-word page)

## C-1. Paged Virtual Memory

| | 20K | 30K | 40K | 50K | 60K | 70K | 80K | 90K |
|---|---|---|---|---|---|---|---|---|
| BFSID | 10,489 | 8574 | 7100 | 5213 | 3734 | 2702 | 2101 | 1747 |
| BFSRefct | 10,537 | 8702 | 7278 | 5404 | 3894 | 2740 | 2144 | 1782 |
| BFSDyn | 10,519 | 8644 | 7118 | 5257 | 3756 | 2721 | 2132 | 1776 |
| DFSID | 8367 | 6780 | 5101 | 3629 | 2519 | 1946 | 1542 | 1261 |
| DFSRefct | 8208 | 6794 | 5131 | 3749 | 2584 | 1997 | 1591 | 1299 |
| DFSDyn | 8308 | 6774 | 5113 | 3752 | 2610 | 2003 | 1582 | 1293 |
| Ooze | 9805 | 7954 | 6224 | 4458 | 2982 | 2209 | 1657 | 1367 |
| OPT1 | 2812 | 1506 | 935 | 694 | 572 | 534 | 492 | 459 |
| Hash | 14,690 | 13,379 | 12,300 | 11,371 | 10,305 | 8363 | 6439 | 4518 |

| | 100K | 110K | 120K | 130K | 140K | 150K | 160K | 170K |
|---|---|---|---|---|---|---|---|---|
| BFSID | 1461 | 1258 | 1195 | 1148 | 1102 | 1066 | 1022 | 998 |
| BFSRefct | 1503 | 1289 | 1211 | 1170 | 1102 | 1078 | 1040 | 1010 |
| BFSDyn | 1494 | 1278 | 1216 | 1166 | 1114 | 1084 | 1040 | 998 |
| DFSID | 1106 | 1039 | 976 | 945 | 903 | 868 | 853 | 811 |
| DFSRefct | 1136 | 1054 | 995 | 960 | 922 | 891 | 873 | 836 |
| DFSDyn | 1146 | 1069 | 1009 | 974 | 934 | 896 | 879 | 854 |
| Ooze | 1152 | 1068 | 1009 | 966 | 917 | 884 | 865 | 831 |
| OPT1 | 437 | 428 | * | * | * | * | * | * |
| Hash | 3669 | 3012 | 2579 | 2336 | 1974 | 1772 | 1625 | 1550 |

| | 180K | 190K | 200K | 210K | 220K | 230K | 240K | 250K |
|---|---|---|---|---|---|---|---|---|
| BFSID | 967 | 931 | 915 | 875 | * | * | * | * |
| BFSRefct | 987 | 968 | 931 | 893 | 887 | * | * | * |
| BFSDyn | 975 | 940 | 921 | 881 | * | * | * | * |
| DFSID | 796 | 770 | * | * | * | * | * | * |
| DFSRefct | 821 | 788 | * | * | * | * | * | * |
| DFSDyn | 820 | 789 | * | * | * | * | * | * |
| Ooze | 817 | 796 | * | * | * | * | * | * |
| OPT1 | * | * | * | * | * | * | * | * |
| Hash | 1501 | 1438 | 1378 | 1333 | 1289 | 1256 | 1232 | 1202 |

| | 260K | 270K | 280K | 290K | 300K | 310K | 320K | 330K |
|---|---|---|---|---|---|---|---|---|
| Hash | 1178 | 1154 | 1134 | * | * | * | * | * |

Page faults as a function of core size (trace # 1)

| | 20K | 30K | 40K | 50K | 60K | 70K | 80K | 90K |
|---|---|---|---|---|---|---|---|---|
| BFSID | 12,407 | 10,203 | 8503 | 6653 | 5137 | 3857 | 3391 | 3135 |
| BFSRefct | 12,478 | 10,381 | 8690 | 6791 | 5194 | 3857 | 3409 | 3154 |
| BFSDyn | 12,427 | 10,257 | 8560 | 6668 | 5220 | 3842 | 3378 | 3164 |
| DFSID | 9877 | 8059 | 6263 | 4952 | 3648 | 3254 | 3083 | 2907 |
| DFSRefct | 9984 | 8256 | 6457 | 5185 | 3770 | 3325 | 3153 | 2979 |
| DFSDyn | 9920 | 8073 | 6257 | 4933 | 3716 | 3338 | 3166 | 2988 |
| Ooze | 10,960 | 8755 | 6780 | 5189 | 3651 | 2672 | 2175 | 1979 |
| OPT2 | 4130 | 2508 | 1677 | 1448 | 1307 | 1189 | 896 | 835 |
| Hash | 17,830 | 16,425 | 14,940 | 13,399 | 12,034 | 9962 | 8515 | 6276 |

| | 100K | 110K | 120K | 130K | 140K | 150K | 160K | 170K |
|---|---|---|---|---|---|---|---|---|
| BFSID | 2928 | 2650 | 2440 | 2320 | 2182 | 1760 | 1701 | 1623 |
| BFSRefct | 2931 | 2677 | 2461 | 2355 | 2236 | 1779 | 1700 | 1630 |
| BFSDyn | 2945 | 2671 | 2472 | 2374 | 2231 | 1783 | 1715 | 1635 |
| DFSID | 2732 | 2602 | 2494 | 2414 | 2340 | 1793 | 1725 | 1684 |
| DFSRefct | 2789 | 2643 | 2509 | 2433 | 2356 | 2277 | 1723 | 1680 |
| DFSDyn | 2802 | 2663 | 2550 | 2453 | 2393 | 2314 | 1750 | 1702 |
| Ooze | 1749 | 1581 | 1436 | 1372 | 1339 | 1289 | 1260 | 1219 |
| OPT2 | 773 | 737 | 716 | 703 | 680 | 662 | 658 | 656 |
| Hash | 5162 | 4666 | 4374 | 4126 | 3827 | 3566 | 3247 | 3000 |

| | 180K | 190K | 200K | 210K | 220K | 230K | 240K | 250K |
|---|---|---|---|---|---|---|---|---|
| BFSID | 1584 | 1515 | 1463 | 1427 | 1411 | 1389 | 1374 | 1359 |
| BFSRefct | 1577 | 1528 | 1453 | 1425 | 1406 | 1382 | 1362 | 1344 |
| BFSDyn | 1593 | 1532 | 1472 | 1434 | 1416 | 1393 | 1373 | 1356 |
| DFSID | 1661 | 1618 | 1576 | 1523 | 1462 | 1430 | 1410 | 1387 |
| DFSRefct | 1656 | 1626 | 1574 | 1539 | 1466 | 1425 | 1408 | 1379 |
| DFSDyn | 1666 | 1641 | 1592 | 1556 | 1486 | 1427 | 1404 | 1383 |
| Ooze | 1200 | 1178 | 1156 | 1123 | 1091 | 1055 | 1007 | 1005 |
| OPT2 | * | * | * | * | * | * | * | * |
| Hash | 2695 | 2252 | 2130 | 2027 | 1928 | 1849 | 1767 | 1709 |

| | 260K | 270K | 280K | 290K | 300K | 310K | 320K | 330K |
|---|---|---|---|---|---|---|---|---|
| BFSID | 1323 | 1277 | 1248 | 1241 | 1221 | * | * | * |
| BFSRefct | 1320 | 1278 | 1249 | 1243 | 1220 | * | * | * |
| BFSDyn | 1315 | 1277 | 1250 | 1246 | 1224 | * | * | * |
| DFSID | 1369 | 1341 | 1303 | 1274 | 1244 | * | * | * |
| DFSRefct | 1352 | 1325 | 1286 | 1243 | 1233 | * | * | * |
| DFSDyn | 1361 | 1335 | 1293 | 1271 | 1245 | 1243 | * | * |
| Ooze | * | * | * | * | * | * | * | * |
| OPT2 | * | * | * | * | * | * | * | * |
| Hash | 1669 | 1627 | 1592 | 1567 | 1532 | 1492 | 1471 | 1409 |

| | 340K | 350K | 360K | 370K | 380K | 390K | 400K | 410K |
|---|---|---|---|---|---|---|---|---|
| Hash | 1396 | * | * | * | * | * | * | * |

