Using Dynamic Sets to Reduce the Aggregate Latency of Data Access

David Cappers Steere

January, 1997
CMU-CS-96-194

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Mahadev Satyanarayanan, Chair
Garth Gibson
Jeannette Wing
Hector Garcia-Molina, Stanford University

This research was supported by the Air Force Materiel Command (AFMC) and the Defense Advanced Research Projects Agency (DARPA) under contract number F19628-93-C-0193. Additional support was provided by the IBM Corporation, Digital Equipment Corporation, and Intel Corporation.

The views and conclusions contained here are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of AFMC, DARPA, IBM, DEC, Intel, CMU, or the U.S. Government.

Keywords: Search, Distributed Systems, File Systems, Information Retrieval, Prefetching, Performance, Latency, Modeling, Dynamic Sets, World Wide Web

In memory of my Mother

For Jody

Abstract

Many users of large distributed systems are plagued by high latency when accessing remote data. Latency is particularly problematic for the critical application of search and retrieval, which tends to access many objects and may suffer a long wait for each object accessed. Existing techniques like caching, inferential prefetching, and explicit prefetching are not suited to search, are ineffective at reducing latency for search applications, or greatly increase the complexity of the programming model. This dissertation shows that extending the file system interface to support a new abstraction called *dynamic sets* can address the problem of latency for search without incurring the penalties of the other techniques.

A dynamic set is a lightweight and transitory collection of objects with well-defined semantics. An application creates a dynamic set on-demand to hold the objects it wishes to process. Adding dynamic sets to the system's interface results in two benefits. First, creation of a set discloses the application's interest in the set's members to the system. This allows the system to reduce the aggregate I/O latency of search through prefetching and reordering of requests. Second, dynamic sets provide direct support for accessing and manipulating sets of objects. Thus dynamic sets improve performance and functionality without unduly increasing the complexity of the programming model.

This dissertation describes the design of the dynamic sets abstraction, an implementation which adds dynamic sets to the 4.3BSD file system interface, and an evaluation of the implementation. The implementation allows several applications, including Unix search tools and a WWW browser, to access sets of Coda, NFS, WWW, and local file system objects. With little effort one can modify other applications to use sets or extend the implementation to allow access to other systems. The evaluation shows that dynamic sets can substantially reduce I/O latency for search on wide and local area distributed systems and on local file systems. For example, replaying traces of real users searching on the WWW shows that sets can reduce latency by over an order of magnitude across a range of factors.

Acknowledgments

An African proverb claims that it takes a village to raise a child. I have found that it takes a community dedicated to excellence and perseverance to produce a doctor of philosophy. I am extremely fortunate to have had the support of the best such community: the faculty, students, and staff of the School of Computer Science at Carnegie Mellon.

I would particularly like to acknowledge my advisor, Satya, for his patience, leadership, and support. Satya's track record as a researcher and educator are well know. The fact that I have completed this dissertation is a testament to his ability to lead, manage, and inspire his students to great things. He provided useful and insightful feedback for all my research ideas, and gave me the confidence to overcome difficult research problems. Any skill I have as a researcher is due in large part to him.

I would also like to thank my thesis committee for their valuable input and support. Garth helped me mature as a systems researcher by teaching me to critically analyze systems through experimentation. His insight and advice on all aspects of this work has been extremely valuable. Jeannette challenged me on many occasions to think beyond engineering aspects and to see the larger picture. Hector's input as both an outside committee member and an expert on databases challenged me to understand the impact of my work outside the domain of file systems. Although other students questioned my choosing such a demanding committee, both I and the dissertation are better as a result.

Various students at CMU have been an invaluable source of friendship, criticism, and advice to me over the years. In particular, Puneet Kumar, Hugo Patterson, Brian Noble, Jay Kistler, and Scott Nettles helped me grow in confidence as a researcher by offering critical evaluations, friendly advice, and warm words of support when needed. In addition, I have been blessed with a large circle of friends in Pittsburgh. They have supported me and my family, and added to our lives immeasurably. Lauri Lafferty deserves special mention for proof-reading early drafts of the dissertation as well as many hours of babysitting.

Several other groups of people are worthy of note. The members of the Coda and Odyssey projects have all helped me more times than I can count. They are all extremely talented individuals and I would relish the opportunity to work with any of them again. I would

like to thank my officemates over the years for putting up with me. Unfortunately for them I concentrate best when loud music blocks out all other distractions. The "Brew Crew", especially Jonathan, Gary, Sue, and James, sat through many tedious practice talks yet still were kind enough to give me a pleasurable new hobby. All the members of the virtual "zephyr" community at CMU have offered valuable technical assistance as well as entertainment these many years. They deserve a hearty "++".

I would like to thank my family for supporting me throughout my years as a student. In particular, my daughter Samantha and son Alex gave me added incentive to finish and a wonderful break from the slings and arrows of the world. Many people have asked me how I could finish a degree with two children. After years of coming home weary to their rejuvenating bright eyes and smiles, I wonder how anyone can finish without them.

Finally, my wife Jody has done more for me than I could ever possibly acknowledge or repay. Finishing this degree and getting on with our life is the best I can offer. Thank you, I love you.

David Cappers Steere
January, 1997

Contents

List of Figures

Chapter 1

Introduction

A central problem facing distributed systems is the high latency of accessing remote data. Latency is problematic because it reduces the benefit typical applications can receive from faster CPUs, and reduces the productivity of users who are forced to wait for data. Long I/O delays can be frustrating as well, especially if the variance in the delay is high. This dissertation demonstrates that a small, carefully designed extension to the system-call interface of an operating system can result in a substantial reduction in the aggregate latency of search applications, while maintaining the integrity of the interface.

The original insight of this dissertation is that the system-call interface of current operating systems is overly restrictive, and limits the system's ability to address this issue of I/O latency. To reduce latency, the dissertation proposes a carefully designed extension to the system called *dynamic sets*. Dynamic sets expose asynchrony to the application through a controlled and well-defined interface. This exposure allows applications to disclose information to the system about the application's future data needs, without unduly increasing the complexity of the programming model. The system can employ this information to reduce the latency seen by the application and user. The benefits resulting from use of dynamic sets include lower aggregate latency to access a set of objects, greater opportunity to adapt system behavior to changing resource availability, and a more powerful interface for applications that process sets of objects.

The remainder of this chapter provides an overview of dynamic sets. Section 1.1 introduces dynamic sets, and discusses their benefits and limitations. Section 1.2 presents the thesis statement and describes three key questions about its validity which are addressed by this dissertation. Section 1.3 sketches the approach I took in validating the thesis statement, and presents a road-map for the remainder of this dissertation.

1.1 Introducing Dynamic Sets

A dynamic set is a lightweight, transitory, and unordered collection of objects that is created on-the-fly by an application to hold the objects it wishes to process. An object's membership in a dynamic set is therefore an implicit hint to the system of the application's interest in the object. With these hints, the system can safely allocate resources to prefetch remote objects that are members of a set, reducing the latency seen by the application when accessing set members.

An application creates a dynamic set by opening it, supplying a membership specification that is evaluated by the system to determine the names of the set members. Applications can then process the set members by iterating on the set. Every call to the iterator returns a handle to an object which has already been fetched. As a result, the application sees either little or no latency to access the object's data. Applications can also choose to manipulate set membership, for instance by merging two sets to form their union. For example, one might create sets to hold the results of queries to two news services, and then intersect the sets to find stories common to both services.

1.1.1 Advantages of Dynamic Sets

Dynamic sets offer three chief advantages. First, use of dynamic sets can reduce the aggregate latency seen by applications to access set members. By prefetching, the system can exploit the available parallelism between servers or disks through concurrent I/O, can overlap I/O and processing, and can improve resource utilization. In addition, dynamic sets allow the application to indicate to the system that the members are no inherent order. This in turn frees the system to determine an efficient order in which to fetch members, employing knowledge of system state which it alone possesses. This benefit is unique to dynamic sets, and can result in additional savings over prefetching alone.

Second, this solution is consistent with widely accepted software engineering principles because it hides information between levels of the system. Dynamic sets offer a well-defined and controlled exposure of asynchrony to the application programmer, and do not unduly burden the programmer with the complexities of multi-threaded applications and asynchronous operations. In addition, applications can benefit from prefetching without requiring the programmer to possess knowledge of the system implementation or current state. The system also benefits from preserving a clean application/system boundary: dynamic sets disclose information that is independent of the particular application using sets, but that can be exploited by the system to improve performance. An added benefit is that greater knowledge of the application's future data needs enables the system to adapt its behavior dynamically to suit changing network performance or resource availability.

A third benefit to this solution is that it is well tuned to support search applications on a broad range of systems. Search is a process of identifying and filtering through a set of objects in order to find objects with some desired property. Search applications can use dynamic sets to hold the objects to be examined, iterating over the set to apply a filter to each object in turn. Although tuned for search, dynamic sets are useful to any application that processes groups of objects, suffers from substantial I/O latencies, can benefit from prefetching, and can tolerate reordering of requests. In addition, experiments presented later in the dissertation show that a single implementation of dynamic sets offers significant reductions in I/O latency for searches on the WWW, NFS, and local disk.

1.1.2 Limitations of Dynamic Sets

The primary limitation of this solution is that it requires modifying applications to use dynamic sets in order to receive benefit. This, in turn, requires access to an application's source code. Further, it is unclear how the process of rewriting an existing application to use dynamic sets could be automated. However, the extent of the modifications is often limited to a small portion of the program, and thus only a modest effort is required. For example, I have been able to modify several existing Unix tools like `grep` to use dynamic sets within several hours.

A second limitation to this solution is that the benefits of dynamic sets are not universal. Applications that do not process collections of objects, or searches that cannot easily identify the names of the members they want to process will not be able to use iteration over sets as the primary mode of access. As a result, these applications will not receive the full benefit of dynamic sets. In addition, applications which have strong ordering requirements are not good candidates for this approach. However, the class of applications that can use and benefit from dynamic sets includes search and data mining.

1.2 The Thesis

The thesis of this dissertation is that

> *Dynamic sets reduce the aggregate I/O latency for search applications in a diverse set of domains, through prefetching and reordering of requests. In addition, dynamic sets offer enhanced functionality by supporting iterators, associative naming, and direct management of sets of objects.*

There are several questions about the validity of this statement whose answers are not immediately obvious. The most important question is whether or not there is any opportunity in practice to improve performance through prefetching and reordering. If there are benefits, are the potential gains worth the cost of modifying a well established paradigm, the file system interface? Can the benefits be quantified and measured through experimentation? Are the benefits available under a variety of conditions, or will dynamic sets only help a small number of specialized applications?

A second collection of questions involves the design of the dynamic sets abstraction. Can a simple interface be designed that provides sufficiently accurate hints yet is easy to program and use? Can such an interface extension be integrated with the file system in a seamless manner? What properties or semantics should an abstraction like dynamic sets possess? Can an implementation offer consistency guarantees that meet application requirements without significant changes to the system?

A third collection of questions concerns the difficulty of implementing the dynamic sets abstraction. How difficult is it to add dynamic sets to an existing system? Will the implementation of the abstraction be easily ported to new systems, or does it require substantial modifications and dependencies to the system to which it is added? Can different types of systems be smoothly integrated under dynamic sets? Can the resulting implementation achieve balance between fetching aggressively and avoiding oversubscription of resources in a single mechanism that works well for a range of systems? A poorly tuned prefetching engine or one that erroneously prefetches objects that will not be used can result in loss of performance. In large distributed systems, a single misbehaving client may inadvertently degrade the performance of the entire user community.

1.3 Thesis Validation and Road-map

The purpose of this document is to answer these questions by demonstrating the benefits of dynamic sets for real search applications on existing systems. The validation of the thesis is based on the design of the dynamic sets abstraction, its implementation as an extension of the 4.3 BSD file system interface, and experiments which examine the performance benefits of dynamic sets. This implementation includes a prefetching engine designed to work on a variety of file systems, and is integrated with a number of different file systems. The experimental results show impressive performance benefits on three different systems due to both prefetching and reordering. In addition, dynamic sets effectively eliminated I/O stalls in some circumstances.

The remainder of this dissertation describes the design, implementation, and evaluation of dynamic sets. Chapter 2 provides motivation for dynamic sets, as well as a brief background on relevant distributed systems concepts. Chapter 3 discusses the design

rationale and design of the dynamic sets abstraction, and closes with a short list of application domains in addition to search for which dynamic sets should work well.

The implementation of dynamic sets as an extension to the 4.3 BSD file system is described in Chapters 4, 5, and 6. Chapter 4 describes the basic architecture of the dynamic sets abstract data type, including a description of the interface between the dynamic sets abstraction and underlying file systems and a brief discussion of the prefetching engine. Chapter 5 discusses how I modified existing application to use dynamic sets, and the implementation of various *wardens* which allow dynamic sets to interact with a number of distributed systems. Chapter 6 describes the issues involved with prefetching, and gives a detailed description of the prefetching engine.

Chapters 7, 8, 9, and 10 describe the experiments used to demonstrate the performance benefits of dynamic sets. Chapter 7 presents a simple performance model to help understand the factors affecting the performance of prefetching. Chapter 7 also describes the experimental methodology of three experiments, each of which examines the performance benefits of dynamic sets to search in a different domain. The experiment described in Chapter 8 uses trace replay to determine the benefits of dynamic sets on the WWW. Chapter 9 presents measurements from a synthetic benchmark which show the performance benefits of dynamics sets to Unix search tools on NFS. Chapter 10 presents the results of a similar experiment on files stored on the local disk.

The dissertation concludes with a discussion of related work in Chapter 11, and a discussion of future work in Chapter 12. The concluding chapter also summarizes the key results, and presents the contributions of this dissertation.

Chapter 2

Motivation for Dynamic Sets

The problem addressed by this thesis is the high latency of accessing remote objects in a distributed system. This chapter discusses this problem in more detail. Section 2.1 starts by discussing the factors which contribute to I/O latency, and argues that existing solutions are not always appropriate, and in particular do not work well for search. Section 2.2 then describes search and retrieval, an important class of applications, and presents several examples of search in distributed systems. Section 2.3 presents four drawbacks to current systems that limit their ability to support search, which leads to the solution proposed by this dissertation, discussed in Section 2.4. The chapter concludes with a background discussion of naming and consistency, two aspects of distributed systems that are related to the design of dynamic sets.

A distributed system is a collection of interconnected computers that facilitate sharing of information between members of the group. Computers called *servers* export data objects such as files, pictures, or movies to other computers called *clients*. Usually the server or client is a program running on the server or client machine. To differentiate between the machine and the program, I refer to the former as the "*client*" and the latter as the "*client subsystem*"; "*server*" and "*server subsystem*" are similarly defined. Some systems do allow a machine to act both as a server and a client simultaneously, but in practice the client and server are on different computers. Special computers called *proxies* act as intermediaries between clients and servers. For instance, some companies that protect their internal network put a proxy on their firewall[1] to allow internal clients to access data on external servers.

For purposes of this dissertation, the two interesting classes of distributed systems are distributed file systems (DFS) and global distributed information systems (GDIS).[2] Both

[1] A firewall is a special router which acts as a barrier between internal and external networks.

[2] To ease exposition, the dissertation will confine the term "distributed system" to refer to just these two classes.

classes provide access to persistent objects which are stored on servers, use client caching to reduce latency, and use the basic open/read/write/close operations that are similar to those contained in the interface of the local file system. The chief differences between them are the scale, mode of access, organization, and autonomy of the components typical of systems in each class. Examples of DFS include Coda[82], NFS[77], and AFS[34][3]. Examples of GDIS include the World Wide Web (Web or WWW)[6], FTP[70], and Gopher[52].

2.1 Latency of Remote Access

One frustrating aspect of using distributed systems is the interminable delays one sometimes see when accessing remote data. A major reason for slowness is *latency*, the time between the request for data and the arrival of the reply. This section argues that high latency is endemic to distributed systems in the sense that remote operations are always slower than equivalent local operations, and that remote accesses are often much longer due to latency. It then describes the existing techniques used to address latency, and why these techniques are often not sufficient to reduce the impact of latency on performance.

2.1.1 Latency Is Endemic to Distributed Systems

To understand why latency is a fundamental aspect of remote access, first consider the work that must be done to fetch remote data. An access begins when the application demands some data object from the system. The client formulates and sends a request packet for the data to the server. Request packets are typically small, and so do not consume much network bandwidth. As the packet travels to the server, it may be routed through different physical networks, the number depending on the topology between the client and the server. Local area networks may only have a couple of these *hops* to traverse, whereas packets in a wide area network may traverse tens of hops. Congestion on these intervening hops may further delay our packet.

When the packet arrives at the server, it is queued with other incoming packets. When the server has processed other requests preceding the packet in the queue, it can begin to service our request. Servicing the request consumes server CPU, and may require disk accesses. Although it is rare, the server may also need to contact other servers at this point introducing further delays. When the server has satisfied the request, it bundles the

[3]The boundaries between DFS and GDIS are not firm, and one could consider AFS to reside in either class. Within a cell, AFS resembles other distributed file systems. Accesses between cells, however, is more like a GDIS.

results and sends them back to the client, where they are passed up to the application. When accessing a remote object, the results consist of the object's data.

One source of latency is *propagation delay*, the time it takes the request to travel to the server and the response to travel back. Even ignoring congestion, the latency from propagation can be substantial. In GDIS, packets may have to travel long distances, and so suffer significant propagation delays due to the speed of light limitation. For example, it takes light roughly 15 milliseconds to travel from Pittsburgh to Berkeley, CA. A photon's round trip between coasts is therefore as long as two local disk accesses. When overhead for protocol processing and routing is figured in, the time is much higher: sending a small packet round trip between Pittsburgh and Berkeley can take 200 milliseconds and makes 17 hops[4].

Another factor in high latency is network and server load. Multisecond response times are common when accessing popular servers on the WWW, even when the communication delay should be negligible. For instance, users at CMU often see long delays when accessing data off the server `www.cs.cmu.edu`, even though it resides on the same physical network. In addition, overloading servers and network components can cause temporary failures, which delay clients of the system for the duration of the timeout period. Because of the large variance in response time due to slow links, congested routers, and overloaded servers, these timeouts are necessarily much longer than the average response time. Although one would expect these failures to be rare, on a system the size of the WWW they are common enough to be a substantial source of latency over time.

An interesting aspect of latency is that the tremendous technological advances in CPU speed, disk bandwidth, and network bandwidth will not substantially reduce it. First, propagation delay is bounded below by the speed of light, which is a hard limit. Second, technological advances are consumed by growth in usage. Although network bandwidth, disk bandwidth and capacity, and server CPU speeds are doubling every year and a half, usage of these resources is also growing as fast or faster. On the Internet, for instance, the amount of traffic has doubled within several months and has been climbing sharply within the last few years[5]. This increase in load is due to both growth in users as well as the larger file sizes common in multimedia applications. Third, the decomposition of the Internet into subnets introduces many hops between network segments, each of which adds to the latency. The hops result from administrative artifacts and physical limitations, and thus cannot be eliminated by technology, although faster router hardware can decrease the cost of internetworking. Finally, as CPU speeds increase exponentially while I/O latency does not, the relative contribution of latency to the performance of many applications will rise. Thus the impact of latency may actually worsen due to

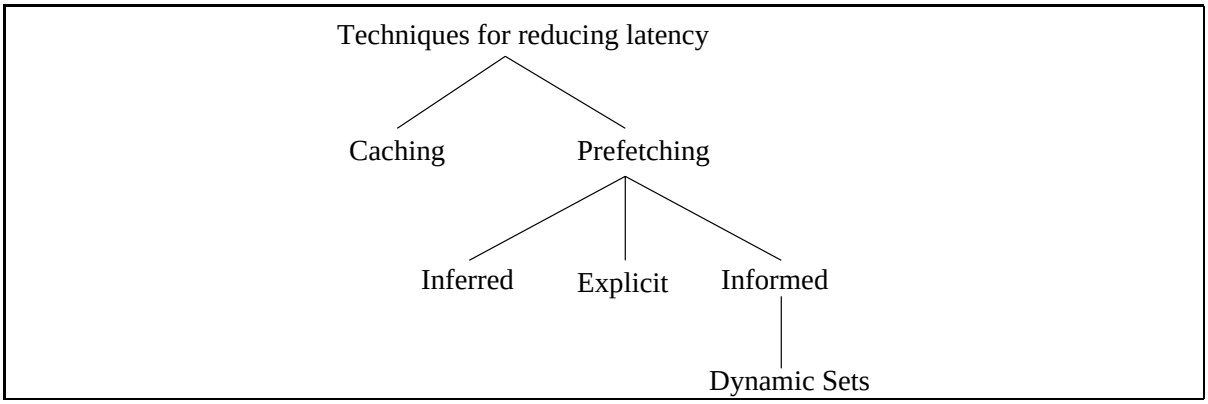
[4]Based on the results of the `traceroute` utility run on a computer at CMU in July, 1995.

[5]From statistics of NSFNET backbone usage collected through early 1995 by Merit Network, Inc., available as ftp://ftp.merit.edu/statistics/nsfnet.

improvements in technology!

2.1.2 Reducing the Latency of Remote Access

Two techniques are commonly used to reduce the effect of high latency: caching and prefetching. Prefetching can be further divided into 3 classes: inferred, explicit, and informed prefetching. This section discusses caching, inferred prefetching, and explicit prefetching, and describes why these solutions by themselves do not solve the issue of latency in general, and are not appropriate for search applications in particular. The dynamic sets abstraction is an example of informed prefetching, which is discussed in Section 3.2.3.1. Figure 2.1 shows the hierarchy of these techniques, and the location of dynamic sets in this hierarchy.