Page faults as a function of core size (trace #2)

| | 20K | 30K | 40K | 50K | 60K | 70K | 80K | 90K |
|---|---|---|---|---|---|---|---|---|
| BFSID | 22,585 | 18,642 | 15,794 | 12,934 | 9157 | 6992 | 5565 | 4451 |
| BFSRefct | 22,874 | 18,967 | 15,978 | 13,192 | 9562 | 7228 | 5862 | 4617 |
| BFSDyn | 22,570 | 18,731 | 15,738 | 12,938 | 9142 | 6968 | 5565 | 4441 |
| DFSID | 17,917 | 14,955 | 12,501 | 9511 | 7238 | 5855 | 4801 | 4075 |
| DFSRefct | 18,040 | 15,253 | 12,935 | 9980 | 7525 | 6160 | 5096 | 4346 |
| DFSDyn | 18,100 | 15,218 | 12,802 | 9754 | 7538 | 6266 | 5224 | 4440 |
| Ooze | 20,759 | 16,613 | 13,390 | 10,291 | 6663 | 4774 | 3862 | 3313 |
| OPT3 | 8176 | 5105 | 3112 | 2168 | 1615 | 1389 | 1200 | 1063 |
| Hash | 31,763 | 28,776 | 26,412 | 24,407 | 22,491 | 20,466 | 17,012 | 12,881 |

| | 100K | 110K | 120K | 130K | 140K | 150K | 160K | 170K |
|---|---|---|---|---|---|---|---|---|
| BFSID | 3656 | 3138 | 2782 | 2568 | 2405 | 2228 | 2120 | 1987 |
| BFSRefct | 3796 | 3220 | 2853 | 2649 | 2471 | 2277 | 2164 | 2017 |
| BFSDyn | 3680 | 3190 | 2769 | 2564 | 2409 | 2250 | 2137 | 2009 |
| DFSID | 3462 | 2990 | 2773 | 2600 | 2450 | 2355 | 2229 | 2141 |
| DFSRefct | 3688 | 3079 | 2880 | 2647 | 2487 | 2370 | 2251 | 2148 |
| DFSDyn | 3889 | 3323 | 2952 | 2699 | 2527 | 2418 | 2295 | 2211 |
| Ooze | 2774 | 2452 | 2222 | 2091 | 1913 | 1839 | 1757 | 1639 |
| OPT3 | 974 | 864 | 790 | 726 | 696 | 671 | 659 | 656 |
| Hash | 10,855 | 9004 | 7564 | 6432 | 5442 | 4864 | 4241 | 3741 |

| | 180K | 190K | 200K | 210K | 220K | 230K | 240K | 250K |
|---|---|---|---|---|---|---|---|---|
| BFSID | 1904 | 1813 | 1721 | 1604 | 1454 | 1313 | 1227 | 1159 |
| BFSRefct | 1941 | 1853 | 1758 | 1638 | 1515 | 1339 | 1250 | 1185 |
| BFSDyn | 1923 | 1832 | 1751 | 1627 | 1480 | 1348 | 1261 | 1203 |
| DFSID | 2061 | 1967 | 1842 | 1718 | 1596 | 1518 | 1440 | 1374 |
| DFSRefct | 2061 | 1965 | 1842 | 1727 | 1597 | 1516 | 1448 | 1392 |
| DFSDyn | 2109 | 2011 | 1922 | 1809 | 1704 | 1586 | 1497 | 1437 |
| Ooze | 1525 | 1445 | 1301 | 1236 | 1162 | 1100 | 1078 | 1069 |
| OPT3 | * | * | * | * | * | * | * | * |
| Hash | 3454 | 3168 | 2959 | 2800 | 2636 | 2463 | 2327 | 2189 |

| | 260K | 270K | 280K | 290K | 300K | 310K | 320K | 330K |
|---|---|---|---|---|---|---|---|---|
| BFSID | 1141 | 1131 | 1126 | * | * | * | * | * |
| BFSRefct | 1152 | 1139 | 1133 | * | * | * | * | * |
| BFSDyn | 1164 | 1151 | 1142 | * | * | * | * | * |
| DFSID | 1328 | 1293 | 1252 | 1242 | 1232 | * | * | * |
| DFSRefct | 1332 | 1290 | 1256 | 1243 | 1234 | * | * | * |
| DFSDyn | 1360 | 1318 | 1275 | 1258 | 1249 | 1244 | * | * |
| Ooze | 1061 | * | * | * | * | * | * | * |
| OPT3 | * | * | * | * | * | * | * | * |
| Hash | 2069 | 1927 | 1802 | 1659 | 1547 | 1488 | 1440 | 1418 |

| | 340K | 350K | 360K | 370K | 380K | 390K | 400K | 410K |
|---|---|---|---|---|---|---|---|---|
| Hash | 1406 | 1401 | * | * | * | * | * | * |

Page faults as a function of core size (trace #3)

**C-2. Loom Simulation**

| CORE SIZE | OBJECT FAULT | LAMBDA FAULT | CLEAN CONTRACT | DIRTY CONTRACT | CLEAN-DIRTY RATIO |
|---|---|---|---|---|---|
| 20K | 6626 | 5669 | 4070 | 2112 | 1.93 |
| 40K | 4836 | 3466 | 1742 | 1575 | 1.11 |
| 60K | 4504 | 2975 | 594 | 1173 | 0.51 |
| 80K | 4432 | 2598 | 0 | 0 | * |