This figure shows the hierarchy of techniques to reduce latency. The dynamic sets abstraction is an example of informed prefetching. The other techniques are either not effective for search, or complicate the programming model.

Figure 2.1: Hierarchy of Techniques to Reduce Latency

2.1.2.1 Caching

Caching involves keeping local copies of recently accessed objects, and using these cached copies to satisfy future requests. Caching consumes local resources in the hopes that the workload will exhibit *temporal locality*, the likelihood that the object currently being accessed will soon be accessed again. If the user requests a cached object, the request

can be satisfied without a remote access, and is called a *cache hit* or just a *hit*. Conversely, requests that are not satisfied by cached data are called *misses*. Since local resources are finite, a cache's capacity is limited. If a cache does not have sufficient space to hold an incoming object, some cached data must be evicted to make space. Most caches employ a *least recently used (LRU)* policy, in which the object that has been accessed least recently is chosen for eviction.

There are two limitations to caching. First, many important applications, such as search, do not exhibit temporal locality. Caching cannot reduce latency on the first access to an object, and does not help if the object is evicted before it can be used to satisfy a request. Second, caches only benefit applications whose working set fits into the cache. If the working set is larger than the cache, the system will evict one member of the working set in order to cache some other member. When the evicted object is recached, it will displace yet another object. In the worst case, every access becomes a cache miss, and the cache is said to *thrash*.

In addition to these limitations, caching data that will not produce a cache hit can degrade performance. Caching consumes local resources such as memory or disk space, and may require computation and network bandwidth to assure the currency of the data in the cache. In addition, caching an object in steady state requires the eviction of some other object. For an application with poor locality, the evicted object may be referenced sooner than the cached object, which will force an extra cache miss and synchronous I/O operation to refetch that object. Applications like search that access many objects may completely flush a cache of all but those objects used in the search, reducing the cache's hit rate and increasing the average cost of I/O for all users of that cache.

2.1.2.2 Inferential Prefetching

Prefetching anticipates future accesses to an object and initiates the fetch operation on it before the object's data is requested. If initiated far enough in advance, the fetch may have completed before the object is requested and the application will not suffer any latency. The difficulty in prefetching is accurately predicting the future. Inaccurate predictions lead to fetching objects that are not going to be used. Since prefetched objects are stored in the cache, these inaccurate predictions can evict valuable data leading to future cache misses. Inaccurate predictions can also degrade performance by overloading the disk, network, or server, which are commonly the bottleneck of I/O intensive applications[44, 87].

The most common form of prefetching *infers* future activity based on observations of past accesses. The idea is that people tend to access information in roughly the same pattern as they have in the past. Of course, this presumes locality since the system must have observed this pattern many times in the past in order to detect it now. Inferential

prefetching is thus poorly suited to search, and is likely to produce inaccurate predictions. In addition to harming other applications at this client, squandering network bandwidth and server capacity on speculative predictions in this way can negatively impact the other users of a distributed system. On a GDIS, not only is this behavior considered inappropriate by the millions of other users sharing the Internet, but it may also prove costly if Internet providers begin to base access charges on bandwidth usage.

2.1.2.3 Explicit Prefetching

Another form of prefetching exposes I/O operations to the application, and has the application manage prefetching by explicitly invoking asynchronous I/O operations. The advantage of this approach is that it avoids erroneously prefetching objects that are not going to be used: the application knows which objects it will need in the future, and can fetch them accordingly. In addition, a correctly tuned application can receive the benefits of prefetching without requiring modifications to the system.

The drawbacks to this approach far outweigh these benefits, however. First, managing asynchrony greatly complicates the task of the programmer. To gain the benefits of asynchrony, the application must manage buffers, connections, and multiple threads of control itself. Multi-threaded applications are often much harder to write and debug than single-threaded applications, introducing race conditions, deadlock detection, and many other issues. Second, having the application manage asynchrony is less efficient. For instance, if the application were prefetching Coda files, it would have to maintain its own cache of the data in addition to the Unix buffer cache in the kernel and Coda's cache container file on disk. Third, applications are unaware of the state of the resources in the system, and are not always able to order the requests efficiently. For instance, the location of files is purposely hidden from applications in many DFS, and so an application would not know to parallelize access to files stored on independent servers. Fourth, the system may be unable to differentiate between the application's synchronous and asynchronous requests, and so cannot prioritize the operations appropriately.

2.2 Search and Retrieval in Distributed Systems

The problem of high latency is especially problematic for search applications on distributed systems. One aspect of distributed systems that make them interesting is that they provide access to a large wealth of information. Unfortunately, finding a particular piece of information, or determining whether such information exists in the system, can be nontrivial. This is especially true of GDIS, which have thousands of information sources, from the status of the latest NASA Space Shuttle mission at `http://shuttle.nasa.gov/`

(including images, a tracking display, and information about the crew), to an online "electronic museum" at `http://sunsite.unc.edu/expo/ticket_office.html`, containing information and images from the US Library of Congress. It is also true of typical DFS installations, which have many servers storing thousands of objects. Effective use of these systems therefore hinges on one's ability to locate data objects of interest. This process of locating and retrieving information is called search and retrieval, or simply *search*.

Search in a distributed system is an iterative process of identifying a set of candidate objects and examining the objects to determine which if any satisfy the search criteria. Identification is the process of determining the names of relevant objects, and may be done by the user directly or with the aid of a search engine. The examination is done by fetching the objects and applying a filter to them. Objects satisfying the filter may then be passed on to the next stage of the search. The job of each stage is to reduce the search's focus to a smaller set of candidate objects. As the set of likely objects grows smaller, more expensive filters can be applied to narrow the search further. Search ends when the current set of objects satisfies the searcher's goals.

Although the behavior of individual searches can vary, all searches share some common characteristics. First, because search accesses many objects, it is particularly subject to the high latency of remote access. The effect of the aggregate latency to access all the candidate objects is a significant portion of the time to search, especially when the amount of processing to filter objects is small. Second, search is a read-only activity, although the user may wish to modify the located object after terminating the search. Third, search tends to exhibit poor locality. One reason for this is that searches for unrelated objects are likely to access disjoint sets of candidate objects. Another is that once an object is discarded from the search, it is unlikely to be reaccessed as part of the same search.

An assumption made by this dissertation is that the duration of search tends to be much shorter than the lifetime of the candidate objects. As a result, the membership of a set created at the beginning of a search is likely to stay current for the duration of the search. There are two reasons why this assumption is reasonable. First, searches tend to focus on objects holding reference data, which have relatively long lifetimes. Reference data includes things like magazine and newspaper articles, technical reports, versioned source code, movies, photographs, and travel information. Once published, these objects tend to change slowly if at all. For example, a newspaper article is never modified once published, although it may be taken off-line after weeks or months. Second, the duration of a search tends to be self-limiting. If the search takes too long to perform, early results may be useless by the time the search completes. For instance, a search for movie listings that takes more than a day to perform is likely to produce outdated results.

The following sections characterize some of the kinds of searches targeted by the disser-

tation by providing three examples. Each example is used to illustrate particular aspects of search, although many of the aspects are common to all the examples. These examples and the characterization of search will be used again to describe the design of dynamic sets in the next chapter.

2.2.1 Example 1: Simple Search

Suppose one wanted to find the definition of a global variable in the source code of a large program. On a Unix-like system, the simplest way to perform this search would be to run "`grep varname *.c`" in each of the program's source directories. `Grep` is a pattern matching program that locates occurrences of strings matching a pattern (`varname` in this example). It opens and sequentially reads each of the input files (those whose names end in "`.c`") serially. This example is a single-stage search which employs the user's knowledge to identify the candidate files, and uses `grep` as the filter.

Although simple, this example is typical of many searches. The names of the files to be processed are known early in search, and the set of candidate objects does not change while the search is running. None of the files involved in the search are modified, and all are read sequentially in their entirety. And like most searches, this search is I/O intensive: `grep` does very little processing per byte; most of the time to perform this search is taken by I/O.

It may be noted that this search is potentially inefficient. Often the files in a group of source directories correspond to several programs. In order to find the variables used by a single program, it is only necessary to search through the source files that compile that particular program. However, it is easier for users to type "`*.c`" than to more exactly specify the desired set, and this leads to inefficient searches. One solution is to automatically parse the source files and build an index of locations of variable definitions, as is done by `etags`[6]. Another solution would be to determine dynamically which files are used to compile the program, and limit the search to those files. Either solution amounts to using tools to identify the candidate objects instead of relying on the user to do so.

2.2.2 Example 2: Using Search Engines

Suppose one were interested in locating WWW objects that contained information on cows. One would start by choosing a search engine and sending it a query asking for objects that contain the word "cow". The search engine responds with a hypertext

[6] `etags`, available at URL `ftp://prep.ai.mit.edu//pub/gnu/GNUinfo/ORDERS`, is part of the Free Software Foundation's gnu-emacs utilities.

document containing the list of names of objects that it knows contain the word "cow". One could then fetch and display some or all of the objects.

This example illustrates several important points. First, many searches on GDIS begin by querying a search engine to determine the names of the objects that are likely to satisfy the search. Second, the searcher in this example is human, and thus processing happens at human speeds: on the order of seconds. This means that the system would have substantial opportunity to prefetch the next object if it knew what the user was going to select. Since the user is productive while reading the data, it also means that the appropriate performance metric is the apparent delay between the request for and return of an object, not the elapsed time for the search. Third, the search criteria are fuzzily defined. The searcher has not specified which objects are of interest, or even what type of objects are desired. Further, the searcher is satisfied even if the system does not return all related documents as long as some interesting documents are found.

This example also raises two interesting points. First, the result set of a query may be quite large, both in terms of cardinality and the aggregate size of the members. Since the resources of any one client are limited, any mechanism which prefetches set members must also manage resources to avoid overrunning the client or network. Second, search engines often provide estimates of how closely the members of the result set match the query. If the estimate is correct, a member with a higher rank is more likely to satisfy the search. As a result, a prefetching strategy which takes these ranks into account may produce the desired data more quickly than one that does not.

It should be noted that this kind of search is the most common on the WWW. Popular clients like Mosaic and Netscape have only rudimentary support for using automated filters, and the WWW is too large for users to know a priori the locations of likely candidates.

2.2.3 Example 3: Multi-Stage Search

The previous two examples consider single-stage searches in which a group of objects is identified and then processed. This example shows how a searcher may wish to refine the membership of a set before examining the members. As an example, consider a personalized daily electronic newspaper. Using the multitude of online news sources, one could search for articles pertaining to, for example, cattle and the meat industry. Currently, no one source indexes articles from multiple newspapers. Instead, each news source, such as the San Jose Mercury News (`http://www.sjmercury.com/`) or the Wall Street Journal (`http://update.wsj.com`) provides indexes to its articles[7]. To collate

[7]In locating these papers, I found references to over 500 online services from newspapers around the world!

the articles from various sources, our user would run a search on each of the indexes, and merge the results to obtain a single set of candidate objects.

A personalized newspaper is an example of a multi-staged search. Each query produces a set of objects. Rather than examining the queries' results immediately, the sets are merged to produce a new set. This new set may also be further refined before arriving at the set of objects that will be read. For instance, the new set may be too large to warrant the system fetching all the objects, such as would be the case if the aggregate size of set's members was several hundred kilobytes and the client was connected over a low bandwidth phone line. In this case, one may wish to fetch a subset containing only small objects, or alternatively fetch only those articles concerning Holsteins.

This example also shows the dynamic nature of search. Each day one runs exactly the same query, but gets back a completely different set of objects. In addition, the set of articles identified by today's query is only valuable for a limited period of time; at most the set is valuable until tomorrow's articles are published. Furthermore, the set has weak currency requirements. One would be satisfied if the articles were slightly out of date, say on the order of hours, as long as they were not days old and completely out of date. For instance, it would be quite reasonable to run the queries automatically every morning, and preserve the set until one had read the articles later in the day.

2.3 Drawbacks of Existing Systems

So far this chapter has stated that latency is problem in distributed systems, and is a particular problem for the critical application of search. This section lists four drawbacks of existing systems that limit their ability to provide better performance and support for search applications.

The first drawback of current systems is that *there is no way to expose to the system the relationship among a group of objects.* Without this support, applications must orchestrate access to the group themselves. Either they access the group members serially, and suffer the resulting performance penalty, or they use explicit asynchrony and become significantly more complicated. Applications which use explicit asynchrony to prefetch end up with multiple copies of data, keeping copies in their virtual memory in addition to the copies in the Unix buffer cache and/or disk. This also leads to poor paging performance, since the paging subsystem is ill equipped to handle prefetching of this sort. Modifying the pager to better manage prefetched data is a risky proposition, and previous systems which have modified the pager to support more advanced paging strategies have suffered serious performance penalties as a result[17].

A second drawback is that *current systems do not support iteration over groups of objects.* Iteration is a natural way of processing groups of objects, and is a key feature of

many high-level programming languages such as Alphard[85], CLU[48] and Modula-3[32]. Providing system support for iteration not only provides a more powerful file system interface, it also provides the system an opportunity to improve performance. For instance, a system could prefetch a member before yielding it, reducing the latency to access the members. Further, it could ensure that a group member's data was in memory before yielding it through the iterator to reduce the disk latency of reading the object's data.

A third drawback is that *current systems force applications to process groups of objects in an imposed order*. Because the current distributed systems only support access to individual files, accesses to the group must follow some serial order. Often the ordering is supplied by a third party, such as the Unix shell. Although the imposed order may occasionally be useful, it is often the case that the application does not care in which order the group members are processed. If the order could be left to the system, the system could exploit information only it knows in order to reduce the aggregate latency of access. For instance, the system could order objects such that cached members are yielded before uncached members. Thus the latency to fetch the uncached members could be overlapped with the processing of the cached data. Alternatively, fetches could be ordered to take maximal advantage of parallelism between servers, providing further reductions in latency.

A fourth drawback is that *current systems only provide persistent collections*. Systems allow groups of objects to be co-located in directories, and some even provide the means of clustering the files within a directory to speed access. However, creating a directory and storing it involves updating persistent storage, and so entails a significant overhead. In replicated systems, persistent updates may involve distributed transactions as well. The cost of a persistent update makes directories inappropriate for holding the results of short-lived collections like the results of queries. In addition, many query results have little value when the search is finished, so persistence offers no advantage to offset its costs.

2.4 The Solution: Dynamic Sets

To summarize, the problem addressed by this dissertation is the high latency of accessing remote data and its impact on the performance of search applications. The problem of latency for search is interesting because existing solutions to latency such as caching or inferred prefetching either offer no performance benefits or have a high cost associated with them.

Fortunately, the very nature of search leads to a solution. Search is primarily a process of identifying sets of candidate objects and processing them. If applications could inform

the system of the membership of these sets, the system could prefetch the members to reduce the aggregate latency seen by the application when accessing them.

This dissertation proposes extending the file system interface with a simple and well-defined abstraction called *dynamic sets*. Dynamic sets address the drawbacks listed above, thus providing better support for search applications. In addition, they disclose information to the system about the application's future data needs, allowing the system to determine the order of prefetching.

2.5 Background: Naming and Consistency

Before discussing dynamic sets in more detail, this chapter presents two aspects of distributed systems relevant to their design: naming and consistency. Naming is relevant because determining the membership of a dynamic sets involves naming (identifying) groups of objects. The process of creating and populating a set is thus closely tied with name resolution. Consistency is relevant because the dissertation must provide suitable guarantees in order to support the needs of search applications. The following two sections describe naming and consistency in more detail, and present terminology that will be used throughout the dissertation.

2.5.1 Naming

All objects in a distributed system have a name and contain data. In order to access an object's data, an application must first *open* the object's name. In response to the open, the system performs *name resolution* on the name to locate the object and returns a *handle* which can then be used to access the object's data. The application can then *close* the object to inform the system it no longer needs to access the object's data.

The structure of the space of names provided by a system is called its name scheme. There are four types of name schemes: flat, hierarchical, hypertext, and associative; systems often provide more than one scheme for greater flexibility or efficiency. The first three kinds of name schemes are structured as graphs, where each node is named by its path from some distinguished object called the root. Associative name schemes are different in that objects are named by attribute and may not have a fixed name. Applications supply a *query* which specifies a list of desired attributes, and the system evaluates the query to identify the names of objects that satisfy the query: those objects possessing the desired attributes.

Search engines are special servers which provide fast query evaluation by identifying the object names without having to access the objects themselves. Search engines do this

by maintaining an index which maps attributes to object names. This index is built by periodically crawling over the name space, extracting the attributes of every object it visits, and inserting each object's name and attributes into the index. Because an object's name is usually orders of magnitude smaller than the object's data, it is feasible for a single search engine to catalog most or all of the objects in very large systems.

It is often the case in distributed systems that obtaining an object's name takes less time than obtaining an object's data. Names may be cached at a client, embedded in related objects, or stored on a search engine. It is this property that makes dynamic sets practical; otherwise the process of creating a set which involves collecting names of objects might be more time consuming than the potential savings from prefetching the objects.

2.5.2 Consistency

Consistency is an umbrella term that encompasses the issues of coherence, currency of data, isolation, and the legality of a sequence of operations. *Coherence* applies to systems that replicate objects, and is the property that all replicas (either first class server replicas or second class cache copies) reflect the same history of updates at any point in time. *Currency* refers to whether or not an object's latest state is visible to readers. A system provides *isolation* if it ensures that the intermediate results of a computation are not visible to others. *Consistency* is most commonly used to mean that the system only allows sequences of operations that bring the system from one legal state to another, where legality is defined by the application.

Although consistency is clearly desirable in theory, in practice it is often too difficult or expensive to provide. For instance, ensuring currency requires tracking all updates in the system; coherency requires propagating or detecting updates when they occur; isolation requires preventing concurrent access to objects that might reflect intermediate results; and legality requires either preventing certain sequences of operations from happening or detecting them when they happen and rolling back the offending operations.

Because of this, DFS typically relax the *one copy* semantics of the non-distributed system from which they are derived. For example, updates to an AFS file are only reflected globally when the file is closed, instead of being instantly visible as are writes to a local file in the Unix Fast File System[53]. Similarly, NFS allows cached data to be incoherent with the server's copy for bounded periods of time.

Unfortunately, the techniques used to ensure even the relaxed consistency of DFS tend not to scale as the cost of communication, the number of potential copies, and the amount of data rises. For instance, in a local area network one can inexpensively track updates to an object by multicasting (sending in parallel) the updates to all of the object's

replicas. However, multicast is too expensive in the WWW where thousands of machines scattered over the globe may have cached copies of an object. Since consistency is so hard to provide, GDIS typically do not offer consistency guarantees at all, or at best provide bounded currency. For instance, the most common form of replication in GDIS is called *mirroring*, where copies of popular data from some site are kept on another independent server, often in another country or continent. The mirror site often makes no promises about whether the copy reflects the same state as the original object, or at best it indicates when the mirroring was done and leaves the user to guess whether or not the data is current. The WWW also uses caching, a form of replication in which a client or its proxy keeps copies of recently accessed information[27, 15]. Coherence of WWW caches is approximated by invalidating the cache copy after some period of time in which an update is believed to have occurred. These systems typically use a heuristic based on the age of the object to determine the length of this period of time. For instance, one caching relay invalidates a cache copy after it has resided in the cache for a period equal to the difference between the object's last time of modification and the time it was cached[27].

Fortunately, this lack of consistency does not appear to be a serious concern to users of these systems. First, data in GDIS tends to change slowly if at all. The heuristic of the caching relay just mentioned is surprisingly accurate: its administrators randomly tested 5% of references to cached objects, and only 8% of the tests indicated that their heuristic would have returned stale data. Second, publishers of information on GDIS have developed policies to aid in the detection or avoidance of inconsistencies. Releases of a system's source code, for instance, tend to be packaged into one object to ensure that all of the objects in the release are internally consistent. This composite object is also labeled with the version number of the release so the identity and version of the release is self-evident. Third, most applications of GDIS apparently have little or no inherent need for strong consistency, which is demonstrated by the explosive growth in the popularity of these systems.

2.5.2.1 Correctness

Related to the concept of consistency is the concept of correctness, the ability of the system to correctly identify which objects satisfy a search. In the world of information retrieval, correctness of queries is described by the terms precision and recall. *Precision* reflects the degree to which the query's results satisfy the search criteria. A system produces *false positives* if it returns objects which did not satisfy the query. A system exhibits *recall*, also known as completeness, if all objects in the system that satisfy the query are returned. A system without a completeness guarantee can return *false negatives*, objects that should have been included in a query's result set but were not.

Chapter 3

Defining Dynamic Sets

This chapter presents the design of *dynamic sets*. Dynamic sets are lightweight, transitory, and unordered collections of objects that do not contain duplicates. An application creates a dynamic set to hold a group of objects, such as the candidate objects in a search or the objects named in a query's results. Membership in a set is determined at the time of set creation. When the application no longer needs the group, it can terminate the set to free up resources consumed by the set.

Section 3.1 presents this new abstraction, its properties, and the operations in its interface. Section 3.2 describes how the addition of dynamic sets can improve the usability of a system. Finally, Section 3.3 lists some application domains which would benefit from the addition of dynamic sets.

3.1 Design Rationale

This section discusses the design rationale of the dynamic sets abstraction. Section 3.1.1 lists the goals of dynamic sets. Section 3.1.2 describes the possible locations for the abstraction within the layers of the system. Section 3.1.3 describes the properties of dynamic sets, and Section 3.1.4 describes the operations in the programming interface. Section 3.1.5 shows how one might use dynamic sets in the three example situations described in Section 2.2.

3.1.1 Goals of Dynamic Sets

The chief motivation behind dynamic sets is to increase the usability of distributed systems by reducing I/O latency and increasing functionality. To achieve these ends, dy-

namic sets need to be general, flexible, lightweight, and must minimize dependence on changes to the underlying system. This section motivates and clarifies these goals.