Trace #1 -- Data Invariant of Initial Placement

| | BUFFER MISS | BUFFER HIT | HIT RATE | BUFFER MISS | BUFFER HIT | HIT RATE |
|---|---|---|---|---|---|---|
| BFSID | 7724 | 7597 | 49.6% | 4890 | 5552 | 53.2% |
| BFSRefct | 7820 | 7515 | 49.1% | 4989 | 5452 | 52.2% |
| BFSDyn | 7668 | 7554 | 49.6% | 4926 | 5495 | 52.7% |
| DFSID | 6705 | 8632 | 56.3% | 4289 | 6158 | 58.9% |
| DFSRefct | 6711 | 8605 | 56.2% | 4285 | 6178 | 59.0% |
| DFSDyn | 6735 | 8574 | 56.0% | 4295 | 6131 | 58.8% |
| Ooze | 7405 | 7881 | 51.6% | 4687 | 5731 | 55.0% |
| OPT1 | 4691 | 10,714 | 69.5% | 2884 | 7640 | 72.6% |
| Hash | 8741 | 6528 | 42.8% | 5589 | 4817 | 46.3% |
| | Trace #1 -- Core Size 20K | | | Trace #1 -- Core Size 40K | | |

| | BUFFER MISS | BUFFER HIT | HIT RATE | BUFFER MISS | BUFFER HIT | HIT RATE |
|---|---|---|---|---|---|---|
| BFSID | 3991 | 5168 | 56.4% | 2780 | 4661 | 62.6% |
| BFSRefct | 4054 | 5112 | 55.8% | 2842 | 4602 | 61.8% |
| BFSDyn | 3983 | 5153 | 56.4% | 2766 | 4646 | 62.7% |
| DFSID | 3414 | 5748 | 62.7% | 2202 | 5245 | 70.4% |
| DFSRefct | 3444 | 5734 | 62.5% | 2212 | 5238 | 70.3% |
| DFSDyn | 3418 | 5722 | 62.6% | 2206 | 5216 | 70.3% |
| Ooze | 3757 | 5380 | 58.9% | 2540 | 4883 | 65.8% |
| OPT1 | 2185 | 7067 | 76.4% | 934 | 6599 | 87.6% |
| Hash | 4625 | 4516 | 49.4% | 3378 | 4045 | 54.5% |
| | Trace #1 --Core Size 60K | | | Trace #1 -- Core Size 80K | | |

| CORE SIZE | OBJECT FAULT | LAMBDA FAULT | CLEAN CONTRACT | DIRTY CONTRACT | CLEAN-DIRTY RATIO |
|---|---|---|---|---|---|
| 20K | 12,356 | 6587 | 9998 | 1724 | 5.80 |
| 40K | 8626 | 4520 | 5594 | 1357 | 4.12 |
| 60K | 8198 | 3999 | 4662 | 693 | 6.73 |
| 80K | 7582 | 3337 | 3402 | 667 | 5.10 |
| 100K | 7195 | 3152 | 391 | 628 | 0.62 |
| 120K | 7195 | 2859 | 0 | 0 | * |

Trace #2 -- Data Invariant of Initial Placement

Trace #2 -- Core Size 20K

| | BUFFER MISS | BUFFER HIT | HIT RATE |
|---|---|---|---|
| BFSID | 10,522 | 11,594 | 52.4% |
| BFSRefct | 10,538 | 11,579 | 52.4% |
| BFSDyn | 10,347 | 11,672 | 53.0% |
| DFSID | 8982 | 13,188 | 59.5% |
| DFSRefct | 8934 | 13,149 | 59.5% |
| DFSDyn | 8959 | 13,111 | 59.4% |
| Ooze | 8598 | 13,420 | 61.0% |
| OPT2 | 5326 | 16,892 | 76.0% |
| Hash | 14,096 | 7969 | 36.1% |

Trace #2 -- Core Size 40K

| | BUFFER MISS | BUFFER HIT | HIT RATE |
|---|---|---|---|
| BFSID | 7182 | 8296 | 53.6% |
| BFSRefct | 7196 | 8286 | 53.5% |
| BFSDyn | 7120 | 8302 | 53.8% |
| DFSID | 6184 | 9344 | 60.2% |
| DFSRefct | 6171 | 9297 | 60.1% |
| DFSDyn | 6177 | 9285 | 60.1% |
| Ooze | 5953 | 9481 | 61.4% |
| OPT2 | 3514 | 12,060 | 77.4% |
| Hash | 9451 | 6017 | 38.9% |

Trace #2 -- Core Size 60K

| | BUFFER MISS | BUFFER HIT | HIT RATE |
|---|---|---|---|
| BFSID | 5941 | 7745 | 56.6% |
| BFSRefct | 5911 | 7779 | 56.8% |
| BFSDyn | 5860 | 7790 | 57.1% |
| DFSID | 5045 | 8686 | 63.3% |
| DFSRefct | 5052 | 8634 | 63.1% |
| DFSDyn | 5057 | 8609 | 63.0% |
| Ooze | 4726 | 8925 | 65.4% |
| OPT2 | 2574 | 11,211 | 81.3% |
| Hash | 8138 | 5547 | 40.5% |

Trace #2 -- Core Size 80K

| | BUFFER MISS | BUFFER HIT | HIT RATE |
|---|---|---|---|
| BFSID | 5214 | 7100 | 57.7% |
| BFSRefct | 5196 | 7134 | 57.9% |
| BFSDyn | 5140 | 7148 | 58.2% |
| DFSID | 4424 | 7927 | 64.2% |
| DFSRefct | 4439 | 7883 | 64.0% |
| DFSDyn | 4444 | 7862 | 63.9% |
| Ooze | 4178 | 8113 | 66.0% |
| OPT2 | 2275 | 10,145 | 81.7% |
| Hash | 7241 | 5082 | 41.2% |

Trace #2 -- Core Size100K

| | BUFFER MISS | BUFFER HIT | HIT RATE |
|---|---|---|---|
| BFSID | 4891 | 6763 | 58.0% |
| BFSRefct | 4873 | 6802 | 58.3% |
| BFSDyn | 4831 | 6813 | 58.5% |
| DFSID | 4077 | 7608 | 65.1% |
| DFSRefct | 4111 | 7557 | 64.8% |
| DFSDyn | 4112 | 7529 | 64.7% |
| Ooze | 3822 | 7807 | 67.1% |
| OPT2 | 2032 | 9724 | 82.7% |
| Hash | 6826 | 4831 | 41.4% |

Trace #2 -- Core Size 120K

| | BUFFER MISS | BUFFER HIT | HIT RATE |
|---|---|---|---|
| BFSID | 4113 | 6565 | 61.5% |
| BFSRefct | 4092 | 6612 | 61.8% |
| BFSDyn | 4038 | 6623 | 62.1% |
| DFSID | 3330 | 7400 | 69.0% |
| DFSRefct | 3327 | 7366 | 68.9% |
| DFSDyn | 3340 | 7334 | 68.7% |
| Ooze | 3022 | 7632 | 71.6% |
| OPT2 | 1302 | 9480 | 87.9% |
| Hash | 6030 | 4653 | 43.6% |

| CORE SIZE | OBJECT FAULT | LAMBDA FAULT | CLEAN CONTRACT | DIRTY CONTRACT | CLEAN-DIRTY RATIO |
|---|---|---|---|---|---|
| 20K | 16,273 | 15,012 | 13,115 | 3056 | 4.29 |
| 40K | 10,471 | 8513 | 6782 | 2593 | 2.62 |
| 60K | 8455 | 5759 | 3841 | 2401 | 1.60 |
| 80K | 7378 | 5003 | 1317 | 1958 | 0.67 |
| 100K | 7012 | 4638 | 406 | 1472 | 0.28 |
| 120K | 6957 | 4315 | 0 | 0 | * |

Trace #3 -- Data Invariant of Initial Placement

| | BUFFER MISS | BUFFER HIT | HIT RATE |
|---|---|---|---|
| BFSID | 19,011 | 17,160 | 47.4% |
| BFSRefct | 19,697 | 16,465 | 45.5% |
| BFSDyn | 18,980 | 17,188 | 47.5% |
| DFSID | 16,045 | 20,215 | 55.8% |
| DFSRefct | 16,033 | 20,254 | 55.8% |
| DFSDyn | 16,288 | 20,054 | 55.2% |
| Ooze | 18,467 | 17,719 | 49.0% |
| OPT3 | 11,397 | 24,985 | 68.7% |
| Hash | 22,434 | 13,822 | 38.1% |

Trace #3 -- Core Size 20K

| | BUFFER MISS | BUFFER HIT | HIT RATE |
|---|---|---|---|
| BFSID | 11,512 | 11,263 | 49.5% |
| BFSRefct | 11,985 | 10,812 | 47.4% |
| BFSDyn | 11,492 | 11,280 | 49.5% |
| DFSID | 9685 | 13,115 | 57.5% |
| DFSRefct | 9728 | 13,110 | 57.4% |
| DFSDyn | 9801 | 13,007 | 57.0% |
| Ooze | 10,979 | 11,800 | 51.8% |
| OPT3 | 6669 | 16,268 | 70.9% |
| Hash | 13,471 | 9340 | 41.0% |

Trace #3 -- Core Size 40K

| | BUFFER MISS | BUFFER HIT | HIT RATE |
|---|---|---|---|
| BFSID | 8839 | 8768 | 49.8% |
| BFSRefct | 9216 | 8426 | 47.8% |
| BFSDyn | 8831 | 8770 | 49.8% |
| DFSID | 7571 | 10,036 | 57.0% |
| DFSRefct | 7595 | 10,057 | 57.0% |
| DFSDyn | 7657 | 9956 | 56.5% |
| Ooze | 8321 | 9267 | 52.7% |
| OPT3 | 4994 | 12,762 | 71.9% |
| Hash | 10,214 | 7407 | 42.0% |

Trace #3 -- Core Size 60K

| | BUFFER MISS | BUFFER HIT | HIT RATE |
|---|---|---|---|
| BFSID | 7135 | 8057 | 53.0% |
| BFSRefct | 7534 | 7695 | 50.5% |
| BFSDyn | 7175 | 8031 | 52.8% |
| DFSID | 6232 | 8956 | 59.0% |
| DFSRefct | 6285 | 8950 | 58.7% |
| DFSDyn | 6336 | 8867 | 58.3% |
| Ooze | 6680 | 8509 | 56.0% |
| OPT3 | 3792 | 11,560 | 75.3% |
| Hash | 8386 | 6820 | 44.9% |

Trace #3 -- Core Size 80K

| | BUFFER MISS | BUFFER HIT | HIT RATE |
|---|---|---|---|
| BFSID | 6236 | 7656 | 55.1% |
| BFSRefct | 6649 | 7293 | 52.3% |
| BFSDyn | 6269 | 7631 | 54.9% |
| DFSID | 5415 | 8476 | 61.0% |
| DFSRefct | 5446 | 8483 | 60.9% |
| DFSDyn | 5514 | 8404 | 60.4% |
| Ooze | 5837 | 8051 | 58.0% |
| OPT3 | 3093 | 10,949 | 78.0% |
| Hash | 7483 | 6410 | 46.1% |

Trace #3 -- Core Size100K

| | BUFFER MISS | BUFFER HIT | HIT RATE |
|---|---|---|---|
| BFSID | 4844 | 7050 | 59.3% |
| BFSRefct | 5226 | 6704 | 56.2% |
| BFSDyn | 4834 | 7061 | 59.4% |
| DFSID | 3989 | 7909 | 66.5% |
| DFSRefct | 4032 | 7899 | 66.2% |
| DFSDyn | 4067 | 7850 | 65.9% |
| Ooze | 4436 | 7460 | 62.7% |
| OPT3 | 1596 | 10,438 | 86.7% |
| Hash | 6027 | 5863 | 49.3% |

Trace #3 -- Core Size 120K

## C-3. Selected Page Utilization and Cleanliness Rates

| CORE SIZE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 20K | 0 | 100 | 10 | 4 | 30 | 22 | 26 |
| 40K | 0 | 100 | 9 | 4 | 29 | 22 | 27 |
| 60K | 0 | 100 | 11 | 4 | 32 | 24 | 28 |
| 80K | 0 | 100 | 16 | 5 | 45 | 30 | 32 |

Trace #1 -- Page usage for DFSID

| CORE SIZE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 20K | 0 | 100 | 0 | 0 | 0 | 3 | 12 |
| 40K | 0 | 100 | 0 | 0 | 2 | 3 | 9 |
| 60K | 0 | 100 | 0 | 0 | 0 | 3 | 9 |
| 80K | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Trace #1 -- Page dirt for DFSID

| CORE SIZE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 20K | 0 | 100 | 14 | 5 | 33 | 24 | 26 |
| 40K | 0 | 100 | 13 | 5 | 33 | 24 | 27 |
| 60K | 0 | 100 | 14 | 5 | 34 | 24 | 26 |
| 80K | 0 | 100 | 14 | 5 | 34 | 25 | 28 |
| 100K | 0 | 100 | 14 | 5 | 36 | 26 | 28 |
| 120K | 0 | 100 | 17 | 6 | 38 | 28 | 29 |

Trace #2 -- Page usage for DFSID

| CORE SIZE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 20K | 0 | 100 | 0 | 0 | 0 | 2 | 10 |
| 40K | 0 | 100 | 0 | 0 | 0 | 2 | 11 |
| 60K | 0 | 100 | 0 | 0 | 0 | 1 | 7 |
| 80K | 0 | 100 | 0 | 0 | 0 | 1 | 8 |
| 100K | 0 | 100 | 0 | 0 | 0 | 1 | 7 |
| 120K | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Trace #2 -- Page dirt for DFSID

| CORE SIZE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 20K | 0 | 100 | 11 | 4 | 28 | 21 | 25 |
| 40K | 0 | 100 | 10 | 4 | 27 | 22 | 26 |
| 60K | 0 | 100 | 9 | 4 | 27 | 20 | 26 |
| 80K | 0 | 100 | 9 | 4 | 27 | 21 | 26 |
| 100K | 0 | 100 | 9 | 4 | 27 | 22 | 27 |
| 120K | 0 | 100 | 12 | 5 | 31 | 25 | 30 |

Trace #3 -- Page usage for DFSID

| CORE SIZE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 20K | 0 | 100 | 0 | 0 | 0 | 2 | 10 |
| 40K | 0 | 100 | 0 | 0 | 0 | 3 | 11 |
| 60K | 0 | 100 | 0 | 0 | 0 | 3 | 12 |
| 80K | 0 | 100 | 0 | 0 | 0 | 3 | 12 |
| 100K | 0 | 100 | 0 | 0 | 0 | 3 | 11 |
| 120K | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Trace #3 -- Page dirt for DFSID

| CORE SIZE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 20K | 1 | 31 | 1 | 1 | 2 | 2 | 2 |
| 40K | 1 | 31 | 1 | 1 | 2 | 2 | 3 |
| 60K | 1 | 31 | 1 | 1 | 2 | 2 | 3 |
| 80K | 1 | 31 | 1 | 1 | 3 | 3 | 4 |