3.1.1.1 Generality

The first goal is to make dynamic sets *general*. Search describes a range of possible behaviors and systems. The strength of the dissertation is greatly increased if the dynamic set abstraction covers much if not all of this range. The use of the mathematical concept of *sets*[1] as the underlying type for the dynamic sets abstraction achieves this goal. Any type of object can be a member of a set, although implementations can restrict the types of objects they support. The use of sets also produces a simple and elegant abstraction, since sets are powerful yet intuitive[22], and should be familiar to any user of a distributed system.

3.1.1.2 Flexibility

The second goal is to make dynamic sets *flexible* in order to support many different kinds of searches. The primary issue is to allow search applications to indicate set membership using a variety of sources. Set creators specify membership in the set when the set is created using a specification language. To achieve flexibility, the design leaves the definition of this specification language to the implementation, but the language should at least satisfy the following two properties. First, the specification language should support some form of an explicit list of the member names. This is the minimal functionality; a set creator could evaluate a more complex specification itself to determine the list of names, and supply that list to the system. Second, the language should support access to search engines by supporting some type of query. The information retrieval community has made significant effort to automate the process of identifying candidate objects for a search. Supporting queries allows dynamic sets users to leverage their advances. Further, letting the system control the execution of the query gives it one more degree of freedom in controlling the aggressiveness of the search.

3.1.1.3 Lightweight

The third goal is to make dynamic sets *lightweight*. Although efficiency is a goal of all engineering systems, it is especially important for dynamic sets because the opportunity to reduce latency may be limited in some contexts. For instance, a system may reduce the I/O latency of an application through prefetching, but may increase the application's

[1] The mathematical concept of sets is the definition of sets used in set theory[22].

runtime if it spends more time in additional computation to manage sets activity than is saved by higher I/O utilization. Similar decreases in performance can occur if the design requires excessive use of the disk, network, or server resources.

3.1.1.4 Only Require Modifications to Client

The fourth goal is to ensure that dynamic sets do not require *modifications to underlying system protocols or servers*. It is extremely difficult to modify the protocols of large distributed systems. Protocols are determined by standards committees, and as such tend to change slowly if at all. Requiring changes to servers is also problematic. In GDIS, it is usually the case that the server belongs to a different administrative domain than the client, and the server's administrators may not be willing to make the change.

Alternatively, modifying the client is appropriate since the costs of upgrading to a newer version of the client subsystem that supports dynamic sets is paid by the beneficiary. Each user can independently decide whether the benefits of dynamic sets warrant the inconvenience of upgrading, and achieve that benefit without imposing on other users of the system.

One implication of this goal is that dynamic sets should not promise stronger semantics than the underlying system can provide. Many target systems of this thesis such as the WWW or NFS do not provide support for strong properties such as isolation. Dynamic sets cannot strengthen these properties without resorting to enhancing the protocols or servers used by the system. For instance, it is impossible to guarantee that set operations preserve isolation unless the underlying system guards intermediate results.

3.1.2 Where Should Dynamic Sets Reside?

This section describes where and how support for dynamic sets should be added to a distributed system. One can imagine integrating them at any layer from the application down to the servers themselves. This question is really one of logical placement, since the boundary between various layers differs between systems. For instance, the client subsystem may be part of the operating system kernel in a Unix environment, but may be implemented as a user-level library in a micro-kernel such as Mach 3.0[1], a nano-kernel such as SPIN[9] or the Exokernel[23], or in a commercial operating system like Windows.

The best choice, and consequently the one taken here, is to add support for dynamic sets to the file system application programming interface (API), leaving the implementation of dynamic sets in the kernel of the client's operating system. This choice has three benefits. First, it allows the system to gain maximal performance benefit from the

hints inherent in a set. Second, it increases the generality of dynamic sets by allowing an application to include any data accessible through the file system in a set. Third, it strikes the right balance between software engineering concerns such as information hiding and implementation concerns such as efficiency. Unfortunately, it also makes the job of porting the implementation of dynamic sets to a new operating system more difficult. However, this penalty is an acceptable tradeoff for better performance and a simpler implementation for the purposes of this thesis. In addition, adding dynamic sets to the system interface makes it easier to port applications of dynamic sets, since the system-specific details are hidden beneath the interface.

Placing support for sets in the system API gives maximal value to hints implicit in a set because it exposes them to the client subsystem, the part of the system that can most benefit. Although the design does not require it, these hints can be exposed by the client to any portion of the system that may need access to them. Further, dynamic sets expose the hints as early as possible, to give the system maximal opportunity to reduce latency.

The hints from dynamic sets can be used by the client to drive informed prefetching and to schedule resource usage for improved utilization. Prefetching can be done because an object's membership in a set is a hint that the application will soon need to access the object. Prefetching a member reduces the latency seen by the application to access it, and results in deeper request queues, which gives the system greater flexibility to reschedule or batch requests to further reduce latency. Exposure of membership also gives the system knowledge which allows it to better adapt its behavior to suit changing resource availability. For instance, while it may be meet to prefetch data aggressively over a high bandwidth link, using the same strategy over a heavily loaded Ethernet or a slow phone line can result in a loss of performance. Also, the system knows that objects are stored on separate servers and can be fetched in parallel. Well designed systems deliberately hide this knowledge from the application in order to achieve location transparency.

Placing dynamic sets higher, for instance as a user-level library linked in with the application, would not disclose knowledge of a set's membership to the system. Although the application could prefetch the objects itself, it could not use the system's state to control the prefetching in systems which hide critical state information below their API. Further, the application's prefetching may result in poor utilization. For example, when an application is prefetching it must buffer the data in its virtual memory until it is used. Since multiple applications could be accessing the data, multiple copies of the same data may be in memory at once. The copies are superfluous if the system caches these objects as well, as most systems do. However, exposing the state of the cache or the client subsystem's data structures to the application would not only increase the complexity faced by the programmer, but would violate a principle of software engineering: always hide implementation details behind a well-designed interface.

Placing dynamic sets below the client subsystem requires modifying protocols and servers,

violating one of the design goals. Although doing so may open further opportunities to reduce latency or improve efficiency, it is left as a future enhancement of this work. Alternatively, one might consider distributing a set across machines to allow multiple clients to share access to a set. However, doing so would also entail costly update procedures to the data in a set which would require techniques such as two-phase commit in order to keep the set state consistent[63]. As such, the added benefit does not outweigh the cost in performance or complexity of implementation.

3.1.3 Properties of Dynamic Sets

The dynamic sets abstraction is designed to possess properties that are needed to support search on distributed systems while satisfying the goals listed in Section 3.1.1. The properties were chosen keeping in mind both the needs of search applications and the ability of the system to reduce latency. Each of the following subsections describes a property, why the property was chosen, and how the choice of that property influences the rest of the design. Many of the design choices described below are based on Saltzer et al.'s end-to-end approach to system design[76]. Functionality is added to dynamic sets when it benefits most or all potential applications of dynamic sets, has low-cost, or increases the efficiency of the system. Functionality that is not universally needed or is expensive to provide in the system is left to the applications.

3.1.3.1 Dynamic Sets Are Created on Demand

A key feature of dynamic sets is that they are created and their membership is determined on demand. Dynamic creation allows a set's membership to capture reasonably current information and ensures that the state of the members is the latest available state at the time the set is created. Both forms of currency are necessary since the results of many searches depend on the state of the system in which they are run. Searches like those used in the personalized newspaper example in Section 2.2.3 produce different results when run on different days; the articles in today's paper will be different from the articles in tomorrow's, even though the query used to create each day's paper is the same (e.g., "articles on cows").

There are three reasons why sets should be dynamic. First, it ensures that applications see current information by default. Applications can alternatively relax currency on a case-by-case basis by opening a set before it is needed, and holding it open (and thus in existence) until ready to process the members. For instance, a personalized newspaper can be precomputed every morning before the user awakes without violating the need for currency of this application. Additionally, a mobile user may wish to create a set and fetch its members while connected to a network to avoid future disconnected cache misses

when travelling. In these cases the potential loss of currency is acceptable because it is explicit, and the cost of precomputing and maintaining the set is paid directly by the user (or application) that chooses to do it. In addition, the application can achieve the right balance between currency and performance without special support from dynamic sets.

Second, the time to identify the members of a set is either shorter than, or can be overlapped with the time to fetch all the members. In the WWW, for instance, the cost of running a query on a search engine is comparable with the time to fetch a single object. Similarly, the time to parse a directory to discover the names of objects in a DFS is equal to or shorter than the time to fetch a file. In addition, the work to determine some members of the set can be performed while the client is fetching or processing other known members if membership is evaluated lazily.

Third, precomputed sets, the alternative to dynamically created sets, introduce a host of problems because they need to be stored and maintained. First, one must provide some reasonable currency and consistency of the set with the state of the system. In the worst case, every time an object is added, removed, or updated, the state of all sets must be reevaluated. In addition, it may be as costly to access the set and its members from its storage site as it is to calculate the membership dynamically.

The main disadvantage of creating a set dynamically is that the overhead of doing so must be paid directly by the application. Fortunately, most of the cost of creating a set is I/O latency. As stated above, this latency is often smaller than the latency of fetching the members, and can be overlapped with other I/O or processing through lazy evaluation. Further, indexes and search engines offer most of the benefit of precomputing membership. And the cost of fetching the objects just in time can be greatly reduced by prefetching, without significantly sacrificing currency.

3.1.3.2 Dynamic Sets Are Short Lived

Since sets need to be dynamic in order to satisfy the temporal nature of search, it is unnecessary to preserve a set beyond the termination of the application that created it. Sets can thus be maintained in volatile memory, making dynamic sets more lightweight. This property also simplifies the design, because it does not require solutions to the problems of ownership, protection, and administration of persistent data.

Making sets persistent necessitates updating non-volatile storage as the set is expanded and consumed. Doing this synchronously is expensive, since each disk access involves CPU overhead to make the request to the disk driver, and latency for the disk write to finish. Doing it asynchronously reduces the impact of disk latencies, but increases the complexity of the implementation without reducing the CPU overhead. For client

subsystems that do not cache data on the disk, the overhead from persistent updates may well exceed the potential savings from prefetching, eliminating the benefit of sets for these systems.

The benefits of persistence do not outweigh these costs. If the application that created a set terminates, it certainly no longer needs the set. Similarly, if the application or system crashes, the set will not be reused and so does not need to be preserved. Other applications that run the same query in the future will get a new set, because the results of the query may be different, and the set membership should reflect the new results.

Although sets are not necessarily persistent, applications can choose to make the set persistent by using mechanisms orthogonal to dynamic sets. For instance, an application could create a new directory on the local file system, and copy the set members to this directory. Thus particular applications can choose to create persistent sets without forcing all applications to pay the overhead for persistence.

It should be noted that although the sets themselves are temporary, the objects are persistent. This distinguishes the data structures that represent the set from the objects that are members of the set. For instance, a set of source files can be stored in memory and be short-lived, although the files themselves are stored on disk and are persistent. In addition, if the client subsystem caches to the local disk, most of the effort to cache objects before a crash is not wasted. Since the objects are likely to remain validly cached after the restart, the recreated set should benefit from the presence of these objects in the cache and will not have to refetch them.

independent of dynamic sets.

3.1.3.3 Dynamic Sets Are Unordered

One property that dynamic sets share with mathematical sets is that all members have equal weight or value. This frees applications from processing the members of a set in some imposed ordering, such as alphabetically by name which is done in current systems. In addition, dynamic sets give the system further opportunity to reduce latency by allowing the system to determine the processing order. For instance, the system can choose to fetch smaller objects before larger ones to minimize the time the application must wait for the first object.

Removing the ordering restriction from the system is an example of applying end-to-end considerations to systems design. Many applications either have no need of order or need some other order than the one provided by the system. For example all items may be equally likely to satisfy a search. Alternatively, the application may require a chronological ordering but the system may order the objects alphabetically. While not all applications require a particular order, enforcing order has a nontrivial cost. Ordering

requires CPU overhead to sort the members, and possibly additional I/O overhead to obtain attributes or meta information in order to perform the sorting. For instance, a chronological sorting may require additional I/O operations to determine the members' creation dates.

Enforcing a specific order also limits opportunities for the system to reschedule requests to reduce latency or to use resources more efficiently. Reordering avoids blocking the application on a slow fetch when other data is immediately available. It also allows the system to use cached data, or hold the data in the cache until it is needed. Without this ability, the desired data may be flushed out of the cache before it can be used, resulting in more I/O than would otherwise be needed. Reordering is particularly valuable in distributed systems which are subject to communication failures. These failures result in pausing for a *timeout* period, usually much longer than the expected time to fetch an object. If the first object is unavailable, the application would block for the duration of the timeout if the system was enforcing an ordering. By allowing the other objects to be processed first, dynamic sets can overlap the timeout period with the processing of the other objects, reducing the latency seen by the application.

The chief drawback of this decision is that dynamic sets may be inappropriate for applications with strong ordering requirements. However, many applications, particularly search on DFS or GDIS, do not have such strong requirements. In addition, applications that have partial-ordering requirements, or whose orderings can be expressed as equivalence classes, can use dynamic sets by creating a set to hold groups of objects with equivalent importance. Further, applications can add weights to members to indicate their relative importance. The system treats these weights as hints, using them to control the behavior of the prefetching engine. Alternatively, it is a simple extension to add some form of ordering on sets for those applications that need it. Applications could sort the results after processing the data in order to present them in the proper order.

3.1.3.4 Dynamic Sets Do Not Have Duplicate Members

Another property that dynamic sets share with mathematical sets is that an object can be a member of a set at most once. No application needs to see the same member twice, and having the system eliminate duplicate names prevents it from fetching the same object more than once. This leads to more efficient search because applications neither have to maintain state in order to avoid processing an object twice nor expend the effort of doing so. For simple applications like `grep` that print out the results of processing, this also reduces the amount of redundant information shown to the user.

It is not immediately obvious which of the various means of determining equivalence is appropriate. On one hand, detection of name equivalence is easy and has low overhead but fails to eliminate duplicates in the presence of aliasing, and aliasing is common in

DFS and GDIS. On the other hand, testing value equivalence would eliminate duplicates, but the cost of doing so might be prohibitive especially for large objects. Since the thesis focuses on performance, name equivalence is used as the test for equivalence in order to keep dynamic sets lightweight.

3.1.3.5 Dynamic Sets Are Immutable

Dynamic sets cannot be mutated by the application once they have been created. Thus there is no operation in the dynamic sets API by which an application can change a set's membership. In addition, applications are not allowed to modify the set members while the set is open. This simplification does not have a significant penalty. Search is read-only by nature, so target applications are not hampered by being unable to write set members. In addition, any operation that would modify a set's membership can be redefined as an immutable operation which returns a new set with the modified membership. The cost of immutability is that more sets may be created, and more sets may be in existence, as a result. Fortunately, this cost is negligible since sets are lightweight and the resources consumed by an open set is small. In addition, an application can close a set to release its resources if the number of sets becomes a problem.

3.1.3.6 Dynamic Sets Offer a Well-Defined but Weak Consistency Model

A client enhancement like dynamic sets cannot ensure a strong consistency property like one-copy serializability if the underlying system does not provide it. In addition, strong consistency guarantees are costly to provide (as described in Section 2.5.2). Thus dynamic sets can at best provide guarantees no stronger than the weakest consistency model of the system accessible through dynamic sets. However, offering no guarantees is also not acceptable. This section describes the consistency ensured by dynamic sets, explains why this level of semantics is acceptable, and then argues that stronger semantics are inappropriate for search.

Ideally, a dynamic set should capture a snapshot of the system: the query used to specify the set's membership should be run atomically (as if there are no concurrent updates) and the members should be fetched and locked to prevent them from being modified while the application is processing them. However, these semantics are costly to provide. Distributed locking necessary to prevent concurrent updates adds additional message exchanges to the work of fetching objects, and each exchange involves the high latency of remote access. In addition, locking an object to prevent mutations to it delays other applications that wish to update it. Queries may have non-trivial execution times, due to a high per-object computation or as a result of accessing many objects. Providing atomic queries would delay updates to objects locked by the query for the duration of

the query. In addition, thousands of searches are likely to be running at any one time in a large GDIS. Locking all the objects involved could effectively block writes to large portions of the system indefinitely.

Dynamic sets relax this ideal consistency by allowing updates which might affect the set's membership to occur while the query is running and by allowing mutations to occur to the set members. However, changes that occur after a set has been created and its members fetched are not reflected in the set. The semantics of dynamic sets require that the implementation prevent set membership from changing once it is established, and keep immutable cache copies of the members. The consistency model can be summarized by saying that all dynamic sets satisfy these two properties:

- Every member must have satisfied the query at some point during the lifetime of the set.
- Once an object is known to be a member, it will remain a member of the set.

In the terminology of consistency described in Section 2.5.2, dynamic sets will not return a false positive, but false negatives may arise.

These weak consistency semantics are acceptable for several reasons. One reason is that most of the information for which people search is reference information, which changes slowly. The likelihood that a search will see stale data is small because searches run for a much shorter duration than the typical lifetime of objects. This is especially true for immutable objects like movies, pictures, or published articles. A second reason is that search is not an exact process. It is difficult to express search criteria accurately, and often queries do not capture exactly what the searcher had in mind. This fuzziness means that queries can be satisfied by a best effort approach: many users are more interested in getting useful data in a timely fashion than in having strong guarantees and potentially suffering delays in getting data as a result. Third, communication failures can be common on large scale GDIS like the WWW. One study found that the Internet is in a permanent state of failure since there is always some host that is inaccessible[49]. Consistency is difficult to maintain in this environment; one must either prevent access to data while communication is down, or risk the chance of releasing stale data. As discussed in Section 2.5.2, most GDIS avoid this issue by offering weak or no guarantees on the currency or consistency of their data.

In short, most users are willing to trade consistency guarantees for improved performance or availability. Anecdotal evidence bears this out: search engines like Lycos[51] or the Webcrawler[69] have seen exponential growth in usage even though they offer only bounded currency guarantees. In fact, this tradeoff has been made in many previous system designs: the use of asynchronous disk writes in the Unix Fast File System [53],

the use of write-back caching in Sun's Network File System[77], or the relaxation of Unix semantics to session semantics in AFS[81], to name a few.

Although the weak consistency semantics of dynamic sets is just one of a number of valid design points which trade consistency for performance or availability[96], they are the appropriate semantics for search on DFS or GDIS. Stronger completeness or currency guarantees cannot be ensured in the presence of failures without decreasing data availability. If the client cannot communicate with the server, it cannot detect updates and therefore cannot ensure the currency of the data it possesses. Weaker guarantees are also not appropriate as they would introduce inefficiency. If any object may become a set member regardless of whether or not it satisfied the search, the application must revalidate every member before processing it.

3.1.4 The Application Programming Interface

The dynamic sets interface is designed to satisfy the goals listed in Section 3.1.1 and to provide the properties listed in Section 3.1.3. Complexity arises due to the presence of communication failures in distributed systems, as well as the problem raised by lazy determination of set membership in the face of concurrent updates to objects from other users of the system.

The interface provides set operations that are clearly useful to searchers and that are sufficiently powerful to satisfy the needs of reasonable searches. It is more important that the interface be useful than provably minimal, since the cost of adding a new operation is small. This section describes how each of the dynamic set operations can be used by different kinds of search. The activities that the interface supports consist of creating a set to hold candidate objects, merging these sets, restricting focus to a subset of the objects of interest, and examining the objects in the set.

The operations in the dynamic set interface are presented in Figure 3.1. The table uses the keywords *proc* and *iter* to indicate whether the operation is a procedure or an iterator. An iterator retains state from invocation to invocation, such as the objects it has previously yielded, while procedures do not. The table uses the keyword *returns* to indicate the type of the value returned by the operator. The keyword *none* means that the function in question does not return a value. An iterator *yields* a previously unyielded object each time it is called until all objects have been yielded. The types *bool*, *int*, and *list* are assumed to be the basic boolean, integer, and array types supported by most computer languages. The meta-types *elem*, *digest*, *digestSelector*, and *specification* denote respectively the basic object type, digest type, way of specifying which of the supported digest types to use, and the type of an expression (written in a specification language) that identifies the names of the set's members. The meta-type *set* is the type

of handles to dynamic sets. For purposes of exposition, the operations are broken into classes, and each class is described in one of the following sections.

Category	Operation	Arguments	Results
Allocation	`setOpen`	*proc* (q : specification)	*returns* (t: set)
	`setClose`	*proc* (s: set)	*none*
Management	`setUnion`	*proc* (r : set, s: set)	*returns* (t: set)
	`setIntersect`	*proc* (r : set, s: set)	*returns* (t: set)
	`setRestrict`	*proc* (s : set, q : specification)	*returns* (t: set)
Processing	`setIterate`	*iter* (s: set)	*yields* (e: elem)
	`setDigest`	*iter* (s: set, t: digestSelector)	*yields* (d: digest)
Miscellaneous	`setSize`	*proc* (s: set)	*returns* (i: int)
	`setMember`	*proc* (s: set, e: elem)	*returns* (b: bool)
	`setWeight`	*proc* (s: set, l: list[elem])	*none*

This table presents the operations in the *dynamic sets* interface. To aid the presentation, the operations have been divided into four classes, listed in the table's first column. The keywords *proc* and *iter* identify operations as being procedures or iterators, respectively, and the keyword *returns* identifies the type of object each operation returns. The keyword *none* means the operator does not return any value. An iterator *yields* a previously unyielded object each time it is called until all objects have been yielded. The meta-types *elem*, *digest*, and *specification* are purposely underspecified to give leverage to the implementor. The types *bool*, *int*, and *list* are assumed to be the basic types boolean, integer, and array. Handles for dynamic sets are of type *set*.