Trace #1 -- Page usage (in terms of objects) for DFSID

| TRACE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| BFSID | 0 | 100 | 9 | 4 | 27 | 20 | 25 |
| BFSRefct | 0 | 100 | 9 | 4 | 27 | 19 | 23 |
| BFSDyn | 0 | 100 | 8 | 4 | 27 | 20 | 25 |
| DFSID | 0 | 100 | 10 | 4 | 30 | 22 | 26 |
| DFSRefct | 0 | 100 | 11 | 4 | 32 | 22 | 25 |
| DFSDyn | 0 | 100 | 11 | 4 | 30 | 22 | 26 |
| Ooze | 0 | 100 | 9 | 5 | 27 | 21 | 26 |
| OPT1 | 0 | 100 | 17 | 5 | 49 | 31 | 31 |
| Hash | 0 | 100 | 7 | 4 | 24 | 18 | 23 |

Trace #1 -- Page usage for core = 20K

| TRACE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| BFSID | 0 | 100 | 9 | 5 | 32 | 25 | 30 |
| BFSRefct | 0 | 100 | 9 | 4 | 30 | 24 | 29 |
| BFSDyn | 0 | 100 | 10 | 5 | 33 | 25 | 30 |
| DFSID | 0 | 100 | 16 | 5 | 45 | 30 | 32 |
| DFSRefct | 0 | 100 | 16 | 5 | 46 | 30 | 31 |
| DFSDyn | 0 | 100 | 15 | 5 | 44 | 32 | 32 |
| Ooze | 0 | 100 | 12 | 5 | 37 | 27 | 31 |
| OPT1 | 0 | 100 | 88 | 27 | 100 | 65 | 38 |
| Hash | 0 | 100 | 8 | 4 | 27 | 21 | 28 |

Trace #1 -- Page usage for core = 80K

## C-4. Effect of Core Purging Policy

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|---|---|---|---|
| 20K | 7385 | 8166 | 52.5% | -10.14% | 9413 | 6090 | 39.3% | - 7.69% |
| 40K | 4665 | 5858 | 55.7% | - 8.77% | 5990 | 4495 | 42.9% | - 7.17% |
| 60K | 3454 | 5460 | 61.3% | - 1.17% | 4686 | 4207 | 47.3% | - 1.32% |
| 80K | 2202 | 5245 | 70.4% | 0.00% | 3378 | 4045 | 54.5% | 0.00% |
| | DFSID -- Trace #1 | | | | Hash -- Trace #1 | | | |

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|---|---|---|---|
| 20K | 9554 | 12,712 | 57.1% | - 6.37% | 14,679 | 7534 | 33.9% | - 4.14% |
| 40K | 6703 | 9428 | 58.4% | - 8.39% | 10,178 | 5897 | 36.7% | - 7.69% |
| 60K | 5343 | 8488 | 61.4% | - 5.91% | 8498 | 5295 | 38.4% | - 4.42% |
| 80K | 4432 | 7582 | 63.1% | - 0.18% | 7159 | 4815 | 40.2% | +1.13% |
| 100K | 3857 | 7521 | 66.1% | +5.40% | 6545 | 4790 | 42.3% | +4.12% |
| 120K | 3330 | 7400 | 69.0% | 0.00% | 6030 | 4653 | 43.6% | 0.00% |
| | DFSID -- Trace #2 | | | | Hash -- Trace #2 | | | |

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|---|---|---|---|
| 20K | 16,836 | 19,555 | 53.7% | - 4.93% | 23,172 | 13,228 | 36.3% | - 3.29% |
| 40K | 10,575 | 12,314 | 53.8% | - 9.19% | 14,272 | 8648 | 37.7% | - 5.95% |
| 60K | 8228 | 9369 | 53.2% | - 8.68% | 10,720 | 6895 | 39.1% | - 4.95% |
| 80K | 6810 | 8753 | 56.2% | - 9.23% | 9151 | 6426 | 41.3% | - 9.12% |
| 100K | 5139 | 8144 | 61.3% | +5.10% | 7224 | 6065 | 45.6% | +3.46% |
| 120K | 3989 | 7909 | 66.5% | 0.00% | 6027 | 5863 | 49.3% | 0.00% |
| | DFSID -- Trace #3 | | | | Hash -- Trace #3 | | | |

### C-5. Effect of Buffer Size

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|
| 20K | 8949 | 5654 | 38.7% | -33.5% |
| 40K | 5704 | 4438 | 43.8% | -33.0% |
| 60K | 4973 | 4114 | 45.3% | -45.7% |
| 80K | 3337 | 4110 | 55.2% | -51.5% |

Buffer Size = 2 (DFSID # 1)

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|
| 20K | 7722 | 7122 | 48.0% | -15.2% |
| 40K | 4859 | 5312 | 52.2% | -13.3% |
| 60K | 4047 | 5054 | 55.5% | -18.5% |
| 80K | 2659 | 4788 | 64.3% | -20.8% |

Buffer Size = 4 (DFSID # 1)

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|
| 20K | 7127 | 8013 | 52.9% | -6.3% |
| 40K | 4466 | 5801 | 56.5% | -4.1% |
| 60K | 3658 | 5458 | 59.9% | -7.1% |
| 80K | 2383 | 5064 | 68.0% | -8.2% |

Buffer Size = 6 (DFSID # 1)

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|
| 20K | 6630 | 9109 | 57.9% | 1.1% |
| 40K | 4067 | 6509 | 61.5% | 5.2% |
| 60K | 3238 | 5957 | 64.8% | 5.2% |
| 80K | 2083 | 5364 | 72.0% | 5.4% |

Buffer Size = 10 (DFSID # 1)

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|
| 20K | 6443 | 9810 | 60.4% | 3.9% |
| 40K | 3888 | 6734 | 63.4% | 9.3% |
| 60K | 3133 | 6091 | 66.0% | 8.2% |
| 80K | 1977 | 5470 | 73.5% | 10.2% |

Buffer Size = 12 (DFSID # 1)

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|
| 20K | 6378 | 10,636 | 62.5% | 4.9% |
| 40K | 3586 | 6986 | 66.1% | 16.4% |
| 60K | 2914 | 6264 | 68.3% | 14.6% |
| 80K | 1855 | 5592 | 75.1% | 15.8% |

Buffer Size = 16 (DFSID # 1)

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|
| 20K | 6452 | 11,563 | 64.2% | 3.8% |
| 40K | 3413 | 7238 | 68.0% | 20.4% |
| 60K | 2710 | 6453 | 70.4% | 20.6% |
| 80K | 1753 | 5694 | 76.5% | 20.4% |

Buffer Size = 20 (DFSID # 1)

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|
| 20K | 6661 | 12,377 | 65.0% | 0.7% |
| 40K | 3204 | 7370 | 69.7% | 25.3% |
| 60K | 2597 | 6556 | 71.6% | 23.9% |
| 80K | 1731 | 5780 | 77.0% | 21.4% |

Buffer Size = 24 (DFSID # 1)

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|
| 20K | 7004 | 13,350 | 65.6% | - 4.5% |
| 40K | 3221 | 7630 | 70.3% | 24.9% |
| 60K | 2470 | 6620 | 72.8% | 27.7% |
| 80K | 1750 | 5883 | 77.1% | 20.5% |

Buffer Size = 28 (DFSID # 1)

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|
| 20K | 7692 | 14,439 | 65.2% | -14.7% |
| 40K | 3117 | 7891 | 71.7% | 27.3% |
| 60K | 2389 | 6688 | 73.7% | 30.0% |
| 80K | 1774 | 5973 | 77.1% | 19.4% |

Buffer Size = 32 (DFSID # 1)

| BUFFER SIZE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 2 | 0 | 100 | 9 | 4 | 27 | 21 | 25 |
| 4 | 0 | 100 | 9 | 4 | 27 | 21 | 25 |
| 6 | 0 | 100 | 10 | 4 | 29 | 21 | 25 |
| 8 | 0 | 100 | 10 | 4 | 30 | 22 | 26 |
| 10 | 0 | 100 | 10 | 4 | 30 | 22 | 26 |
| 12 | 0 | 100 | 11 | 4 | 31 | 23 | 26 |
| 16 | 0 | 100 | 11 | 5 | 32 | 23 | 27 |
| 20 | 0 | 100 | 12 | 5 | 33 | 24 | 27 |
| 24 | 0 | 100 | 12 | 5 | 33 | 24 | 27 |
| 28 | 0 | 100 | 12 | 5 | 34 | 24 | 28 |
| 32 | 0 | 100 | 12 | 5 | 35 | 25 | 28 |

Trace #1 -- Page usage for DFSID (20K core)

| BUFFER SIZE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 2 | 0 | 100 | 14 | 5 | 43 | 29 | 31 |
| 4 | 0 | 100 | 14 | 5 | 41 | 28 | 31 |
| 6 | 0 | 100 | 15 | 5 | 43 | 29 | 31 |
| 8 | 0 | 100 | 16 | 5 | 45 | 30 | 32 |
| 10 | 0 | 100 | 16 | 5 | 47 | 31 | 32 |
| 12 | 0 | 100 | 16 | 5 | 50 | 31 | 33 |
| 16 | 0 | 100 | 16 | 6 | 52 | 32 | 34 |
| 20 | 0 | 100 | 17 | 6 | 52 | 33 | 34 |
| 24 | 0 | 100 | 18 | 6 | 52 | 33 | 35 |
| 28 | 0 | 100 | 17 | 6 | 52 | 33 | 34 |
| 32 | 0 | 100 | 16 | 6 | 52 | 33 | 34 |

Trace #1 -- Page usage for DFSID (80K core)

| BUFFER SIZE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 2 | 0 | 100 | 0 | 0 | 0 | 2 | 10 |
| 4 | 0 | 100 | 0 | 0 | 0 | 3 | 10 |
| 6 | 0 | 100 | 0 | 0 | 0 | 3 | 12 |
| 8 | 0 | 100 | 0 | 0 | 0 | 3 | 12 |
| 10 | 0 | 100 | 0 | 0 | 0 | 3 | 12 |
| 12 | 0 | 100 | 0 | 0 | 0 | 4 | 13 |
| 16 | 0 | 100 | 0 | 0 | 0 | 4 | 13 |
| 20 | 0 | 100 | 0 | 0 | 0 | 4 | 14 |
| 24 | 0 | 100 | 0 | 0 | 0 | 4 | 14 |
| 28 | 0 | 100 | 0 | 0 | 0 | 4 | 14 |
| 32 | 0 | 100 | 0 | 0 | 0 | 3 | 14 |

Trace #1 -- Page dirt for DFSID (20K core)

**C-6. Effect of Disk Buffer Purging Policy**

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|
| 20K | 6405 | 8932 | 58.2% | 4.48% |
| 40K | 4046 | 6401 | 61.3% | 5.67% |
| 60K | 3207 | 5955 | 65.0% | 6.06% |
| 80K | 2085 | 5362 | 72.0% | 5.31% |

Trace #1 -- DFSID

| BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|
| 8444 | 6825 | 44.7% | 3.40% |
| 5359 | 5047 | 48.5% | 4.12% |
| 4423 | 4718 | 51.6% | 4.37% |
| 3247 | 4176 | 56.3% | 3.88% |

Trace #1 -- Hash

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|
| 20K | 8645 | 13,525 | 61.0% | 3.75% |
| 40K | 5877 | 9651 | 62.2% | 4.96% |
| 60K | 4830 | 8901 | 64.8% | 4.26% |
| 80K | 4237 | 8114 | 65.7% | 4.23% |
| 100K | 3874 | 7811 | 66.8% | 4.98% |
| 120K | 3190 | 7540 | 70.3% | 4.20% |

Trace #2 -- DFSID

| BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|
| 13,743 | 8322 | 37.7% | 2.50% |
| 9123 | 6345 | 41.0% | 3.47% |
| 7881 | 5804 | 42.4% | 3.16% |
| 7019 | 5304 | 43.0% | 3.07% |
| 6555 | 5102 | 43.8% | 3.97% |
| 5832 | 4851 | 45.4% | 3.28% |

Trace #2 -- Hash

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|
| 20K | 15,430 | 20,830 | 57.4% | 3.83% |
| 40K | 9280 | 13,520 | 59.3% | 4.18% |
| 60K | 7244 | 10,363 | 58.9% | 4.32% |
| 80K | 5926 | 9262 | 61.0% | 4.91% |
| 100K | 5145 | 8746 | 63.0% | 4.99% |
| 120K | 3816 | 8082 | 67.9% | 4.34% |

Trace #3 -- DFSID

| BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|
| 21,798 | 14,458 | 39.9% | 2.83% |
| 13,051 | 9760 | 42.8% | 3.12% |
| 9858 | 7763 | 44.1% | 3.49% |
| 8055 | 7151 | 47.0% | 3.95% |
| 7169 | 6724 | 48.4% | 4.20% |
| 5809 | 6081 | 51.1% | 3.62% |

Trace #3 -- Hash

## C-7. In-Core Residence Times[1]

| CORE SIZE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 20K | 3 | 148 | 15 | 11 | 19 | 18 | 13 |
| 40K | 7 | 110 | 38 | 30 | 48 | 39 | 15 |
| 60K | 5 | 122 | 67 | 50 | 80 | 65 | 22 |
| 80K | * | * | * | * | * | * | * |

Trace #1 -- Purged objects only

| CORE SIZE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 20K | 0 | 186 | 15 | 11 | 19 | 19 | 19 |
| 40K | 0 | 143 | 36 | 25 | 48 | 42 | 30 |
| 60K | 0 | 137 | 64 | 30 | 84 | 61 | 36 |
| 80K | 0 | 136 | 70 | 32 | 101 | 67 | 39 |

Trace #1 -- All objects

| CORE SIZE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 20K | 2 | 143 | 16 | 12 | 20 | 17 | 9 |
| 40K | 2 | 116 | 38 | 28 | 46 | 38 | 15 |
| 60K | 16 | 150 | 55 | 40 | 66 | 54 | 19 |
| 80K | 2 | 153 | 68 | 50 | 90 | 69 | 26 |
| 100K | 8 | 142 | 83 | 34 | 110 | 74 | 39 |
| 120K | * | * | * | * | * | * | * |