Figure 3.1: Dynamic Sets Application Programming Interface

To allow the implementor maximum flexibility in customizing dynamic sets to each target domain, the operations are purposely underspecified where possible. For instance, the exact composition of the set membership specification used to create a set is left undefined, as is the object's type *elem*. *Elem* could consist solely of simple objects (source file, hypertext page, image), or could themselves be dynamic sets or other compound objects. The appropriate choice depends on the implementation and the richness of the type system of the target system. Underspecification also allows the implementor to exploit asynchrony for improved performance and better system adaptability to heterogeneity. For instance, the work of fetching members can be overlapped with the application's processing of other members because the design does not insist that membership be fully

determined when `setOpen()` returns. A third benefit of underspecification is that it allows the implementor to choose types that fit with the existing interface. This results in interface extensions that match the idiom of the original system, which makes dynamic sets easier to program.

The key contribution of this interface is that it hides the complexity of asynchronous I/O behind a simple yet powerful collection of operations. Although asynchrony is very powerful, it adds tremendous complexity to the programming model. With dynamic sets, applications like `grep` can benefit from asynchronous prefetching without the extensive modifications needed to add multiple threads, concurrent execution, mutual exclusion, etc.

3.1.4.1 Allocation Operations

Dynamic sets are created with the `setOpen()` procedure. Callers supply a specification of membership that is used to identify which members belong to the set, and receive a handle for the newly created set in return. All members in the set are guaranteed to satisfy the specification, but not all objects that satisfy the specification are guaranteed to be in the set. Although the design leaves the type *specification* undefined, example specification languages include regular expressions such as Unix's `csh` wildcard notation, and query languages such as SQL's select statement.

An application can disallow further operations on the set and cause the set's resources to be released by closing it when the set is no longer needed. The operation `setClose()` provides this function. After `setClose()` returns, its argument is no longer accessible to this process, and the resources consumed by the set may be released. Note that this does not affect the accessibility of other sets that have been derived from this one. For instance, a searcher may create a set, use `setRestrict()` to create a subset, and then close the original set without losing his ability to process the subset.

3.1.4.2 Management Operations

In order to support multi-stage searches like those described in Section 2.2.3, the interface includes the operations `setUnion()`, `setIntersect()`, and `setRestrict()` which provide the union, intersection, and subset functions of standard set notation. These operations allow searchers to create sets that combine or refine the membership functions of existing sets. The parameters to `setUnion()` and `setIntersect()` are handles to existing sets. The arguments to `setRestrict` are the handle of an existing set and a membership specification whose type is the same as the membership specification passed to `setOpen()`. The result is a new set whose members satisfy both the input set's membership specification and the specification passed to `setRestrict()`, and is guaranteed

to be a subset of the input set. To preserve immutability, these operations return handles for newly created sets, and leave the argument sets unmodified.

Since a set's membership may not be fully determined when the set is created, it may be possible for a new set to be derived from a set whose membership is partially evaluated. How should the new set's membership be determined in this case? Fortunately, answering this question is made easier by two facts. First, objects cannot be removed from the base sets, so an object can be definitely said to be a member of a derived set once its membership in the base sets is known. In the case of a set resulting from `setIntersect()`, an object that is known to be a member of both base sets can be added to the derived set before the base sets' memberships are fully determined, without violating the consistency semantics of sets. Second, sets are not distributed and are stored in memory, which means it is inexpensive to check whether a base set's membership has changed. It is thus a simple matter to poll the base sets to find new members whenever the derived set is accessed.

3.1.4.3 Processing Operations

The ultimate goal of creating a dynamic set is to examine its members, and in particular the data of its members. This is performed with `setIterate()`. Each call to `setIterate()` returns *handles* to one or more members of the set, and each member is yielded (a handle to it returned) at most once. Calling the iterator a sufficient number of times will cause all members to be yielded. Possession of a handle allows the application to perform any non-mutating operation or method that the object supports, and in particular allows the application to read the object's data. In order to return the handle, the system must have obtained the object – either fetching it to the local client, locking it, or in some other way having captured a snapshot of the object's state for the searcher to use. All of these mechanisms have an associated cost. For instance, fetching an object involves substantial latency. However, this cost must be paid if the application needs to read the objects in the set.

If the size of a set is large, it may not be practical to pay this cost. Instead one may wish to restrict one's focus to a smaller subset using meta-information about the members. The second iterator, `setDigest()`, allows an application to access information about a member, such as its name, attributes, or a summary of its data. This collection of an object's meta-information is called its digest, and could be used by an application to generate the parameters to the `setRestrict()` operation to produce a subset which could more reasonably be processed via `setIterate()`.

To be practical, the latency to access an object's digest must be substantially lower than the latency to access the object itself. For instance, the digest of a journal article could be its title and abstract. The time to fetch just this information is often much smaller than the time to fetch the entire article. In addition, one might consider extending the

notion of cost to include more than just time. The journal's publishers might give away the title and abstract for free, but charge money for the article itself.

There are several important implications in the use of a digest. First, the digest must contain some reference to the object itself so that the searcher can identify which objects to contain in the subset based on their digests. Second, the system must be able to obtain a digest (either a static representation or dynamic evaluation of the object such as the method proposed by Fox et al.[25]) cheaply. As such, the types of digests supported by the implementation, *digestType*, depend on the kind of information that is cheaply available. Third, it may be useful for an implementation to provide several different kinds of digests, and allow the application to select which type of digest to return. Thus one can think of a range of possible digests, each more expensive to obtain but providing more information about the object.

One should also note that current systems can obtain useful information for a digest. For instance, all systems can at least return the names of members – the system must possess the name of a member in order to know it is part of a set. In addition, many WWW search engines return some amount of summary information, such as the type, date, title, or the first few lines of text.

3.1.4.4 Miscellaneous Operations

In addition to the operations listed above, there are several operations which may be useful although they are not required. The `setSize()` operation returns the number of members in the set, `setMember()` is a predicate which returns $TRUE$ if the argument `e` is a member of `s`, and `setWeight()` allows users to inform the system of the relative importance of the members.

The semantics of the first two of the operations are complicated by the need to preserve the illusion of immutability in the face of lazy determination of set membership. The `setSize()` operator can only return a lower bound on the set's cardinality until membership has been fully determined. `SetMember()` cannot safely return $FALSE$ until membership is fully determined, although it can return $TRUE$ as soon as the object in question is known to be a member. Although the behavior is not specified here, implementors have two choices: they can have the operation block until a safe answer can be given; or they can add a return value or exception which the system raises when membership has not been fully determined. If this is the case, the answer should be viewed as tentative.

As mentioned previously, searches do not rely on processing elements in any specific order. However, some applications may terminate more quickly if candidate objects are presented in some order known to the searcher. For instance, one member may be much

more likely to satisfy the search than the others, and should be considered first. The `setWeight()` operation allows a searcher to inform the system of the relative importance of the members. The system will endeavor to yield objects in order of their weight, but pragmatic considerations such as communication failures or server load may lead the system to violate the ordering to avoid blocking the searcher until the next ranking member is available.

3.1.5 Examples Using the Dynamic Sets Interface

As an aid to understanding how dynamic sets can be used by searchers, I now revisit the three examples from Section 2.2 and show how one can use dynamic sets in each of these cases.

3.1.5.1 Example 1: Simple Search

The first example was a simple search for variable definitions in a large source tree using grep. Two things must happen in order to have `grep` use dynamic sets. First, the application must be rewritten to iterate over sets. Figure 3.2 shows how this might be accomplished.

Second, the list of file names that is passed to `grep`, `*.c` in the example, should be used as the membership specification when opening the set. The code in the example assumes the specification will be in the second argument, which means the `csh` should not expand the wildcard notation but instead pass it to the application. However, one could imagine more powerful ways of identifying the names of the members, such as file system indexes such as GLIMPSE[50] or programs such as `etags` which was mentioned in Section 2.2.1.

This example points out two advantages of dynamic sets. First, modifying search applications like `grep` to use dynamic sets is simple. These applications effectively iterate over sets of objects, but since current interfaces do not directly support iteration must step through a list instead. In particular, the code that implements `grep`'s functionality (contained in `execute()`) need not be rewritten to use dynamic sets. Second, it is possible to tune an implementation to match the idiom of the system that is being extended to support sets. In Figure 3.2, for example, `setIterate()` returns an open file descriptor which can then be used by `execute()` without requiring further action.

3.1.5.2 Example 2: Using Search Engines

The second example involved search using WWW search engines to identify a set of candidate objects, those objects containing some keywords supplied by the searcher (e.g.

Main loop of `grep`	Main loop using dynamic sets

while (*argv) {
  fd = open(argv++);
  execute(fd);
  close(fd);
}
	s = setOpen(argv[2]);
	while (fd = setIterate(s)) {
	  execute(fd);
	  close(fd);
	}
	setClose(s);

The two sections of code reflect how `grep` can be modified to use dynamic sets. The code on the left is the main loop of `grep`. `Grep` takes a list of names as input, opens each file in the list, processes its data with the `execute()` procedure, and closes it. The code on the right shows the main loop of `grep` using dynamic sets. `Grep` first opens the set, using the input as the specification of membership. It then processes every member yielded by the iterator. There are two key points of this example. First, it shows the ease with which one can modify common search applications to use dynamic sets. Second, the main functionality of `grep`, locating substrings in the set of files, does not need to be modified to use dynamic sets.

Figure 3.2: Code Example Showing the Use of Dynamic Sets

"cow"). In many respects this example is similar to the previous one, the chief difference being use of query evaluation to identify members rather than supplying an explicit list of member names. For instance, one could imagine a system in which it was legal and sensible to run `grep varname /coda-db/\select filename where defines like "varname"\`, where the pathname is an SQL query to a database (coda-db) of symbolic links which identifies those source files that define variables similar to the word "varname". With a suitably defined specification language (the *specification* type), this example is not only feasible but is legal in the implementation described later in the document.

Use of a search engines raises other interesting questions. First, most search engine results include object rankings which are the engine's estimation of how closely the object matched the query. Since a set is unordered, how would one utilize or preserve this information while iterating? There are two possible approaches than an implementor can take. First, although the set has no inherent order, the system does have to fetch the objects in some order and could take the rankings into account when determining this order. Second, the application could use the rankings (possibly obtained from the digests) to assign weight to the members via the `setWeight()` operation.

The second question is "What happens if the searcher does not examine all the members of the result set?" Current WWW browsers present the result set as a list of objects, and users choose which of the members to examine and in what order. One does not have

this control when iterating over a set. It may be the case that this sort of control is not necessary. If so, the system is free to prefetch as much of the set as it deems appropriate, which is important if bandwidth is limited. If not, the system could fetch all of the set, and allow the searcher to determine which of the objects have been fetched at any given point. For instance, the browser could obtain this information through `setDigest()` or `setMember()` and use a different color to identify links to prefetched objects.

3.1.5.3 Example 3: Multi-Stage Search

The third example involves multi-stage search, in which a user creates a number of sets and then merges them into a single set for perusal. The dynamic sets API provides three mechanisms for manipulating the membership of a set, allowing applications to create the union or the intersection of two sets as well as the subset of a set.

Union is useful for merging result sets from search engines that cover non-overlapping portions of the system's name space. For instance, many large sites provide an engine that only indexes that site's objects. Intersection is useful to merge sets from overlapping search engines. For instance, the objects that are indexed by two different search engines may be more likely to satisfy a search than those that are only indexed by one. As an example, Altavista allows one to search either posts to Usenet bulletin boards or the WWW. One may wish to combine a search of both forums for a particular piece of information.

One might also wish to create a subset, for instance if the current set is too large to fetch or contains spurious members. One would first identify the names (and possibly other information) of the members using `setDigest()`, and then use this information to create a specification of which members to add to the subset. As an example, one could select all the members that are WWW pages with the specification "`*ht{m,l}`".

3.2 Beneficial Properties of Dynamic Sets

The previous section outlined the design of the dynamic sets abstraction. This section describes the benefits of dynamic sets for search, and how these benefits can be achieved. These benefits fall into three categories: general properties, functionality enhancements, and performance improvements. Each is discussed in the following subsections.

3.2.1 Dynamic Sets Are General, Simple, and Easy to Program.

One benefit of dynamic sets is their generality. Dynamic sets, although tuned for search applications, are applicable to a wide range of domains. Section 3.3 sketches how dynamic sets can provide benefit to several such domains. Further, dynamic sets can be used on a variety of systems. This dissertation explores the use of sets in three different systems: the WWW, NFS, and the local file system.

A second benefit of dynamic sets is their simplicity, both for programmers and for users of dynamic sets applications. The concept of a dynamic grouping abstraction such as dynamic sets is a natural addition to the interface of a distributed system. Their ease of use comes from the choice of sets as the underlying structure. Sets are a basic concept in mathematics, commonly taught to elementary school children. Thus is it natural for users to think of groups of objects as sets. For instance, the `csh` wildcard notation is derived from regular expressions, a mathematical language used to describe membership in sets.

The dynamic sets API is also easy to program. The basic operations are similar in concept to either standard file system operations such as `open()` and `close()`, or standard set operations such as `union()`. Although existing applications must be rewritten to work with dynamic sets, doing so for many common applications is a trivial exercise. Figure 3.2 gives an example of this by showing how the source code for `grep` is modified to use dynamic sets.

3.2.2 Dynamic Sets Improve Functionality

Dynamic sets improve the functionality of the system's API by supporting iteration over groups of objects and execution of queries. It should be noted that the concepts underlying dynamic sets are not new to this dissertation; high level programming languages provide support for sets and iterators, and database query languages such as SQL use a similar construct called a *cursor* to hold the results of queries. However, the idea of providing system support for them is new, as is leveraging their use to reduce latency.

Iteration is important for two reasons. First, it simplifies the job of the programmer, since the system maintains the state necessary to track what has and has not been processed. A related benefit of iterators is that they leave the order of processing the objects to the system, which allows the system greater flexibility in fetching and presenting the set members to the application. A second benefit of iteration is that it provides an easy way to add support for breadth first search to the interface of WWW browsers. Current browsers only allow search through depth first search with backtracking. For example,

suppose a user has queried a search engine and is perusing the resulting `HTML` page. The user starts by clicking on the first link in the page, examining it, and possibly moving on to its children. When reaching a dead end (when there are no more interesting children to explore), the user backtracks by moving back up the list of visited pages until he is back at the query results page, and then explores a different branch by clicking on another link. Because cached information may be lost while searching, backtracking occasionally suffers I/O delays by causing objects to be reloaded. If one modified the browser's user interface to expose sets and iteration to the user, the user could use the iterator to keep track of his progress in processing the set. When the dead end is reached, the user can call the iterator to get the next branch instead of having to backtrack up the tree, saving effort by the user and avoiding the delay of reloading the objects.

Dynamic sets also improve functionality by providing direct support for queries. This means systems that support dynamic sets can be trivially extended to integrate associative naming into environments in which it was not available before. For instance, current Unix systems allow applications to specify sets of objects using syntactic pattern matching. For example, the string "`*.c`" gets expanded to be the set of objects in the current directory whose names end in "`.c`". By supporting queries, dynamic sets allow a set's membership to be specified by association. Thus an application could request a set of objects containing the word "cow", or a set of objects created last week, etc. Of course, the ability to create such sets depends on the power of the implementation's query language in which these specifications are stated.

3.2.3 Dynamic Sets Improve Performance

Dynamic sets improve the performance of search applications by reducing the latency of accessing remote objects. The benefit comes from the disclosure of hints about an application's future data needs to the system. These hints can be used in a number of ways, four of which are discussed here.

Although all four are interesting, this dissertation focuses on only the first two, informed prefetching and reordering of requests, because they offer substantial improvements and satisfy the goals of dynamic sets listed in Section 3.1.1. The latter two mechanisms, batching and function shipping, require modifications to protocols and servers, and thus violate one of the goals of the design.

3.2.3.1 Informed Prefetching

Membership in a dynamic set is a implicit hint of future access. When a set is created, the system can use these hints to prefetch the set members assuming sufficient resources

are available. Prefetching set members yields three benefits. First, it allows the system to use available parallelism between independent components. For instance, fetching from n independent servers can be done in parallel, reducing the aggregate latency by up to a factor of n.

Second, the time to fetch a member can be overlapped with application processing of other members. Although this overlap can only produce a factor of 2 increase in the application's elapsed run time, it can substantially reduce the amount of time the application waits for data. For interactive applications, reducing wait time is more important than reducing overall elapsed time. For example, dynamic sets can eliminate almost all of the latency of remote access in the WWW by fetching remote members while the user is reading other members.

Third, informed prefetching can result in improved disk, network, and server utilization. By prefetching aggressively, the system can keep request queues filled so that the next request can be processed as soon as a device or server is finished with the current one. By filling these queues with prefetch activity, the dynamic sets abstraction also introduces an opportunity to reorder the queues to provide lower aggregate response times. For instance, one could sort the requests in a disk queue by proximity to the disk head to reduce the average seek times seen by the requests. Current resource schedulers have little chance to optimize utilization because request queues are usually empty or very small[73].

3.2.3.2 Reordering Requests

A unique property of dynamic sets is that their members are unordered. Because of this, the system is free to choose the order in which the iterator yields members. Although it seems simple, this benefit has far-reaching consequences for the system.

The first consequence is that it allows the system to take advantage of cache state to reduce latency. Suppose some members of a set are cached and others are not. If the system had to obey an external ordering, it might delay the application in order to fetch a remote object. Dynamic sets, allow the iterator to yield the local objects to the application immediately, giving itself time to prefetch the remote data so that the application sees no I/O stalls. Dynamic sets can also yield the members that are in the cache before they are evicted. Without this ability to reorder, cached members may be evicted before they can be processed adding additional I/O stalls to refetch the evicted objects.

Reordering can also be used to reduce the amount the aggregate latency even when none of the members are cached. The chief benefit of reordering can be seen in the access to the first member. Since the application is likely to start iterating soon after opening the

set, there is little or no opportunity to hide the first fetch behind application processing. Thus most or all of the latency of the first fetch results in a stall of the application. However, dynamic sets can reduce this stall by fetching objects with low response times. For instance, the system could fetch small objects if the network bandwidth is limited, or fetch objects from servers that are close to the client. This ability to determine the order in which members are yielded based of response time is especially important in the presence of communication failures, as discussed in Section 3.1.3.3.

Similarly, dynamic sets can reorder the requests to improve the amount of concurrency in fetching the members. Consider the possibility that a set contains 10 members stored on three servers, and that the first four members are on the first server, the next three on the second, and the final three on the third server. Fetching them in order eliminates the chance for concurrent execution of the fetches. By reordering the fetches, however, the members can be fetched from the three servers in parallel, greatly reducing the aggregate time to access the objects. Determining the location of an object in order to reorder requests adds little overhead since the client already needs to know an object's server in order to fetch it.

A key question is how difficult it will be for the system to determine an order. Fortunately, there is a range of possible solutions. A simple strategy is to start N fetches and yield the one that returns first to the application. More complicated strategies may provide more optimal schedules, but require more computation or I/O. In some cases, a more optimal schedule may be too expensive to compute if the cost of gathering the data needed to schedule the fetches is nearly as expensive as the fetch itself. For instance, a client determines an object's size in the WWW's `HTTP` protocol by examining the header of the response to a fetch request. For small to medium sized objects, this is just as expensive as fetching the object. Although the brute force technique may perform poorly in the worst case, in practice it should prove reasonable and does have the advantage of a low overhead.

In general it is difficult to know the probability that a particular member will be accessed. But since a system that supports dynamic sets controls the order in which objects are returned, the system can assign the likelihood of access to objects instead of estimating it. The highest probability that can be assigned is the probability that the application will call the iterator at least one more time. Similarly the second highest probability is that the application will call the iterator at least twice. Rather than recalculate this probability at every step, this logic can be used to limit the number of wasted fetches. For instance, the client can fetch any 10 members knowing that they will be the first 10 members accessed. At worst, those 10 members will be wasted only if the application does not call the iterator again.

3.2.3.3 Batching Requests

Although not explored in this dissertation, one could use hints and the ability to reorder to batch requests to the same server. Batching sends a single packet which contains many requests, and receives many objects in return. For instance, one could ask for five objects stored on a server with a single request. The advantage to batching is that it amortizes the latency of remote access over several operations.

Batching has three benefits in addition to the reduction of propagation delay. First, batching reduces the overhead of packet setup and request processing. Since server CPU is a precious resource in distributed systems[34], batching can increase the system's scalability.

Second, networks are designed with a maximum packet size, and so the cost of transmitting data is a step function over the amount of data, with steps located at multiples of the packet size. If a current response is 1000 bytes and the packet size is 4KB, 4 responses could be packaged together with only a small overhead for transmitting the extra bytes.

Third, transfers of large data can be improved by streaming protocols, such as those used by TCP[35] or SFTP[79]. Streaming protocols work by allowing a single response to acknowledge multiple packets, lowering the cost of reliable delivery of data. Batching can utilize streaming protocols because it ships more data in a single exchange.

3.2.3.4 Function Shipping

The best way to reduce the latency of communication is to eliminate or reduce the amount of communication required by an application. One way to reduce bandwidth requirements is to employ function shipping: shipping (a portion of) the application to the data instead of the data to the application. Function shipping is naturally suited to multi-stage search. For instance, imagine searching for videos of cattle from a library of thousands of videos. In a data shipping model, the client fetches the videos and examines them to determine which if any contained pictures of cows. Potentially all of these large objects need to be transferred to the client to be examined. In a function shipping model, the client first ships an image recognition program to the server (or a proxy located near the server) which identifies those images or videos of interest to this search. If this program is sufficiently powerful, it can perform complex analysis of the data to reduce it to a much smaller subset of the available videos. The client can then fetch a substantially smaller amount of data, reducing the elapsed time of the search as well as the consumption of network bandwidth.

In some sense, function shipping is a dynamic and more general form of querying a search engine to determine a set of candidate objects. Thus dynamic sets are an ideal abstraction

to support function shipping. A set can be created to hold the objects returned by a remote function. The members can either be immediately fetched, or the set could be merged with sets from other remote functions. A client could ship a function to all the servers storing members in the set by using an operator similar to `apply` in Lisp[90]. Although the dynamic set API does not have such an operation currently, it would not be difficult to add it.

One of the difficulties of function shipping is that executing the function remotely poses security and administrative problems. Security issues arise if the function is powerful enough to affect state on the remote server, such as removing files or communicating over the network. Recent developments in Internet software such as the Java[2] or ActiveX[3] programming languages partially address these technical problems, but other problems remain. For instance, the remotely executing function can consume resources such as server CPU cycles, but the infrastructure needed to ensure suitable recompense to the server's owners does not currently exist.

3.3 Other Application Domains for Sets

Although dynamic sets have been designed and tuned to support search applications, their benefit is not limited to search applications alone. This section discusses how dynamic sets could be used in a variety of application domains. However, exploration of the benefits of dynamic sets in these domains is left as a future exercise.

In general, dynamic sets should benefit any application that processes a group of objects, whose performance is dominated by I/O latency, and is satisfied with the weak consistency model provided by dynamic sets. In addition, prefetching will only offer performance improvements if it can increase the utilization of some I/O device such as by overlapping I/O and computation.

3.3.1 Data Mining

Data mining is the process of examining data from multiple information sources to extract or infer trends. Data mining consists of running queries on information systems, such as databases, and then applying various heuristic filters on the resulting sets. These filters often combine elements from different sets to find relationships between data that may not be expressible in the query languages of the information sources.

[2]Java is a trademark of Sun Microsystems, Inc.

[3]ActiveX is a trademark of Microsoft Corporation.

Data mining provides an ideal application domain for sets. Dynamic sets could be used to hold the results of queries or as the intermediate representation for collections of candidate objects. Filters could be written to iterate over these sets. The lack of an explicit order in dynamic sets is also appropriate for this domain, as data miners could create sets to hold objects of equal importance since the overhead of creating sets is low.

3.3.2 Multimedia Object Repositories

Digital libraries and multimedia object repositories are another potential domain in which to use dynamic sets. These systems consist of servers which store and index a vast amount of information, such as documents, images, videos, audio recordings, etc. This multimedia data is stored as objects and is accessed through the object's name, much like files in a file system. Unlike a typical file system, however, associative naming is built into the system. It is thus a conceptually simple extension to add dynamic sets to the interface of one of these systems.

An example of this type of repository is the QBIC image database[61]. QBIC allows images to be indexed by their content, such as color, texture, shape, the kinds of pictures they contain, etc. as opposed to textual descriptions of the pictures. QBIC stores the images as file system objects. Thus a query returns the names of the files containing the desired images, and the system then returns the files themselves to the client for processing on demand. One can imagine modifying QBIC to use SETS: forming a dynamic set using the names returned from the query, and having SETS prefetch the files on behalf of the application.

3.3.3 Searching Archival Information

Many sites archive information on removable media, such as tape libraries or CD juke-boxes. These media provide high bandwidth access to data but extremely high latencies, particularly when the tape is offline when the request is made. However, if the system had knowledge of multiple requests, it could reorder them to take advantage of batching, amortizing the cost of mounting a tape or CD over access to multiple objects on that volume. In addition, some archives have multiple drives, and so can satisfy some number of concurrent requests to increase throughput. Thus there is opportunity to increase the utilization of these drives in order to reduce the latency seen by an application.

Dynamic sets provide a convenient interface to applications using these libraries. Search applications could use sets directly, or requests for off-line data could be spooled, then expressed in terms of sets where each set contained objects with equivalent priorities. With more knowledge of future requests that dynamic sets provide, the system could

schedule requests to amortize the time to mount a tape over multiple requests for that tape's data, and could access tapes in parallel using the library's multiple drives.

3.4 Summary

This chapter defined the dynamic sets abstract data type. Dynamic sets are lightweight, temporary, and unordered collections of objects whose membership is determined on demand. In addition to motivating and describing these properties, this chapter also presented the rationale for the design, and the operations used to manipulate and process dynamic sets. This chapter also listed the benefits of dynamic sets and potential application domains in which dynamic sets could be employed.

Chapter 4

SETS: An Implementation of Dynamic Sets

This chapter describes SETS, an implementation of the dynamic sets abstraction described in the previous chapter. SETS extends the environment used by the Odyssey project[80] at Carnegie Mellon University by adding support for dynamic sets to the file system interface. Section 4.1 describes the context in which SETS is intended to run. Section 4.2 describes how the interface for dynamic sets is instantiated in SETS. Section 4.3 discusses the architecture and detailed implementation of SETS. It also describes how the structure of SETS is designed to increase the autonomy of the various components of SETS, in order to allow maximal asynchrony in processing a set of objects.

4.1 Context and Background for SETS

Because the dissertation is concerned with improving the performance of typical users of a distributed system, the platform of interest is that of a typical client computer. Clients are usually personal computers or workstations, possibly mobile, connected via a range of communication media. For instance a GDIS client may be a home personal computer dialed into an online service, or a workstation connected via Ethernet. Low end machines have limited memory and processing power, for instance a common configuration today is a 66MHz Intel 486PC with 8 or 16MB of memory. More powerful workstations have 64MB or more of memory and hundreds of MIPS of processing power.

Although these computers run a range of operating systems, this dissertation will use a variant of BSD 4.3 Unix[47]. The reason for this decision is that BSD provides a common set of functionality such as virtual memory, multitasking, and support for multiple file systems. In addition, the source code to BSD is publically available. Commercial

personal operating systems such as OS/2, Windows, or MacOS support more applications and are more widely used, but do not have source code in the public domain. The variant of BSD used at CMU is Mach 2.6[1]. Mach extends the core functionality of BSD with advanced operating system features like kernel threads, external pager control, and modern interprocess communication.

For the purposes of this thesis, however, Mach 2.6 can be considered as equivalent to BSD 4.3. Throughout the implementation and design of SETS, I have studiously avoided any Mach functionality that is not provided by other implementations of BSD Unix. As a result, the basic SETS functionality should be portable between unix-like operating systems. Due to limited manpower resources, however, verification of these claims of portability must be left as future enhancements.

SETS is one component of Odyssey: a distributed data repository to explore the issues of application aware adaptation in the context of mobile computing and distributed systems[80]. Odyssey extends the notion of DFS to include support for multiple types of objects. The Odyssey client subsystem is decomposed into generic and type-specific components. The type-specific functionality for a particular type of object, such as a file or a video stream, is encapsulated in a component called a *warden*.

4.1.1 Background: BSD

This section provides some background for readers that are unfamiliar with the details of BSD. BSD is a kernel which runs in protected mode and provides services to user-level processes through operations called *system calls*. System calls are similar to procedures in functionality, but have a substantially higher cost since they cross the protection domain between the user level process and the kernel.

The component of BSD of most interest to this dissertation is the file system. The file system provides access to persistent objects called files. A file is an untyped object of varying size, which is accessed through a subset of the system calls. These file system calls take the name of or a reference to a file on which to perform the operation, and resolve the name to obtain a low-level file handle called a *vnode*. File names in BSD are hierarchical; files are the leaves of the hierarchy's tree.

For SETS, the most important of the file system operations are `open()`, `close()`, and `read()`. `Open()` takes the name of a file and returns an *open file descriptor* which is used as a handle for the open file. `Read()` copies data from the file into a buffer supplied by the caller. A side effect of `read()` is that the *file pointer* associated with the open file is incremented by the amount of data that was read, so that successive reads will see successive portions of the file. When the process is finished with the file, it can `close()`

it to release the descriptor and tell the system it has finished processing the file. A process's open files are closed automatically when the process exits.

Two other system calls of interest to SETS are the `fork()` and `dup()` operations. `Fork()` creates a new process based on the calling process. In particular, the new process inherits (and can share) the file descriptors of the old process. `Dup()` creates a new descriptor for the same file. The new descriptor shares the file pointer with the old descriptor, so that a file could be read sequentially by alternating reads to the old and new descriptors.

In BSD, a file system's name space can be segmented into different (sub-)file systems[1]. A new file system is added to the name space by *mounting* it at some point in the existing hierarchy. BSD supports multiple file system types through the VFS switch[42]. Under VFS, each file has a type which identifies the VFS driver responsible for that file. Examples of file system types are distributed file systems such as NFS[77] and Coda[82, 41], and local file systems such as the Fast File System[53] and the Log-Structured File System (LFS)[72].

The VFS driver acts as the interface between the file system API and the DFS client subsystem. In some cases, the DFS client subsystem and its VFS driver are closely intertwined. This is usually true when the client subsystem is part of the operating system kernel, as is NFS and AFS. For others, the driver and client subsystem are distinct and communicate through a well defined interface, possibly over a protection boundary. Coda's client cache manager, *Venus*, is an example of this approach, and makes extensive use of name and attribute caching in the driver to reduce the cost of communicating between the driver in the kernel and Venus in a user-level process.

The actions taken by the driver in response to an `open()` or `read()` operation on a file depend on the semantics of the file system responsible for the file. To handle an `open()` in Coda, Venus first resolves the filename, fetching uncached directories on the path from the root to the file. It then fetches the entire file if it is uncached, writing it to a local disk file called the *container file* for that object. Reads to the open file are directed to the container file without contacting Venus, and so essentially have the same cost as reads to a local file. In NFS, `open()` performs name resolution, but does not fetch the file's data. Instead, the data is fetched in blocks as a result of the `read()` operation and is cached in the client's memory. Thus in Coda, an open can be an expensive operation since it incurs most of the I/O latency that will be seen by the application, namely that to fetch the whole file from the server if the file is not in the cache. In NFS, however, reads typically take longer than opens, since they often involve contacting the server and fetching a block of data.

[1] Unfortunately the same term is used to denote both the entire name space seen at a computer and components of it.

4.2 Adding Dynamic Sets to the BSD API

The previous chapter described dynamic sets in an abstract context. This section describes the detailed design of adding dynamic sets to BSD. Section 4.2.1 discusses the instantiation of the types used in the design, and argues that the type decisions are appropriate for BSD. Section 4.2.2 revisits the operations in the dynamic sets API to describe changes to the operations made for SETS. The `man` pages in Appendix A contain detailed descriptions of the operations and the use of the parameters.

4.2.1 Instantiating Dynamic Sets in BSD

Category	Operation
Allocation	`int setOpen( char *name, int flag );` `int setClose( int s );`

Management	`int setUnion( int r, int s, int flag );`
	`int setIntersect( int r, int s, int flag );`
	`int setRestrict( int s, char *rname, int flag );`

Processing	`int setIterate( int s, int f, char *b, int size );` `int setDigest( int s, char *buffer, int size );`

Miscellaneous	`int setSize( int s );`
	`int setMember( int s, char *elem );`
	`int setWeight( int s, int *weights );`

This table presents the type signatures of the operations in the SETS API. The operations are those listed in Figure 3.1, modified to use types more appropriate to BSD, the environment in which SETS is implemented. Other changes include adding parameters to some of the operations to allow applications to tune the behavior of an operation to suit their needs. `Man` pages describing these operations are presented in Appendix A.

Figure 4.1: SETS Application Programming Interface

Figure 4.1 lists the SETS instantiation of the type signatures of the dynamic set operations listed in Figure 3.1, replacing the abstract data types with the types chosen for the BSD environment. The previous chapter used, but did not define the type signatures of

the *elem*, *specification*, *summary*, and *set* types. This is in fact a strength of the design, for it allows this implementation to employ BSD's idiom. As a result, the SETS API fits naturally with existing BSD operations, and can use existing types such as file descriptors more easily.

The definition of these types for BSD is necessarily vague because BSD does not support a strong type interface. Thus, for example, all strings are *char **, regardless of how they are used. Since providing a richer type system is beyond the scope of this thesis, the descriptions that follow describe how a type is used rather than supplying a more formal type definition. Another limitation of BSD's type system is that type errors cannot be detected statically; SETS must check that the arguments are valid on use, and produce errors in response to illegal values. The definition of errors and the manner in which they are reported is described in Section 4.2.2.1.

4.2.1.1 The Basic Object Type: *elem*

The members of a set are the basic data objects in the information system. In BSD, data objects are files. Files can be referenced in one of two ways in BSD. The object itself is identified by its *file name*, a string which identifies the path from the root to the file in the name hierarchy. An object's data can only be accessed by *opening* the file, the open file is represented by an *open file descriptor*. In BSD, the open file descriptor is an integer which is a reference to the open file.

Dynamic set operations have 2 different kinds of references to files. First, operations that access the file's data use the open file descriptor to refer to the file. For instance, `setIterate()` operation returns the file descriptor of the member it is yielding. With the descriptor, the application can access the file's data using the `read()` operation. Second, operations can refer to the file itself by its name. For example, the argument to the operation `setMember()` is a file name. However, since Unix does not support a filename type, the type signature used in Figure 4.1 is simply that of a character string (*char **). In addition to these 2 kinds of references to files, the implementation of SETS stores the file's vnode in its internal data structures. This allows fast access to the file's meta-data. However, vnodes are not exposed to applications through the SETS API.

4.2.1.2 Membership Specification: *specification*

When a set is created, the creator supplies a *specification* which is evaluated by SETS to determine which objects should be members of the set. The result of evaluating the *specification* is a list of file names. This approach has two advantages. First, the use of file names to identify members means that SETS can be added to a system without requiring the modification or addition of a new name scheme. Second, name resolution

Explicit:	`/projects/*src*/*.c`
Interpreted:	`/staff/\select home where name like "%david%"\`
Executable:	`/sources/pkgs/contrib/%myMakeDepend foo.c%`

This figure gives examples of the three different kinds of membership specifications supported by SETS. *Explicit* specifications list the names using `csh`'s regular expressions. *Interpreted* specifications allow applications to use strings which are interpreted by search engines as queries, returning the names of the objects that satisfy the query. *Executable* specifications name binaries whose execution results in a list of names. With these types of specifications, SETS can easily be extended to support a variety of query languages and modes of search.

Figure 4.2: Examples of the Three Types of Names Supported by SETS

of the members can be done asynchronously, allowing the cost of fetching the directories involved in resolving the name to be overlapped with application computation or fetching other members of the set.

The *specification* type consists of strings from a *specification language*. The language must have the following three properties. First, it must be intuitive in order to be useful for interactive searches, since interactive search applications might expose set creation, and thus the need to specify membership, to users. If specifying membership is too difficult, the user is likely to access the objects directly instead of creating a set. Second, the language should be concise, because long specification strings are unwieldy. Further, BSD has limits on the length of valid file names and path names, which limits the length of a membership specification. Third, there are many different sources or tools with which a user can identify interesting objects. In order to support new or different types of queries, SETS specification language must be easily extensible to add support for new types of search engines, query languages, etc.

Because many BSD users are familiar with `csh`'s wildcard set notation[38], SETS extends this notation and uses it for its specification language. In BSD, a file name consists of a sequence of *components*, each of which identifies one node on the path from the root of the file system to the file being named. The `csh` notation extends file names by allowing a component to contain regular expressions which identify several nodes at each stage in the path. For instance, `{src1,src2}/*.[ch]` would be the set of all files ending in "`.c`" or "`.h`" in the subdirectories `src1` or `src2`.

SETS extends the `csh's` set notation to allow a component to contain three kinds of set specifications; examples of each are given in Figure 4.2. *Explicit* specifications use standard `csh` wildcard notation to indicate the names of the members of the set. This type of specification is syntactically equivalent to the `csh` notation and should be easy for Unix users to employ. One difference is that explicit specifications are expanded

by SETS and not by the shell. Although explicit specifications may be quite long, these specifications are much smaller than the limit on name length imposed by BSD in practice. If necessary, one can shorten these specifications by more aggressive use of wildcards.

The second kind of specification is the *interpreted specification*. Interpreted specifications contain strings in some query language, such as SQL, delimited by "\". The portion of the specification to the left of the query identifies an object with which the query is evaluated. Query evaluation is performed by the *warden* responsible for the object. In SETS, wardens are VFS drivers extended to support queries and prefetching. For instance, SETS provides an SQL warden which mounts relational databases as files in the file system name space. A valid interpreted specification would contain the name of such an object such as "`/staff`" in Figure 4.2 and an SQL query like "`select home where name like "%david%"`". Assuming the `home` field contains the names of directories, the result of evaluating this SQL query is a list of home directories of people whose name contains the string "david". The names that result from query evaluation are assumed to contain no further set specifications. Illegal query strings or the invocation of a query on an object whose warden does not support queries result in the empty set.

The third kind of specification is the *executable specification*. Executable specifications are programs that act as predicates over a portion of the system's name space. The program and its arguments are delimited by "%", the prefix of the specification before the first "%" is the directory in which to execute the program. SETS executes the program in a separate, dynamically created process. The program returns to SETS the names of the objects that satisfied its predicate. One interesting application for executable specifications is the evaluation of dynamically generated code. A search tool could generate specialized code for a particular search, and supply the name of the file containing the newly generated code in the executable specification. The result would be a search filter which was tuned for the current application's needs.

As with `csh` wildcard notation, any component of a name can contain a set specification. As an example, consider the specification `/project/*/src/%myMakeDepend *.c%`. SETS begins by parsing a specification from left to right, although the order of the members is undefined in keeping with the no-order property of dynamic sets. Components with no specification characters uniquely identify the next file or directory on which to continue expansion. Components containing set specifications cause SETS to evaluate the specification to produce a list of names. Each name either identifies a file (if there are no more components to evaluate) or a directory which is selected as the current directory for the evaluation of the next component. An error at any point (such as the set creator does not have access rights on the directory or the next component does not exist in the current directory) means that this portion of the name is invalid. If all paths terminate in error, the result of expansion will be the empty set. From the example above, SETS would run the program `myMakeDepend *.c` for each directory that matches `/project/*/src/`.

4.2.1.3 The Digest Type: *summary*

One mode of processing a set's members is to examine a digest, or summary of each member, instead of examining the members directly. For instance, if there are too many members to fetch, or if there is a significant cost associated with fetching them, an application may wish to view the attributes of the members instead. The *summary* type is used in the design to denote the kinds of digests that are supported.

Ideally, applications could supply dynamic filters which would create summaries from the members, and SETS would ship these filters to the server to avoid the cost of fetching the members. In addition, dynamic sets could support a rich selection of summary types; calls to `setDigest()` would indicate which kind of summary to use. However, these features are expensive to provide and unnecessary to evaluate the thesis. User supplied filters are expensive because of the complexity involved in supporting a function shipping framework. In addition, many summary types require nearly as much work as fetching the object. For instance, if the summary is the date of an object, SETS must send a request to the server which can involve most of the latency of fetching the object; thus little is saved by using `setDigest()`.

Instead, SETS currently provides only one form of summary: the object's name. The name can be provided with no overhead, since SETS needs the name to fetch members, and stores the members' names as part of the set. In addition, the name is often useful in discerning between members. For instance, the location of an object, and thus an estimate of the time to fetch it, can be inferred from the names of WWW objects. Similarly, the pathname of a source file can indicate the program or library to which that file belongs. Extending the interface to support a richer collection of summary types is left as future work.

4.2.1.4 The Open Set Type: *set*

The final type binding is that of *set*, which is the handle or reference to an open set. Open set handles are similar in nature to open file descriptors in BSD. Both are created by opening an object, deallocated by closing the open object, and used to manipulate the open object. In addition, an open set is a temporary object created exclusively for a single application. Like open files, an open set can be inherited by the children of a process through the `fork()` system call, and processing of a set can be done by threads within a process or by a process and its children. An open set handle, like an open file descriptor, is an integer whose value is an offset into a per-process table containing pointers to open sets. Although they are similar in nature, open files and open sets are different objects, and the set of operations that manipulate open files do not overlap with the operations that manipulate sets.

4.2.2 Revisiting the Dynamic Sets API

Implementing SETS in the context of BSD required modifying the dynamic sets API. One reason for the change is to make the operations look more like typical Unix operations, such as to overload return codes to indicate error conditions. Another reason is to allow applications to tune the behavior of certain operations to better suit their needs. For instance, `setOpen()` accepts flags with which an application can tell SETS that it plans on using `setIterate()` to process the set. This knowledge allows SETS to more aggressively initiate prefetches. The following sections describe the changes in more detail.

4.2.2.1 Returning Error Codes

In BSD, operations indicate error conditions by returning a special value (usually -1), and setting the global variable `errno` to contain a numeric code indicating the error. SETS conforms to this practice by having all operations return integers, and by mapping SETS errors into standard error codes. For instance, if `setSize()` is called with illegal value for the `set` parameter, it returns -1 and sets `errno` to `EBADF`.[2]

4.2.2.2 Tuning the Behavior of SETS Operations

In the detailed design of SETS, there are several points at which a decision concerning the appropriate semantics needs to be made. Sometimes choosing a single behavior is overly restrictive yet providing multiple behaviors is not difficult. In these cases, it is meet to leave the decision to the application by letting it specify the behavior it chooses.

In order to work within BSD's idiom, SETS operations provide a flags parameter which allows this tuning of behaviors. A caller can set certain bits in this flag to request alternate behaviors. One example of the need for this flag is the `setOpen()` operation. For instance, the application can specify how a set should be shared with children processes, in a similar manner to the way an application can specify how to share an open file. In addition, the application can inform SETS that it should begin prefetching the set's members as soon as possible, rather than using the default policy of not prefetching until the application begins to iterate.