Trace #2 -- Purged objects only

| CORE SIZE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 20K | 0 | 254 | 16 | 11 | 20 | 17 | 12 |
| 40K | 0 | 187 | 35 | 24 | 46 | 36 | 20 |
| 60K | 0 | 170 | 48 | 31 | 63 | 49 | 27 |
| 80K | 0 | 161 | 60 | 36 | 90 | 64 | 38 |
| 100K | 0 | 153 | 76 | 43 | 100 | 75 | 40 |
| 120K | 0 | 153 | 76 | 45 | 102 | 76 | 40 |

Trace #2 -- All objects

| CORE SIZE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 20K | 1 | 357 | 13 | 10 | 18 | 17 | 17 |
| 40K | 3 | 248 | 35 | 28 | 44 | 41 | 26 |
| 60K | 5 | 219 | 55 | 42 | 82 | 66 | 36 |
| 80K | 6 | 199 | 78 | 58 | 116 | 85 | 33 |
| 100K | 3 | 190 | 89 | 51 | 146 | 96 | 48 |
| 120K | * | * | * | * | * | * | * |

Trace #3 -- Purged objects only

| CORE SIZE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 20K | 0 | 368 | 13 | 10 | 18 | 17 | 21 |
| 40K | 0 | 265 | 35 | 27 | 44 | 41 | 33 |
| 60K | 0 | 230 | 52 | 38 | 81 | 65 | 45 |
| 80K | 0 | 212 | 72 | 46 | 115 | 85 | 53 |
| 100K | 0 | 205 | 89 | 42 | 148 | 94 | 57 |
| 120K | 0 | 203 | 91 | 44 | 152 | 99 | 58 |

Trace #3 -- All objects

[1] All values are expressed in terms of 1K ticks.
*No objects were purged.

**C-8. Leaf/No-Leaf Page Faulting Rates**

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|---|---|---|---|
| 20K | 4577 | 4708 | 50.7% | 31.7% | 6176 | 3033 | 32.9% | 29.3% |
| 40K | 2997 | 3762 | 55.6% | 30.1% | 4064 | 2654 | 39.5% | 27.2% |
| 60K | 2488 | 3536 | 58.6% | 27.1% | 3448 | 2552 | 42.5% | 25.4% |
| 80K | 1631 | 3218 | 66.3% | 25.9% | 2615 | 2210 | 45.8% | 22.5% |
| | DFSID -- Trace #1 | | | | Hash -- Trace #1 | | | |

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|---|---|---|---|
| 20K | 6817 | 8678 | 56.0% | 24.1% | 11,491 | 3908 | 25.3% | 18.4% |
| 40K | 4626 | 6113 | 56.9% | 25.1% | 7591 | 3090 | 28.9% | 19.6% |
| 60K | 3944 | 5764 | 59.3% | 21.8% | 6756 | 2907 | 30.0% | 16.9% |
| 80K | 3460 | 5511 | 61.4% | 21.7% | 6016 | 2930 | 32.7% | 16.9% |
| 100K | 3102 | 5274 | 62.9% | 23.9% | 5572 | 2768 | 33.1% | 18.3% |
| 120K | 2656 | 5215 | 66.2% | 20.2% | 5122 | 2703 | 34.5% | 15.0% |
| | DFSID -- Trace #2 | | | | Hash -- Trace #2 | | | |

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|---|---|---|---|
| 20K | 10,581 | 8703 | 45.1% | 34.0% | 15,202 | 4109 | 21.2% | 32.2% |
| 40K | 7075 | 6848 | 49.1% | 26.9% | 10,094 | 3937 | 28.0% | 25.0% |
| 60K | 5753 | 5571 | 49.1% | 24.0% | 7643 | 3688 | 32.5% | 25.1% |
| 80K | 4771 | 5178 | 52.0% | 23.4% | 6425 | 3523 | 35.4% | 23.3% |
| 100K | 4054 | 4920 | 54.8% | 25.1% | 5637 | 3356 | 37.3% | 24.6% |
| 120K | 3031 | 4552 | 60.0% | 24.0% | 4590 | 2984 | 39.3% | 23.8% |
| | DFSID -- Trace #3 | | | | Hash -- Trace #3 | | | |

| | 30K | 60K | 90K | 120K | 150K | 180K | 210K | 240K |
|---|---|---|---|---|---|---|---|---|
| DFSID #1 | 30.1% | 47.1% | 29.9% | 11.8% | 8.6% | 6.2% | 4.6% | 4.6% |
| DFSID #2 | 22.5% | 26.0% | 10.9% | 5.6% | 28.1% | 3.7% | 4.7% | 2.6% |
| DFSID #3 | 22.9% | 31.0% | 15.9% | 9.6% | 5.8% | 4.6% | 5.4% | 4.2% |
| Hash #1 | 35.3% | 38.7% | 110.5% | 62.8% | 38.7% | 8.1% | 6.2% | 3.1% |
| Hash #2 | 27.8% | 31.2% | 71.2% | 33.8% | 22.5% | 22.1% | 15.1% | 13.8% |
| Hash #3 | 30.0% | 29.6% | 74.2% | 48.8% | 35.6% | 22.7% | 10.5% | 6.0% |

Page fault increases due to leaf references for the Paged Virtual Memory

### C-9. Warm-Start Page Faulting Rates

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|---|---|---|---|
| 20K | 6875 | 8015 | 53.8% | - 2.5% | 8949 | 5902 | 39.7% | - 2.3% |
| 40K | 3975 | 4893 | 55.1% | 7.3% | 5079 | 3773 | 42.6% | 9.1% |
| 60K | 3154 | 3535 | 52.8% | 7.6% | 3934 | 2753 | 41.1% | 14.9% |
| 80K | 741 | 2205 | 74.8% | 66.3% | 1290 | 1639 | 55.9% | 61.8% |

DFSID -- Trace #1 Hash -- Trace #1

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|---|---|---|---|
| 20K | 8878 | 12,351 | 58.1% | 1.1% | 13,993 | 7145 | 33.8% | 0.7% |
| 40K | 5882 | 8122 | 57.9% | 4.8% | 9117 | 4857 | 34.7% | 3.5% |
| 60K | 4834 | 7146 | 59.6% | 4.1% | 7490 | 4436 | 37.1% | 7.9% |
| 80K | 4182 | 5954 | 58.7% | 5.4% | 6757 | 3370 | 33.2% | 6.6% |
| 100K | 3009 | 4918 | 62.0% | 26.1% | 5022 | 2891 | 36.5% | 26.4% |
| 120K | 1311 | 3488 | 72.6% | 60.6% | 3079 | 1706 | 35.6% | 48.9% |