4.2.2.3 Partial Evaluation

As described in Section 3.1.4.4, set membership may be lazily evaluated. This raises a problem for the `setSize()` and `setMember()` operations. Rather than blocking these

[2]Other versions of Unix use different but equivalent mechanisms. For instance, a negative return value in Linux indicates an error, and the absolute value of the number is the error code.

operations until membership has been fully determined, SETS returns a special error code along with a valid result. For instance, if a set has 9 members currently, but membership is not fully expanded, `setSize` will return 9, but set `errno` to contain the special error code `PARTIAL_RESULT`.

4.2.2.4 Batching Summaries

Although `setDigest()` is an iterator, it is inefficient and unnecessary to return only one summary per invocation. On one hand, the cost of getting a summary is largely the cost of making the invocation. Summaries are relatively small and very inexpensive to get since a summary is just the name of a member. On the other hand, the value of a digest is that it gives sufficient information to guide the search with minimal overhead. Thus, the operation `setDigest()` in SETS allows multiple summaries to be returned by a single invocation.

Callers to `setDigest()` supply a buffer and its size. SETS treats the buffer as an array of variably sized structures. Each structure has two fields: an integer containing the offset of the next structure in the buffer, and a string holding the member's name. The last structure in the buffer has zero for an offset value since there is no next structure. The value returned by `setDigest()` is the number of bytes that have been written into the buffer. SETS remembers which names have been yielded by `setDigest()`, and so a name cannot be yielded more than once. `SetDigest()` returns a zero value to indicate that all names have been yielded.

If the set's membership has not yet been fully determined, the last entry `setDigest` puts in the buffer is the special string "...". If there are no unyielded names and the set's membership has not yet been fully expanded, `setDigest()` puts one structure in the buffer whose offset field value is zero and whose name field value is the string "...".

4.2.2.5 Identifying Yielded Members

One problem I discovered when modifying interactive search tools to use dynamic sets is that there is no way in BSD to determine the name of an object given a file descriptor for it. One alternative is to have `setIterate()` return the name of the member instead of an open descriptor. However, this would require applications to open the file themselves in addition to calling the iterator. Since not all applications need the member's name, it is more efficient to have `setIterate()` return the file descriptor, and those applications that want the name can ask for it.

The arguments `b` and `size` serve this purpose for `setIterate()`. If `b` is nonzero, it must point to a buffer whose size is contained in the `size` parameter. When `setIterate()`

is called with a nonzero value for `b`, it copies the name of the member it is yielding into `b`. The application can then display the name to the user along with the contents of the file.

4.2.2.6 Adding Weights to Members

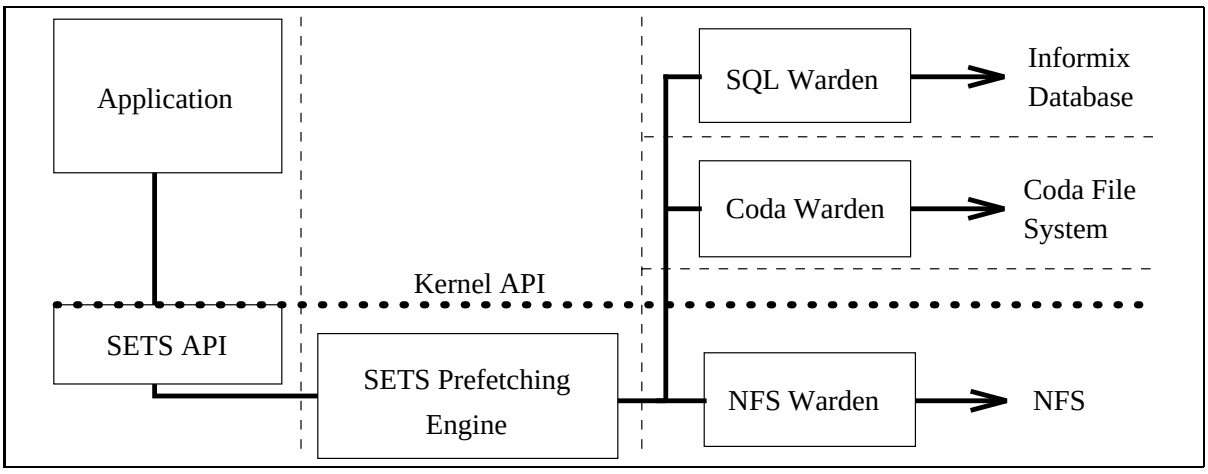
The `setWeight()` operation allows applications to inform SETS of the relative importance of the members. The `weights` parameter is an array of integers, one integer for each member of the set. The order of the array should match the order in which the member names were yielded by `setDigest()`.

In order to specify weights, an application first calls `setSize()` to determine the number of members, and allocates an integer array of the appropriate size. It then calls `setDigest()` to get the names of the members. Because the number of names returned depends on the amount of space in the buffer and SETS's progress in expanding the specification, the application may need to call `setDigest()` a number of times in order to get all the names. For each member, the application assigns a weight by filling in the associated entry in the integer array. When ready, the application informs SETS of the weights using the `setWeight()` operation.

4.3 SETS Internals

SETS is divided into three components: the API support, the prefetching engine, and the wardens. Each component communicates asynchronously through well defined interfaces to the other components. Decomposing the functionality in this way allows different portions of the expansion and processing of a set to be interleaved. Figure 4.3 illustrates the three components, and the following subsections discuss the components and interfaces in more detail.

To minimize the cost of interactions between SETS and the file system, SETS is implemented as an extension to the VFS mechanism. In BSD, this means that the operations in the SETS API are system calls, and that SETS is implemented in the kernel. The drawback of this decision is that kernel code is more difficult to implement, and is less portable than user level code. Trading implementation complexity and loss of portability for tight integration with the file system is reasonable for the SETS API layer and the prefetching engine, because these components manipulate file system objects (vnodes and buffers) and so benefit from the integration. However, the behavior of a warden depends on the distributed system to which it provides access, and may not need close interaction with the local file system.



The main components of SETS, the API layer, the prefetching engine, and the wardens, are depicted in this figure. Dashed lines separate different threads of control. The bold dashed line indicates the kernel boundary. The API layer extends the kernel interface with the SETS operations, the prefetching engine sits within the kernel. The wardens may either be in the kernel or not, depending on their implementation. The three wardens listed are implemented in SETS; other wardens (AFS, FTP, WAIS, etc) are possible but are beyond the scope of this dissertation.

Figure 4.3: The Architecture of SETS

Each of the components in SETS is implemented in a separate thread of control to allow maximal concurrency. The API layer runs in the thread of the caller[3], and asynchronously invokes operations in the prefetching engine to avoid blocking the application. The prefetching engine has a dynamically configurable number of worker threads, which communicate synchronously with wardens. Wardens may also be multi-threaded, and may interact with servers. Although this communication may be synchronous or asynchronous, most clients use synchronous communication in the form of remote procedure calls[10]. The boundaries between the thread domains in SETS is illustrated in Figure 4.3 using dashed lines.

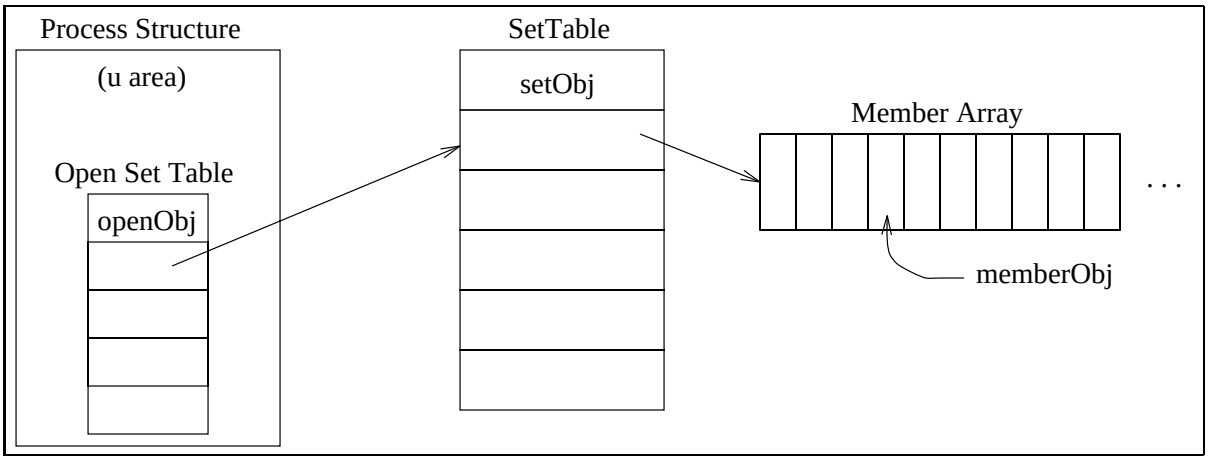
4.3.1 SETS API Support

The SETS API support layer has three functions. The first is to supply the operations in the SETS API, listed in Figure 4.1. The second is to maintain the data structures

[3]In BSD, there is a one-to-one mapping between threads and processes, whereas Mach allows multiple threads per process. SETS is designed to work in the multi-threaded case, and will therefore work in the trivial (and common) case of one thread per process.

which hold the state of an open set. The third is to invoke asynchronous operations to expand and populate the set.

4.3.1.1 The Data Structures

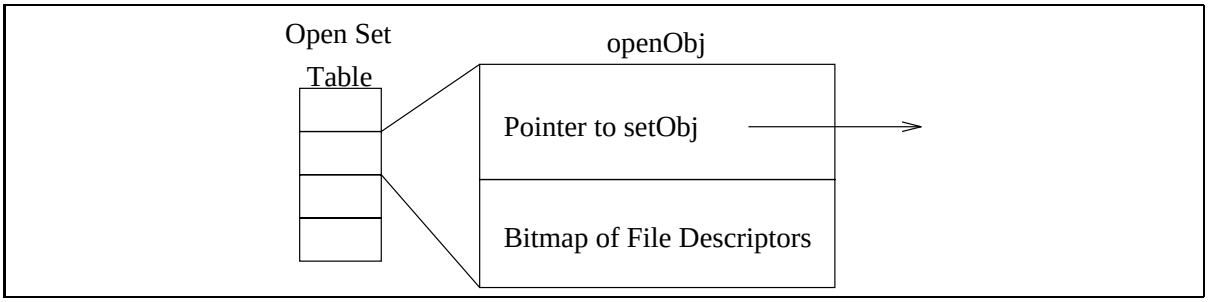


This figure illustrates the relationship between the three main SETS data structures. Each process keeps a table of `openObj`s as part of its process state (`u area`), each `openObj` contains the per-process state of a set and includes a reference to the `setObj` for the set. One or more processes, and thus multiple `openObj`s, can hold references to a single `setObj`. The `setObj`s reside in a system wide array called the `SetTable`. The members of a set are stored in an array of per-member structures called `memberObj`.

Figure 4.4: The SETS Data Structures

A dynamic set is represented by three data structures: a per-process open set structure, a structure for the set itself, and an array of per-member structures. Figure 4.4 depicts the relationship between these three data structures. Since overhead is a prime concern, the structures are designed to minimize both maintenance and storage costs. The data structures are maintained in volatile storage for lower overhead, which is reasonable since sets are temporary and thus a set's state need not be recoverable. Storing the SETS data structures in virtual memory allows quick access to the data when SETS is in use, and also allows the data to be paged out when no application is using SETS and the memory is needed for other purposes.

Per-Process Open Set Structure



The `openObj` structure contains the per-process state of a set. This consists of a reference to the `setObj` and an indication of which of this process's open files belong to (are members of) this set.

Figure 4.5: The `openObj` Data Structure.

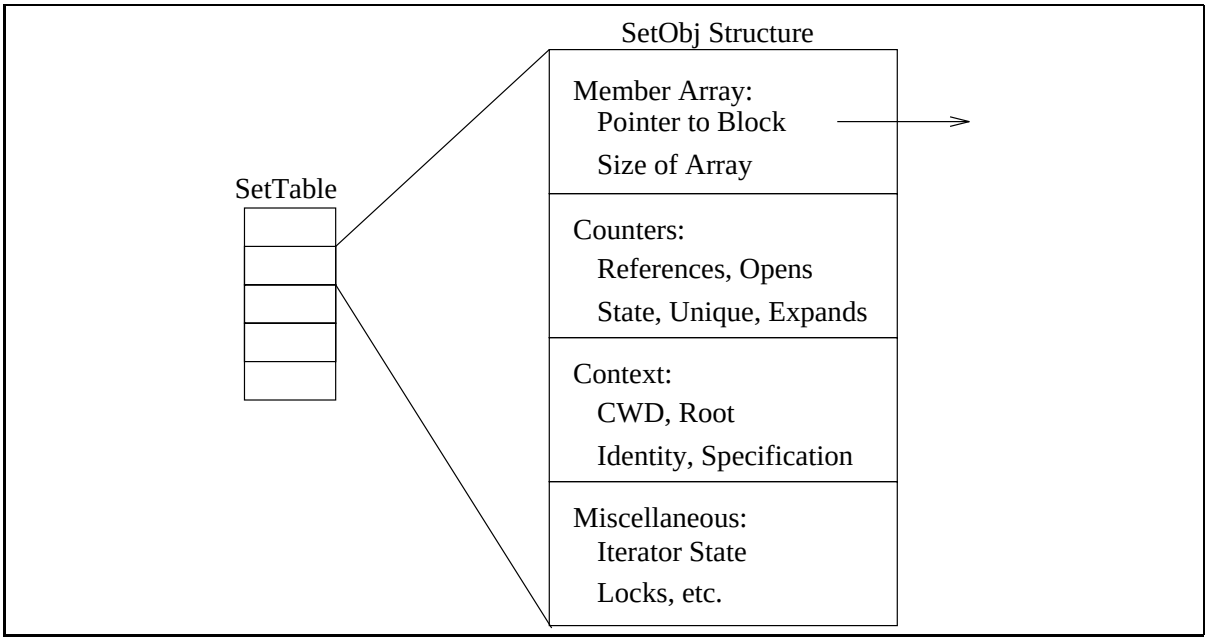
Although a set can be shared between processes, certain portions of the set's state are process specific. For this reason, a set is represented by the combination of its `setObj` and the collection of per-process `openObj`s, one for each process that has a reference to the open set. The `openObj` contains a reference to the `setObj`, and a bitmap to indicate which of the process's file descriptors have been allocated for this set's members. The `openObj` structure is illustrated in Figure 4.5.

Since a process may open more than one set at a time, SETS allocates an array of `openObj` for a process as a side-effect of the first set operation it calls. This array is similar to the open file descriptor table. It has a fixed size; SETS returns the error code `ENFILE` when all open set descriptors are taken. The open set handle returned by operations like `setOpen()` is an index into this table. The array is automatically deallocated when the process terminates.

Set Structure

The set structure, `setObj` is the central data structure used by SETS. Each open set has exactly one of these structures associated with it. The structure consists of four groups of components, each described in the following paragraphs. Figure 4.6 illustrates the layout of `setObj`.

The first part of `setObj` contains the array of members. The array consists of a pointer to a block of dynamically allocated memory holding the per-member structures. When the block is full, SETS allocates a new block which is twice as big as the old one, copies the members from the old block to the new one, and releases the old block. This technique allows a set to be arbitrarily big, while minimizing overhead and wasted space. Associated



The `setObj` structure is the basic data structure for a set. It contains an array of per-member data, counters, some of the state of the creating process, iterator state, and a lock to provide mutually exclusive access to the set.

Figure 4.6: The `setObj` Data Structure.

with the array are a counter holding the size of the block of memory, and a count of the number of members.

Although this member array could be quite large, SETS places a sliding limit on the set's cardinality by holding members open until they have been yielded to, processed by, and closed by the application. Thus at most there can be as many prefetched but unyielded members on the client as the number of slots in the system-wide file table.[4] The reason for this limit is that SETS needs to be able to keep objects in the cache and unmodified until it yields them. The easiest way of doing this short of modifying the cache management in the DFS client subsystem is to keep the file open.

The second part of `setObj` consists of a number of counters. One counts the references on this set to determine when the set can be released. Another counter is a uniquifier to differentiate this use of the `setObj` from past uses. A third counter holds the number of prefetches that have been started for this set, which in turn is used to control how aggressively SETS prefetches this set's members. A fourth counter holds the number of requests to expand this set's membership specification. SETS knows that the set's mem-

[4] Although tunable, the limit is 370 on the Mach 2.6 systems used here at CMU.

bership has been fully expanded when this counter drops to zero. Another counter is used to track the state of the set; SETS sets bits in this counter to identify state changes. For instance, SETS indicates that a set is fully expanded by setting the DONEEXPANDING bit in this counter.

The third part contains the context in which the set was created. This context is necessary to allow the prefetching engine's worker threads to resolve names and access members on behalf of the process that created the set. Three pieces of data are needed; all are available from the creating process's per-process structure in the kernel. First, the process's current working directory is necessary to resolve relative pathnames. Second, the process's file system root is necessary to resolve absolute pathnames.[5] Third, the process's access rights are necessary to ensure that the objects accessible through SETS are exactly those accessible to the process through normal file system operations. The context also contains a fourth piece, the string containing the membership specification used to create the set.

Finally, there are five pieces of miscellaneous information needed by different operations that are stored as part of the set. First, the `setObj` contains a pointer to a copy of the specification string, if one was used to create the set. Second, the iterators use two counters and a linked list, whose head is stored in the set. Third, the operations that derive a set's membership from other sets keep references to these sets in the `setObj`. Fourth, the set contains the head of a linked list of file system buffers holding member's data. The fifth piece is a lock data structure so that SETS can ensure the mutual exclusion of concurrent access to the set.

Member Structure

Each member of a set is represented by the `memberObj` structure. This structure contains the name of the member, a data field whose value depends on the state of the member, and two counters. The first counter, `nincore`, holds the amount of the member's data that is in memory. The use of this counter is described in Chapter 6. The second counter, `state`, holds a flag indicating the state of this structure with respect to expansion, fetching, and iteration. Figure 4.7 depicts the basic structure of the `memberObj` structure.

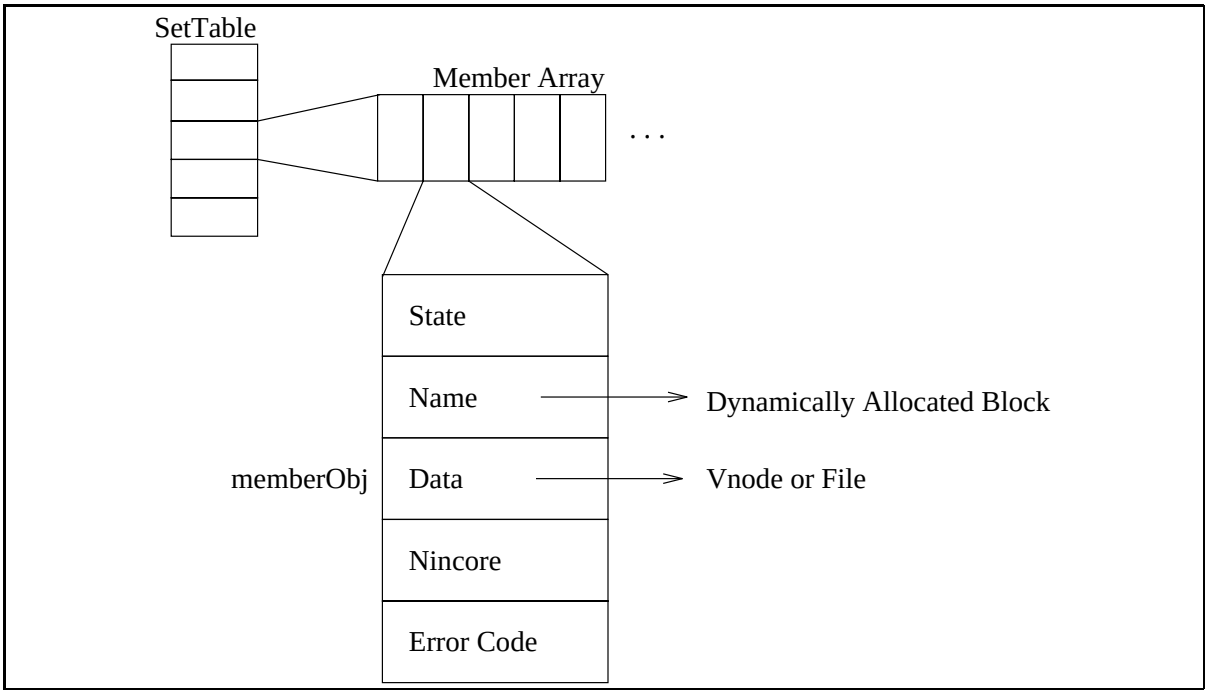
A member can be in one of six states: FREE, TAKEN, FETCHING, OPENED, FAILED, and SEEN. Initially, all elements of the member array are FREE, and the data in those structures is uninitialized. The `memberObj` structures are allocated as needed in order of increasing index in the member array. Thus the state of structures whose indices are greater than the size of the set must be FREE. When a member

[5] Unix administrators can deny a process access to portions of the name space by setting that process's root via the `chroot()` system call.

is added to the set, SETS allocates the first FREE structure by changing its state to TAKEN, filling in the name, and incrementing the counter containing the set's cardinality. The pointer to the member's data is left uninitialized.

When SETS decides to prefetch a member, it changes the set's state from TAKEN to FETCHING, and zeros the pointer to the member's data. When the fetch completes, the worker thread performing the fetch will change this pointer to point to the vnode holding the member's data, and set the state of the `memberObj` to OPENED. At this point, the member must now be local (it has been fetched) and may not be evicted from the local disk cache until the set is closed. If the prefetch ended in failure, the thread stores the error code in the data field and sets the state of the member to FAILED.

When an opened member is yielded, SETS changes its state from OPENED to SEEN. SETS also changes the data pointer to point to the open file structure, rather than the vnode. This change allows SETS to keep track of the files that have been opened by `setIterate()`. Since the open file contains a pointer to the vnode, SETS can still access the vnodes of yielded members.



The `memberObj` structure holds information on one member of a set. This information includes the member's name, a pointer to its data, an error code, and two counters.

Figure 4.7: The `memberObj` Data Structure.

4.3.1.2 Managing the SETS Data Structures

One of the main purposes of the SETS API layer is to manage the SETS data structures in response to invocations of SETS operations. This section describes at a high level how the API layer performs this task. Each of the following subsections describes one of the hard problems facing SETS, and how it solves the problem. Aside from the solution of these problems, the remainder of the functionality in the SETS API layer is straightforward and is not described further.

Set Allocation

When an application creates a new set, such as by `setOpen()`, SETS allocates new data structures to hold the set and its data. Allocation consists of finding an empty `openObj` in the process's open set table and an empty set structure in the system wide `SetTable`. An error code is returned if no empty slot exists in either table. SETS initializes the empty `openObj` to point to the empty `setObj` and sets the file descriptor bitmap to zero since no member has yet been yielded. The `setObj` structure is initialized by allocating a small array to hold the members, setting the counters to zero, initializing the locks, and initializing the iterators to indicate that expansion has not yet been started.

If the operation that creates the set is invoked with `SETS_ANTICIPATE_DIGEST`, a message to start membership expansion is passed to the prefetching engine. Thus the work to expand the set can be interleaved with application processing before any call is made on the open set. The expansion is I/O dominated, and so may be effectively interleaved with the application with little negative impact.

Similarly, applications can cause SETS to aggressively start prefetch operations on members as soon as their membership has been determined. This behavior, as well as aggressively determining membership, is selected by setting the `SETS_ANTICIPATE_ITERATE` flag. Setting this flag informs SETS that the application plans to use `setIterate()` to process the set. This hint declares to SETS that it is safe to initiate prefetching, since the application has declared that it will process at least some of the members. This flag is useful to those applications that can create a set some time before it is needed, and which are sure to iterate on the set.

Set Deallocation

Although the point at which to create a set is clearly defined, the point at which it is safe to deallocate the set is more ambiguous. In particular, it may not be appropriate to deallocate the set as a side effect of `setClose()`, since other outstanding references to the set might still exist. These references can come from different sources. For instance,

each process that has a handle on a set, such as would result from `fork()`, contains a reference to it; a set that is derived from another set, e.g. the union of two other sets, contains references to the sets from which it is derived; and every outstanding prefetch or expansion operation on this set contains references to the set.

In order to avoid reclaiming a set while one of these references is active, SETS maintains a count of active references in the `setObj`. When this reference count drops to zero, SETS can reclaim the set and the resources it consumes. SETS first releases any file system resources it may hold, such as buffers in the buffer cache, see Section 6.4.2. SETS then closes or releases any members in the set, depending on whether they have been opened. Finally, the member array, member name strings, and the set name are deallocated.

Iteration over Sets

The key operations in the SETS API are iterators, of which SETS provides two: `setDigest()` and `setIterate()`. `SetDigest()` yields the names of members, and allows searchers to scan the membership of a set without requiring SETS to fetch the members' data. With the names of members, searchers can decide to manipulate set membership through the set management operations. `SetIterate()` yields an open file, and requires that the entire file's data be fetched before it is yielded. `SetIterate()` is thus the primary call for applications wishing to process a set of objects.

The simpler of the two iterators is `setDigest()`. On the first call to `setDigest()`, SETS initiates the expansion of the membership specification into the list of names, if expansion has not already been initiated. SETS then copies the names of members into the input buffer until either the names of all known members have been copied or the buffer is full. As described in Section 4.2.2.4, the last entry in the buffer will contain the name "..." if the membership is not fully determined. Names are copied in the order in which they appear in the member array, a counter stored in the `setObj` stores the highest member index that has been yielded by `setDigest()`. Subsequent calls to `setDigest()` can then start at the next index, and thus do not have to rescan the entire array on each call.

`SetIterate()` is complicated by the need to fetch members before they can be yielded. Fetching is expensive, and the strategy used by SETS is sufficiently important and complicated to warrant an entire chapter (see Chapter 6). However, for purposes of this discussion it suffices to state that SETS initiates the prefetch of an object using a strategy designed to reduce latency while minimizing wasted resources. In addition, SETS ensures that all of an object's data is local when the prefetch operation terminates. Chapter 6 also describes the manner in which the prefetcher determines the order in which to yield prefetched members.

The job of `setIterate()` is to determine which objects are ready to yield and to pick one and yield it. To aid in the identification of unyielded objects, SETS keeps a counter in

the `setObj` to track the progress of the iterator. When `setIterate()` yields a member, it stores the index of the member in the counter if all members of lower index have been yielded. Thus the counter's value is a good place to start looking for unyielded members on the next iteration. As a further optimization, SETS also keeps a linked list of members which are ready to be processed. If more than one member is ready to be yielded, `setIterate()` can pull the first member off the list without having to search through the member array.

Sharing Sets Between Processes

Sharing open sets between processes introduces two issues. First, it raises the question of whether or not a set should be inherited by children of a process. Second, it raises the question of whether use of sets alters the file system's security model.

A process creates children processes with the `fork()` system call. To allow the child to share in the processing of set members, a set and its iterators can be shared between the child and its creator if the set is inherited across the `fork()`. An application can choose one of three inheritance behaviors for a set by setting bits in the `flag` argument when the set is created. The default behavior is to prevent sharing by not giving the child a reference to the set. By setting the `SETS_DUP_ON_FORK`, the application causes the child to get a new set that is identical to the parent's set, but future accesses to this set's iterator will not be reflected among the copies. To allow a child and parent to share the iterator, the application should create the set with the `SETS_SHARE_SET` flag. This causes a process and its children to share access to a set and its iterators. SETS manages concurrent access to the iterator to ensure that each member is processed at most once, allowing access to the members to be interleaved between parent and children.

Because set members are fetched using the access rights of the creating process, but can be accessed by children of the creating process, there is a potential breach in the file system's security mechanism. Fortunately, this impact of this breach is weakened by several factors. First, the access rights of the parent are inherited by their children, thus SETS provides no additional and unwarranted accesses. However, a process that changes its access rights will still access set members with the access rights of the set's creator. Second the children processes are running the same executable as the parent. Thus the introduction of a hole, for instance by changing the owner (`uid`) of the child process, can only happen if the application was intended to function in this manner. Third, the default behavior is to prevent the duplication of a set in a process's children, thus the security hole is only manifest if the application explicitly chooses to pass on the set to its children.

Managing Open File Descriptors of Members

In order to maintain the appearance that members belong to a set, SETS only allows a process to access the file descriptors of open members while the set is open. To prevent access after a set has been closed, SETS closes all files opened by `setIterate()` as a side effect of `setClose()`. Future references to these descriptors will fail and return the error code `EBADF`.

Unfortunately, keeping a list of the file descriptors is not sufficient to identify the descriptors that need to be closed. The process may have closed the file, and the descriptor may have been reallocated for another open file. Alternatively, the iterator may have yielded open members to more than one process if the set is shared. SETS addresses these problems by maintaining a list of open descriptors in the per-process `openObj` structure. When `setClose()` is called, SETS examines this list, and for every entry it looks at the member's list to determine if the open file descriptor corresponds to a member of this set. Note that simply marking the descriptor as belonging to a set and closing every descriptor so marked is insufficient, as a descriptor may belong to a different set than the one being closed. For reasons of efficiency this list of opened descriptors is maintained as a bitmap in the `openObj`. Other alternatives, such as keeping open descriptors on a linked list, have higher overhead and require changes to the kernel's generic file system data structures.

File descriptor management is further complicated by the `fork()` and `dup()` system calls. First, `fork()` can cause a set to be inherited by a child process. To ensure consistency between the child's open files and open sets, the per-process, per-set list of open files is duplicated in the new process. Second, `dup()` causes new descriptors to refer to existing open files. If a process called `dup()` on a file opened by `setIterate()`, SETS would not know that the new descriptor also corresponds to a set member, and would not close that descriptor when the set was closed. Rather than forcing these files to be closed, SETS retains this behavior to allow applications to retain access to members after a set has been closed. For instance, a search application could use `dup()` to retain access to the files that satisfied the search and close the set to free the resources consumed by the many other members of the set that did not satisfy the search.

Weights

The `setWeight()` operation allows an application to inform SETS of the relative importance of the set members. SETS treats these weights as hints and not as directives, which means it does not guarantee that objects will be fetched or yielded in order of largest weight to lowest weight. However, SETS does initiate prefetch operations on objects in

order of weight. In addition, the iterator will yield the object with highest weight out of the objects that are ready to be yielded when the iterator is invoked.

SETS implements this behavior by reordering the member array so that members with greater weight have a lower index than members of lesser weight. Since SETS uses the array to decide which object to prefetch next and which object to yield, the order of the member array affects the order in which objects are prefetched and yielded. However, this does not ensure that members will be yielded in order of greatest weight. For instance, the member with greatest weight may require the highest latency to fetch. If the application calls `setIterate()` before this member's fetch has completed, SETS will yield another member rather than force the application to block. Although the implementation does not do so presently, it would be possible to allow applications to request strict adherence to the weights via an option when the set is created.

4.3.2 SETS Prefetching Engine

Operation	Parameters	Description
`expand`	`pathname, suffix`	Expand a set specification.
`query`	`pathname, suffix, cursor`	Read names from a `cursor`, a handle on a running query.
`execute`	`pathname, suffix, pipe`	Read names from a `pipe` to an executing predicate.
`subset`	`bitmap`	Expand the membership of a subset.
`union`	`bitmaps`	Expand the union of two sets.
`intersect`	`bitmaps`	Expand the intersection of two sets.
`prefetch`	`memberObj`	Prefetch a (remote) set member.
`preread`	`memberObj`	Read a prefetched member into memory.

This table lists the messages and operation-specific parameters used by the SETS API to invoke asynchronous operations on a set. The first six message types correspond to operations which determine a set's membership, either from its specification or from the membership of other sets. For `expand`, `query`, and `execute`, `pathname` holds the portion of the specification that has already been examined and `suffix` holds the portion of the specification that needs to be expanded. The bitmaps in the `subset, union` and `intersect` messages are used to identify which members of the base sets have already been added to the membership of the derived set. The `prefetch` message requests the prefetch of a member, and the `preread` message tells SETS to read the data of a previously prefetched or local member from the disk into the Unix buffer cache.

Figure 4.8: Operations in the Interface to the SETS Prefetching Engine

The SETS prefetching engine performs the work of determining a set's membership and fetching its members. The operations it performs are invoked asynchronously by the SETS API layer in the form of messages. SETS API operations create a message containing the necessary information to accomplish the task at hand. The message is received by a worker thread, which performs the task. Since the messages are asynchronous, the worker thread has no direct way of informing the API layer that the operation has completed. Instead, it updates the SETS data structures to reflect the results of the operation. For instance, the worker changes the state of a `memberObj` from FETCHING to OPENED when an asynchronous prefetch completes.

The asynchronous messages are illustrated in Figure 4.8. Because the messages deliver the arguments to the worker threads, the type signature of the message is effectively the interface between the SETS API layer and the prefetching engine. There are eight operations in the interface: `expand`, `query`, `execute`, `subset`, `union`, `intersect`, `prefetch`, and `preread`. The semantics and syntax of these operations are discussed in the following sections, except the `preread` operation which is discussed in Chapter 6. Each message contains some general arguments (not listed in Figure 4.8) such as a reference to the set on which to perform the operation and a tag identifying the operation being invoked. This tag provides sufficient context to allow the worker thread to decipher the rest of the message.

The worker threads are implemented as *daemon* processes, a common technique for getting independent threads of control in the kernel. Although Mach supports kernel threads, support for independent threads is not universal and is therefore not assumed by SETS. The daemon makes one system call, `setDaemon()`, which never returns. Instead, it acts as a de-multiplexer, taking messages off a queue and directing them to the appropriate service routines. When it gets a message, the worker thread first assumes the identity of the process that created the set using the context stored in the set. It then decodes the message using the tag to identify the types of arguments in the message, and calls the service routine. When the service routine completes, the worker blocks until the next message arrives. Although it would be a simple matter for SETS to dynamically control the number of worker threads, the current implementation does not do so. The bootstrap scripts[6] start a default number (currently 4) of daemons at boot time, and the user can kill or start other threads as need arises.

Expanding the Set Membership Specification

Expansion of a specification into the names of a set's members starts with the `expand` message. When the API requests an expansion, it puts the set's specification (or name)

[6] The bootstrap scripts on BSD are `/etc/rc` or `/etc/rc.local`.

in the `suffix` field, and allocates another string of the same length in the `pathname` field. The expansion routine steps through the `suffix` string looking for special notation which indicates the need for expansion. When one of these special characters (such as "*", "\", or "%") is found, the portion of the `suffix` to the left of the character is appended to the `pathname`, as it has been fully expanded.

If the character indicates simple syntactic expansion, such as "`{foo,bar}`" or "`[abc]`", the expansion can be done directly. SETS first finds the delimiter, and determines the set of values which this token represents, for instance "foo" and "bar" in the first example. For each value, it creates a new message containing a copy of the `pathname` and a new `suffix` string. The new `suffix` has the old suffix appended to one of the expanded values. Because the expanded value is placed in the `suffix`, or unprocessed portion of the name, SETS will validate that the string is a valid file name when it processes the new message. To save the cost of allocation, the current message and its strings can be reused for one of the expanded values.

Wildcard expansion is similar, but instead of expanding the specification into possible names SETS must pattern match against the names of existing objects since the set of possible values is infinite. To do this, SETS first isolates the component that contains the wildcard character, and performs name resolution on the components in the prefix to get the vnode of the directory in which expansion will be performed. For instance, if the specification is the string "/dog/cat/h*rse", SETS would resolve the name "/dog/cat" to get the vnode of that directory. It then lists the names in that directory to find those that match the wildcard specification. In the example, SETS would examine the names of the children of "/dog/cat" to find those that begin with "h" and end with "rse". For all such names, for instance "horse" and "hearse", SETS creates a new expansion message. Unlike the syntactic case, names that match the wildcard specification are known to be valid, and so can be appended to the new `pathname` string instead of `suffix`.

The process of expanding *interpreted* or *executable* specifications is more complicated, but the basic process is similar to that for wildcards. SETS first identifies the component containing the query, which is simplified because this component can contain no other characters but those in the query. SETS resolves the prefix (leading up to the query's component) to obtain a vnode, and uses this vnode to execute the query. The suffix is used after the query executes to continue expansion. The details of query execution are discussed in the following two sections.

Evaluating a Query

When a set is opened using a interpreted membership specification, SETS runs the query delimited by the "\" character on the vnode named by the prefix of the specification. For example, given the second specification in Figure 4.2, SETS would run the query

"`select home where name like "%david%"`" on the vnode named by "`/staff`". The query is a string which expands into a list of file names when interpreted by the warden responsible for the "`/staff`" vnode.

SETS invokes a query expansion by passing the string containing the query to the warden using the `ody_lookup` operation[7]. The warden immediately returns a special vnode called a *cursor*, which acts as a handle for the query. This allows the query to be executed asynchronously without forcing the worker thread to block. Instead, the worker thread queues a `query` message which will cause the results of the query execution to be read from the warden. The `query` message contains a reference to the cursor, as well as the `pathname` and `suffix` fields of the `expand` message which triggered this query expansion.

When the `query` message is received by some worker thread, it invokes the `ody_expand` operation on the cursor contained in the message. For each name returned by `ody_expand`, SETS creates a new `expand` message. If the name is an absolute pathname, the `pathname` string contains only the new name, otherwise the name is appended to the `pathname` from the `query` message. The `suffix` field of the new message is a copy of the `suffix` field of the query. Because the names returned by the query are put in the `pathname` field, they are assumed to be simple file names and not to contain set specification characters. Because existing search engines do not currently support dynamic sets, it is reasonable to assume that the names they return will not contain set specifications. However, one could modify SETS to allow a query to return the name of a set of objects in addition to the name of a single object by putting the name in the `suffix` field of the new messages instead of the `pathname` field.

The motivation behind providing two messages for query execution is to allow the query to run asynchronously. The evaluation of the query may take some time because the warden may need to communicate with servers or fetch objects to resolve the query. By queueing a message to retrieve the results at a later time, the worker thread can perform useful work while the warden is blocked on I/O. If there is no other work to perform, the `query` message will be dequeued and the worker will block until the query expansion has produced some results. After the worker has processed any available query results, the `query` message can be requeued to stimulate future expansion.

Reading the Results of an Executable Predicate

The processing of an `execute` message is similar in nature to the processing of a `query` message. As stated above, SETS runs executable predicates in a newly created process. The predicate communicates the identity of files which satisfy the predicate by writing their names to standard output. Before invoking the predicate, SETS redirects the new

[7]The "ody" prefix and the name "warden" are inherited from the Odyssey project.

process's standard output to a pipe. Thus SETS can read the results of the predicate by reading the names from the pipe. Because the execution of the predicate may take time, SETS queues an `execute` message, similar in nature to a `query` message, and allows the predicate to execute asynchronously. As with names returned by a query, SETS creates an `expand` message for each name the predicate returns. The new name either replaces or is added to a copy of the `pathname` string, and so must not contain set specifications.

The implementation of the support for executable queries has been delayed by the need for a means of sandboxing[94] these queries. When the dissertation was started, the sandboxing mechanism was supposed to have been ported to Mach 2.6, but has unfortunately been delayed indefinitely. As a result, the mechanism to support executable queries is the only portion of this implementation that has not been completed.

Fetching Members

When SETS has completed the expansion of an `expand` message, the `suffix` field is null and the `pathname` field contains a complete pathname of one member of the set. The worker thread which finished the expansion adds this member to the member array by allocating an unused entry (growing the array if necessary) and setting the entry's state to TAKEN. It then moves the `pathname` from the message to the entry, and deallocates the message.

When SETS chooses to prefetch this member, it creates a `prefetch` message which contains the index of this entry. When this message is dequeued, the worker thread invokes the `ody_prefetch` operation on the object. This operation is performed by the warden supporting the object; the actual work depends on the implementation of the warden. See Chapter 6 for more details.

Deriving Membership from Other Sets

The set operations `setUnion()`, `setIntersect()`, and `setRestrict()` create a new set whose membership is based on the argument sets. When one of these operations is invoked, SETS creates a new set, adds references to the arguments (base sets) and creates a `union`, `intersect`, or `subset` message. Each message contains a bitmap for each base set. The bitmap is used to identify which members of the base set have already been included in the derived set. The basic algorithms are straightforward, the bitmaps are used to reduce the cost of comparing members from that of a name comparison to that of testing a bit.

When a worker thread receives a `union`, `intersect`, or `subset` message, it first checks that the bitmap accurately reflects the state of the base sets. The bitmap may be incorrect

if the member array of a base set was grown after the message was queued, or because a `setWeight()` operation reordered a base set's member array. In the former case, the bitmap needs to be grown to reflect the new size of the member array. In the latter, the bitmap state is suspect and needs to be rebuilt.

Although not at first apparent, zeroing out the bitmap and restarting the derivation of membership will not introduce membership errors and is the simplest way to regenerate the bitmap. The argument for correctness of membership must consider three cases. The first case consists of objects that were members of the derived set, but will be re-inserted since the bitmap information is lost. However, SETS prevents duplicate members so the re-insertion will not affect the derived set's membership. The second case consists of objects that were a part of the derived set, but for some reason no longer satisfy its membership criteria. These objects will remain members, even though recalculation of membership would exclude them. This is acceptable given the semantics of dynamic sets described in Section 3.1.3.6, which state that a member need only satisfy the membership criteria *at some point* between `setOpen()` and `setClose()`. The third case to consider consists of objects that were not part of the derived set, but will be added as a result of zeroing the bitmaps. As with the second case, adding these objects is appropriate given the semantics of dynamic sets.

The use of the bitmaps depends on the operation being performed. For a `union` operation, the bitmaps indicate which members of the source sets have been added to the derived set. For `intersect`, the bitmap is used to detect which elements of one of the source sets have been added. Since a member can be added only if it belongs to both source sets, maintaining one bitmap is sufficient to detect which objects have been added. Finally, `subset` sets a bit when a member of the source set has been examined. Once it has failed the predicate used to restrict membership, it can be safely discarded from consideration since the member may not change while the source set is opened.

4.3.3 Wardens

As in Odyssey, SETS relies on wardens to provide type-specific functionality. For SETS, type-specific means system specific: each warden contains the client subsystem for some DFS, GDIS, or other distributed system. Since the client subsystems for DFS already use the VFS interface, SETS uses the VFS interface augmented with four SETS-specific operations listed in Figure 4.9 as the interface between it and the wardens. The VFS interface consists of basic file system operations. The SETS extensions allow wardens to register themselves with SETS, execute interpreted (e.g. specific to this warden) queries, report the results of these queries, and prefetch objects.

There are several advantages of this approach. First, existing DFS must already provide a VFS driver. Extending this driver to support the SETS operations is straightforward.