DFSID -- Trace #2 Hash -- Trace #2

| CORE SIZE | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. | BUFFER MISS | BUFFER HIT | HIT RATE | FAULT RED. |
|---|---|---|---|---|---|---|---|---|
| 20K | 16,954 | 20,061 | 54.1% | - 5.6% | 23,593 | 13,450 | 36.3% | - 5.1% |
| 40K | 10,047 | 11,724 | 53.8% | - 3.7% | 13,725 | 8065 | 37.0% | - 1.8% |
| 60K | 7282 | 7985 | 52.3% | 3.8% | 9519 | 5768 | 37.7% | 6.8% |
| 80K | 5698 | 5919 | 50.9% | 8.5% | 7164 | 4465 | 38.3% | 14.5% |
| 100K | 3718 | 3465 | 48.2% | 31.3% | 4309 | 2854 | 39.8% | 42.4% |
| 120K | 691 | 1813 | 72.4% | 82.6% | 907 | 1640 | 64.3% | 84.9% |

DFSID -- Trace #3 Hash -- Trace #3

| | 30K | 60K | 90K | 120K | 150K | 180K | 210K | 240K |
|---|---|---|---|---|---|---|---|---|
| DFSID #1 | 6.9% | 15.9% | 34.3% | 41.0% | 42.6% | 54.0% | 70.2% | 70.2% |
| DFSID #2 | 6.9% | 15.5% | 17.2% | 17.1% | 25.6% | 25.8% | 26.3% | 30.0% |
| DFSID #3 | 5.4% | 9.0% | 19.2% | 26.7% | 31.4% | 42.4% | 47.5% | 47.8% |
| Hash #1 | 5.9% | 6.3% | 14.3% | 25.3% | 39.4% | 43.7% | 43.1% | 50.9% |
| Hash #2 | 5.7% | 7.0% | 13.2% | 15.5% | 21.2% | 24.3% | 33.1% | 38.2% |
| Hash #3 | 3.4% | 4.0% | 4.9% | 11.8% | 18.4% | 26.1% | 30.6% | 43.8% |

Warm-start page fault reductions for the Paged Virtual Memory

# Appendix D

## D-0. Trace Data

| TRACE NUMBER | STORE OPERATIONS | BYTECODES EXECUTED | NEW OBJECTS | NEW OBJECTS REMAINING | OLD OBJECT DEATHS |
|---|---|---|---|---|---|
| #1 | 98,031 | 87,998 | 6757 | 90 | 18 |
| #2 | 41,779 | 39,659 | 2393 | 26 | 23 |
| #3 | 126,402 | 118,236 | 9014 | 5 | 0 |
| #4 | 150,532 | 145,197 | 8342 | 159 | 78 |
| #5 | 129,235 | 130,140 | 4795 | 82 | 57 |

| TRACE NUMBER | MAX STACK POINTER | LAST STACK POINTER | MAX OBJECT COUNT | LAST OBJECT COUNT | WORDS COPIED | WORDS KEPT |
|---|---|---|---|---|---|---|
| #1 | 1227 | 1208 | 48 | 8 | 970 | 608 |
| #2 | 314 | 262 | 44 | 2 | 452 | 348 |
| #3 | 149 | 42 | 62 | 0 | 75 | 75 |
| #4 | 1506 | 1504 | 181 | 11 | 1253 | 1083 |
| #5 | 484 | 482 | 58 | 54 | 227 | 227 |

| TRACE NUMBER | MAX WORDS NEW CORE | LAST WORDS NEW CORE | OLD-NEW STORES | NEW-OLD STORES | OLD-OLD STORES |
|---|---|---|---|---|---|
| #1 | 861 | 158 | 7 | 26,067 | 55 |
| #2 | 604 | 4 | 22 | 6862 | 132 |
| #3 | 1117 | 0 | 2 | 32,195 | 6 |
| #4 | 2174 | 56 | 6 | 31,394 | 129 |
| #5 | 1250 | 1134 | 2 | 25,327 | 203 |

| TRACE NUMBER | SMALL POS STORES | SMALL NEG STORES | ALL POS STORES | ALL NEG STORES |
|---|---|---|---|---|
| #1 | 12,597 | 7159 | 14,687 | 9253 |
| #2 | 4153 | 1002 | 5218 | 1099 |
| #3 | 15,827 | 5672 | 18,074 | 6168 |
| #4 | 13,627 | 3654 | 21,076 | 4265 |
| #5 | 9448 | 2195 | 16,310 | 2382 |

## D-1. Transient Object Lifetimes

| TRACE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| #1 | 1 | 55K | 13 | 4 | 53 | 269 | 1849 |
| #2 | 1 | 39K | 13 | 5 | 35 | 436 | 2869 |
| #3 | 1 | 116K | 30 | 8 | 103 | 333 | 4452 |
| #4 | 1 | 142K | 14 | 6 | 51 | 696 | 7041 |
| #5 | 1 | 51K | 13 | 6 | 25 | 264 | 1763 |

## D-2. Stack-like Memory Management

| TRACE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| #1 | 0 | 86 | 1 | 0 | 1 | 1 | 3 |
| #2 | 0 | 37 | 0 | 0 | 0 | 1 | 3 |
| #3 | 0 | 5 | 0 | 0 | 1 | 0 | 1 |
| #4 | 0 | 176 | 0 | 0 | 0 | 1 | 10 |
| #5 | 0 | 33 | 0 | 0 | 0 | 0 | 2 |

Long Deallocation Distance

| TRACE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| #1 | 0 | 29 | 1 | 0 | 1 | 1 | 1 |
| #2 | 0 | 6 | 0 | 0 | 0 | 0 | 1 |
| #3 | 0 | 5 | 0 | 0 | 1 | 0 | 1 |
| #4 | 0 | 154 | 0 | 0 | 0 | 1 | 5 |
| #5 | 0 | 33 | 0 | 0 | 0 | 0 | 2 |

Short Deallocation Distance

**D-3. Dynamic Pointer Distance**

| TRACE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| #1 | 0 | 75K | 13 | 4 | 157 | 1832 | 6046 |
| #2 | 0 | 38K | 25 | 5 | 542 | 2286 | 5988 |
| #3 | 0 | 115K | 8 | 3 | 77 | 6063 | 21,034 |
| #4 | 0 | 90K | 187 | 7 | 4K | 10,167 | 21,125 |
| #5 | 0 | 67K | 375 | 6 | 17K | 10,417 | 16,479 |

Positive Pointer Distance

| TRACE | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| #1 | 2 | 32K | 6 | 2 | 672 | 1294 | 2851 |
| #2 | 2 | 22K | 12 | 2 | 68 | 599 | 2182 |
| #3 | 2 | 113K | 12 | 7 | 91 | 3897 | 16,882 |
| #4 | 2 | 91K | 7 | 3 | 211 | 3367 | 13,065 |
| #5 | 2 | 53K | 2 | 2 | 23 | 1108 | 4907 |

Negative Pointer Distance

**D-4. Degradation of an Initial Placement**

| HOURS USED | MIN | MAX | MED | Q1 | Q3 | MEAN | ST. DEV. |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1563 | 419 | 11 | 956 | 521 | 501 |
| 4 | 0 | 1563 | 427 | 16 | 956 | 525 | 499 |
| 8 | 0 | 1563 | 434 | 18 | 959 | 528 | 499 |
| 12 | 0 | 1565 | 423 | 17 | 976 | 524 | 502 |

Static Pointer Distance

| Hours of Use: | 0 | 4 | 8 | 12 |
|---|---|---|---|---|
| Close Pointers: | 16.9% | 16.2% | 16.2% | 16.1% |

On-page (512-word) pointer ratio