Second, Coda's VFS driver is written to allow the Coda client subsystem to run as a user level process[91]. It is also a simple extension to add support for the SETS operations to this driver, allowing wardens to run as user level processes as well. This in turn greatly simplifies the implementation and distribution of new wardens, and allows new wardens to be dynamically added and removed from a running system. Third, wardens need only implement a subset of the warden interface: those operations that make sense for the type of data the warden supports. For instance, the SQL warden only supports query execution and does not support any basic file system operations.[8]

Operation	Input	Output	Description
`ody_mount`	Mount data	Vnode	Mount warden in name space.
`ody_lookup`	Vnode, Filename	Cursor	Invoke a interpreted query.
`ody_expand`	Cursor	List of names	Get the names of objects returned by the query.
`ody_prefetch`	FileID, Flags	Vnode of Cache File	Prefetch an object.

This table lists the operations that have been added to the VFS driver interface[42] and to Coda's VFS driver[91], called the Minicache. The operations allow a warden to be dynamically added, for the warden to execute queries and report their results, and for the warden to prefetch objects as directed by the SETS prefetching engine. The Minicache allows other VFS drivers (and thus wardens) to run in user-level processes.

Figure 4.9: Operations in the SETS Warden Interface

A disadvantage of this approach, providing a well-defined interface between SETS and wardens, is that it limits the sharing of information between these levels. For the purposes of the thesis, this limitation has not proven to be a significant impediment. However, one may wish to add more operations to share information or to more tightly integrate wardens with the SETS prefetching engine if one were implementing SETS as a commercial system.

4.3.3.1 Interacting with Wardens

In order to service requests, a warden first registers itself with the Coda VFS driver, or Minicache, by supplying a name for the type of service it provides. The Minicache

[8]Providing a mapping between file system and database operations would be a very interesting research topic, but is unfortunately outside the scope of the thesis.

keeps a table of these names for currently active wardens. Mount points are symbolic links, whose link value matches the pattern "`%*.*`". The string between the "`%`" and the "`.`" should be the type name of a warden, the string following the "`.`" is data that is interpreted by the warden to identify the portion of the name space it represents that should be mounted at this name. For instance, the mount point "`%sql.gershwin@10000`" causes the SQL warden to mount the Informix database whose server is located on the machine "`gershwin`" and is listening at port 10000.

When one of these symbolic links is discovered during name resolution, the Minicache looks in its table to find if the warden supporting this service is currently active. If it is not, the operation that invoked name resolution will fail because the mount point looks like a dangling symbolic link. If the warden is found, the Minicache sends the data string to the warden using the `ody_mount` operation, and the warden returns a vnode which serves as the root of this file system. The Minicache then creates a *VFS*, virtual file system, for this mount point, and links it into the name space. Future references to the mount point will discover the VFS, and can proceed without remounting. All children of this mount point belong to this warden (unless one of the descendants is another mount point), operations on them are automatically sent to the warden by the Minicache.

The advantage of this mounting mechanism is twofold. First, wardens can be dynamically added or removed from a system simply by starting the warden program. Because mount points are symbolic links, they can be added by users, administrators, or the warden itself using existing Unix tools. It should be noted that this mount mechanism is only needed for wardens using the extended Minicache to interact with the kernel. Wardens for existing file systems (such as NFS) can utilize their own mount mechanism. These systems are tightly integrated with the file system, and SETS can easily determine the system that owns an object by examining the object's vnode.

The `ody_lookup` and `ody_expand` operations are used to invoke and determine the results of a query. `Ody_lookup` returns immediately with a handle on the active query called a *cursor*. A cursor is a temporary vnode with no name in the name space, and is only known to SETS and the warden. SETS uses the `ody_expand` operation to obtain the query results: the names of the objects that satisfied the query. To reduce the potentially high cost of contacting the warden, `ody_expand` batches the results, returning as many names as will fit in the buffer or are currently available. If no names area available, `ody_expand` will block until the executing query produces another result.

The `ody_prefetch` command fetches the object identified by `fileID`. The `fileID` is a low-level identifier or name used by the distributed system client subsystem to uniquely identify the object. The `fileID` is discovered during name resolution on the object's name, and is stored in the object's vnode. The `ody_prefetch` operation should block until the object has been fetched. Because the precise meaning of fetching an object depends on the semantics of the file system, SETS defines the following minimal condition: A

prefetched object must be able to be opened and read by an application without suffering any network delays. The subject of prefetching and the reason for this condition is described in Chapter 6.

4.4 Summary

This chapter has described the detailed design of SETS, which extends the file system API of the BSD operating system to support dynamic sets. The next two chapters discuss to related but orthogonal pieces of work. Chapter 5 describes several applications and wardens that respectively use dynamic sets and support them. Chapter 6 describes the SETS prefetching engine in more detail.

Chapter 5

Examples of Applications and Wardens

This chapter describes how I modified several applications and distributed system client subsystems to use and support dynamic sets. The goal of these modifications was to gain experience in the use of SETS and feedback on the design of the interface, as well as to provide a vehicle for the experiments described later in this dissertation. Section 5.1 describes wardens, client subsystem extensions which allow SETS to prefetch file system objects and to submit interpreted queries for evaluation. Section 5.2 describes several Unix applications which I modified to use dynamic sets.

5.1 Wardens: File System Support for SETS

In order to create a realistic environment in which to explore the benefits of SETS, I have implemented several wardens which give SETS applications access to objects in several different kinds of distributed systems. The Coda and NFS wardens are small extensions to existing local-area distributed file systems, and do not support queries. The HTTP warden allows SETS to prefetch objects using the WWW's HTTP protocol[7], and to run queries to WWW search engines. The SQL warden allows SETS to run SQL `select` statements as queries, but does not support normal file system operations.

The following sections describe each of these wardens in more detail. The selection of these four wardens is based on the need to provide a realistic environment in which to evaluate the usefulness of dynamic sets. The list should not be considered exhaustive nor exclusive. Adding new services, such as wardens which provide access to AFS objects, or use of WAIS or GLIMPSE search engines, should be straightforward for programmers with knowledge of the client subsystem code and of SETS.

5.1.1 The Coda Warden

The Coda warden consists of a few simple extensions to the Coda client cache manager, *Venus*, which allow SETS to prefetch Coda objects. Coda is a highly available file system which approximates Unix semantics in a distributed file system context[82]. High availability is achieved by replicating objects at multiple servers[46] and by exploiting local cache state through disconnected operation[41]. Venus is a user-level daemon which caches whole-files on the local disk in response to a reference to an uncached file. Caching a file is a heavy weight two-phase operation. The first phase involves fetching attributes of the file from the servers storing the file's replica and comparing the state to determine if the replicas have diverged (which can happen in the presence of network partitions between the servers, for instance). In the normal case, the replicas are identical and the second phase involves picking one of the servers and fetching the file's data from it. If the replicas have diverged, Venus initiates a *resolution* operation to bring the replicas into a consistent state.

Since Coda caches whole files to a cache container file on the local disk, the `ody_prefetch` operation is similar to opening a Coda file for reading. If the file is cached, there is no difference between prefetch and open. If the file is not cached, a prefetch causes Venus to open the cache container file with a special flag which indicates the file is being prefetched. This flag causes the SETS buffer cache management to manage the file's data, as described in Chapter 6. In addition, Venus prevents mutations to the cached object by marking it *read-only* for the duration of the set.[1] The Coda Warden does not support interpreted queries, and so returns an error if `ody_lookup` is invoked on a Coda file.

The current Coda warden does leave a potential semantic violation of the dynamic sets consistency guarantees because a file may already be cached and opened by some other process when the set is opened. If the file is opened for writing, its state may change while the set is being processed. A solution would be to create a separate read-only cache file (possibly recaching the object). However, since this case is unlikely to occur frequently in practice, the solution to this hole is left as a future enhancement.

5.1.2 The NFS Warden

The Network File System[77] is another local-area network file system. Unlike Coda, NFS was designed to work with diskless workstations and therefore does not cache information

[1] Although set open and close calls are not passed through to the warden, Venus can still satisfy the semantics of dynamic sets by preventing mutations to the file after the `ody_prefetch` operation returns. When the file is closed, Venus can once again allow mutations because this file can no longer be accessed as part of this set.

to the local disk. Instead, it caches blocks of a file in the kernel's buffer cache just as the local file system caches disk blocks. When a block is evicted, NFS refetches the block from the servers. NFS Servers are designed for maximal simplicity and speed, and in the best case a block fetch will hit in the server's buffer cache. For fetches that do hit in the server's buffer cache, a remote NFS read can take less time than a read of data off the local disk would take. Further, the NFS client subsystem has low overhead because it does not need to manipulate local disk cache container files when it fetches data.

Extending this client subsystem to support dynamic sets thus posed a problem. Caching prefetching data in the buffer cache could result in wasted I/O if the prefetched block was evicted before it was read. Since buffer caches can be small, this either greatly reduces the amount of data that can be prefetched or forces the application to block on network stalls, violating the conditions required of the prefetch operation.

Two solutions to this problem exist. First, since network reads may be faster than disk reads, it may be faster to refetch evicted blocks from the server (in the best case for NFS performance). Thus the prefetch operation should read as much of the file as it can, limiting reading to the amount of buffer cache space available. Any blocks that are evicted will be refetched using the normal NFS read operation. The second solution is to maintain a local disk cache for evicted blocks. This second alternative should perform better than the first when local disk reads are faster than remote server accesses, such as when the server or network is under load, or when it is likely that the block has been flushed from the servers cache. Blocks can either be stored in cache container files as in Coda, or as raw blocks on the disk. The former approach is cleaner and simpler, but has higher overhead due to maintenance of the file system meta-data for these container files.

In SETS, the NFS warden implements both of these solutions. The solution that is used is selected by the system administrator by setting a parameter in the warden's memory. A more elegant solution would dynamically select whether to evict a block or store it on disk, basing the decision on the client's observations of disk and server/network speeds. However, dynamic adaptability is left as future work.

Like Coda, the NFS warden prefetches whole files. Worker threads submit synchronous reads to the server using the standard NFS client/server interface. If no buffer is available to hold incoming data, the worker thread blocks until a buffer becomes available. The drawback of whole file caching is that SETS must wait until the entire file is cached before yielding it to the application. This places a limitation on the performance improvement SETS can offer: the application must wait until one file has been fetched at the least. This limitation could be relaxed by allowing SETS to yield a file before all its data has been fetched, but these changes are left as a future enhancement.

This aggressive strategy works well in practice, but can be inefficient if the file is larger than the buffer cache, since later blocks in a file will evict earlier ones. When the

application reads the file, it will miss on the evicted blocks and suffer a delay to reload the blocks into the cache. If this problematic behavior proves to be commonplace, the NFS warden can instead terminate the fetch of a large file when it runs out of buffers, and let SETS preread the rest of the buffers into the cache when it yields the object to the application.

5.1.3 The HTTP Warden

The HTTP warden serves three purposes. First, it allows WWW objects to be accessed through the file system, allowing them to be members of dynamic sets. Second, it supports the SETS prefetch operation, allowing WWW objects to be prefetched and cached locally. Third, it supports WWW queries, allowing applications to query WWW search engines using interpreted specifications. The following subsections describe these functions in more detail.

Out of the variety of WWW protocols, this warden only supports HTTP, the hypertext transport protocol. Although HTTP is only one of the protocols used in the WWW, it does provide access to most of the objects in the WWW. Support for other protocols, such as FTP or Gopher, could be added for more complete coverage of the WWW, but would not significantly strengthen the dissertation.[2]

The HTTP warden is structured as a single multiplexor thread which executes the operations in the warden interface, and a dynamically configurable number of worker threads. The multiplexor answers any requests that can be immediately satisfied, such as fetch requests that hit in the cache. The multiplexor passes operations that require network access off to a worker thread. For most operations, the caller is blocked until the worker thread has completed the work. Queries, however, are run asynchronously from the caller using a cursor to represent the running query as described in Section 4.3.2. The HTTP warden only supports non-mutating file system operations and the four SETS extensions, other requests are rejected by the multiplexor.

5.1.3.1 Accessing WWW Objects

The HTTP warden allows WWW objects to be accessed through the local file system name space. The root of this subtree is a pseudo-directory on which the only valid operation is name lookup. The HTTP warden presents the WWW as a flat name space

[2]The HTTP warden also supports the trivial "`file://localhost`" protocol, by which files on the local machine can be accessed. This allows users to access data through the HTTP warden which they could also access directly from the file system, and is thus an inefficient if trivial extension to the HTTP warden.

below this root, thus URLs are treated as a single component in a Unix pathname. For instance the WWW page with URL "`http://www.cs.cmu.edu/`" looks like a child of the HTTP warden's mount point. In response to a name lookup of a URL, the HTTP warden first determines if the object has been cached. If it has, it returns the identification of the cache file. If not, it creates a local vnode to hold the object, but does not fetch the object until the object's data is requested.

The HTTP warden fetches an object in response to a request for the object's state or data, such as a prefetch or open operation. Fetching consists of opening a TCP connection to the server, sending a request for the object, and reading the response. The response consists of header information such as the size and type of the object followed by the object itself. The HTTP warden writes the entire response (header and body) to a cache file because some applications use the header information, e.g. to determine the object's type. As with Coda files, reads and writes to the object will be directed to the cache container file by the Minicache without contacting the warden.

Caching WWW objects introduces the possibility that the cache copy will become stale if the server's copy is updated. Since true consistency requires revalidation on use of the cached copy, and since this validation can be nearly as expensive as fetching the object if the object is small, the HTTP warden evicts objects from its cache when the last reference to it is closed. This approach simplifies the implementation by avoiding the need for cache coherence mechanisms, but is also likely to lower cache hit rates in practice. For purposes of the dissertation this is an acceptable penalty. The experiments which use the HTTP warden do not make extensive use of the cache, and the simplified warden demonstrates the generality of SETS as well as a more thorough implementation would do.

Many objects in the WWW are HTML documents. An HTML document is a hypertext *page*, a node in the hypertext graph. HTML pages contain a mixture of text and tags. The tags either dictate the layout of the document, or contain links to other objects. Many HTML pages also contain *inlined image* tags that contain the URL of a picture which is displayed as part of the document, and so must be fetched along with the WWW page. The latency of fetching these images can be a significant component of viewing HTML documents.

To prevent SETS applications from stalling on inlined images contained in prefetched HTML documents, the warden prefetches these objects as well. It does so by parsing incoming HTML documents looking for inlined images. When an image tag is found, the HTTP warden initiates a fetch on the URL contained in the tag. This behavior can be disabled to avoid wasting bandwidth if the images are not desired by the application fetching the document. In the current implementation, this behavior can only be selected at warden start-up time, although it would be interesting to explore interface extensions

to allow the application to select automatically fetching inlined images at set creation time, or when a fetch is initiated.

5.1.3.2 Running Queries on the WWW

The chief purpose of running queries on the WWW is to allow set membership specifications to utilize WWW search engines. In response to a query, WWW search engines return an HTML document which contains links to the documents that satisfied the query, along with other information such as advertisements, links to other services provided by the search engine, etc. The query is a URL which contains the name of the search engine and the arguments to the query function. Although it does not at first appear elegant, formatting query results as an HTML document allows search engines to be utilized without any direct support by the browser.

To allow a set membership specification to utilize search engines, the HTTP warden services *http-specific* queries. An http-specific query is simply the URL of an HTML document. The objects identified by such a query are the objects which are referenced or *linked* by the HTML document. Treating the URL of HTML pages as queries thus achieves the goal of allowing SETS applications to query search engines. However, it can also be useful to think of other HTML documents as sets of objects. Many HTML pages are in essence collections of objects. For instance, the URL "`http://www.brandonu.ca/~ennsnr/Cows/pictures.html`" is the name of an HTML document which is almost entirely composed of hypertext links to pictures of cows. It is natural to think of this URL as referencing a set of cow pictures instead of just the document.[3]

The HTTP warden handles queries by fetching the object to which they refer. If the object is an HTML object, it parses the object to find anchor tags which reference other WWW objects. An anchor tag contains the URL (name) of the linked document; the warden extracts this name and returns it to SETS through the `ody_expand` operation. The warden does not include inlined images in the result set, since they are actually part of the HTML document itself, and not part of the set of objects this document references.

Because some search engines return more links than just those that satisfied the query, the HTTP warden has a few search-engine specific routines that understand the format used by the search engine and filter out extraneous links. For instance, the `Lycos` search engine[51][4] includes an advertisement and links for related queries. This extraneous material is filtered from membership by a routine in the warden that is hand-written to parse the results of Lycos queries. Similar routines parse the results from other popular

[3]The reader should note that allowing HTML documents to be treated as queries does not require all HTML pages to be treated as such, the user chooses which HTML pages to treat as sets and which to parse and display.

[4]Lycos is available as "`http://www.lycos.com`".

WWW search engines. When parsing the HTML file, the warden first examines the URL to see if this object came from one of the known servers.

Although this mechanism may be error prone, it works well enough in practice. First, eliminating extraneous links in this manner is only an optimization, failure to recognize a search engine only results in seeing the extra material returned by the engine in the set's membership. Second, there are only a small number of general purpose search engines, so tuning the warden to these engines is not difficult. Third, the likelihood that an HTML page would have a similar name and format to a search engine's response is extremely small.

5.1.4 The SQL Warden

One of the early motivations for wardens was to provide navigational databases which index the contents of a file system allowing users to associatively name objects. SETS provides this functionality through the SQL warden. Queries to the SQL warden are SQL `select` statements which select a field (or column) of the database which contains a pathname. For instance, a query to an employee database could select the field which contains the pathname of the employee's home directory. The SQL warden only services `ody_mount`, `ody_lookup`, and `ody_expand` operations, it does not perform prefetches or any of the VFS operations.

The SQL warden consists of client and server subsystems which interact using the RPC2 remote procedure call package[79]. The server interacts with an Informix SQL database[71] using "embedded SQL", a library of functions supplied by Informix to allow programs written in C to invoke SQL operations. The server exports three operations: `OpenCursor`, `ExpandCursor`, and `CloseCursor`. `OpenCursor` takes an SQL `select` statement and submits it to the database to create an SQL cursor. `ExpandCursor` reads information out of the cursor and writes it into a buffer for transport back to the client. The buffer is passed as an RPC argument, and so is limited in size by the maximum transfer unit of the network. When the results of the `select` statement have been read, the client deallocates the cursor using the `CloseCursor` operation. The database is stored on the local disk of the machine on which the SQL server is running.

The client satisfies query requests from a SETS worker thread by invoking operations on the SQL server. When it receives an `ody_lookup`, it transmits the query to the SQL server using the `OpenCursor` operation. Similarly, for each `ody_expand` operation the warden invokes a `ExpandCursor` and sends the results to the kernel. When `ExpandCursor` indicates that all data has been read from a cursor, the warden calls `CloseCursor` and returns an error code to the kernel to inform it that the query is exhausted.

Each SQL server can access only one database, but a single SQL warden can submit

queries to any number of different SQL servers. SQL mount points are used to identify which server, and thus which database, to submit a request. SQL mount points include the name of the machine on which the database resides and a port to which to connect. When the warden receives an `ody_mount` operation, it establishes a connection with the server using the machine name and port number contained in the mount point.

The reader should note that the SQL warden does not provide consistency between the database and the file system which it catalogs; consistency must be ensured by the database administrator. For instance, SETS will not automatically update a database that indexes a user's mail files when a new mail message arrives.

5.2 SETS Applications

To validate that dynamic sets can be used by writers of search applications, I modified several existing tools to use dynamic sets. As with wardens, the applications are not the focus of the dissertation but a means with which to explore the benefits of dynamic sets. I have chosen to modify existing applications to use dynamic sets rather than write new ones for three reasons. First, using existing applications saves me the effort of reimplementing existing functionality that is orthogonal to the use of sets. For instance, a significant portion of the Mosaic browser consists of the code to parse and display HTML documents, and does not need to be modified for Mosaic to use SETS. Second, use of existing applications provides a better means of determining the benefits of sets. For instance, many Unix users have direct experience with `grep`. Comparing the performance of `grep` modified to use sets to that of an unmodified version of `grep` isolates the benefits of using SETS. Third, this exercise is a qualitative evaluation of the design of dynamic sets, and will offer support for the claim that dynamic sets are simple to program.

The following two subsections describe the two classes of applications that I have modified. First, Section 5.2.1 describes the modifications made to several standard Unix utilities. The key observation is that making these modifications is a trivial process. Then Section 5.2.2 discusses extensions I made to an interactive WWW browser to expose dynamic sets to the graphical user interface.

5.2.1 Extending Unix Utilities to Use Dynamic Sets

Many Unix utilities such as `grep` and `more` are written to allow users to utilize their ability to specify groups of files using the `csh` wildcard notation. For instance, given a list of files, `more` prints out the contents of each file in turn. Currently, the shell expands the wildcard notation into a list of file names. Many applications iterate over this list

processing each member serially, and consequently share a similar top level structure. In this structure, applications first process any command line switches, then loop over the remainder of the arguments, processing each one in turn. The details of the processing are hidden in lower level procedures.

Modifying these applications to use sets is a trivial exercise. Rather than letting the `csh` expand the wildcard notation[5] and looping over the result, the SETS version of these applications uses the notation to open a dynamic set and iterates over the set using `setIterate()`. The details of the application remain unchanged. Figure 3.2 presents the structure of the main loop of `grep` both with and without the use of SETS.

The benefit of using these modified Unix utilities extends beyond those of prefetching. For instance, it is now possible to examine a set of WWW objects using `more` using the command line "`sets_more '/tmp/www/\http://altavista.digital.com/cgi-bin/query?pg=q&what=web&fmt=.&q=cow\'`", which will print out a set of WWW objects containing the word "cow". Similarly, one can use grep and an SQL query to locate the `.cshrc` files of Coda members and search for occurrence of a substring. As an example, one can determine the approved values of `csh PATH` variables with the command line "`sets_grep PATH '/tmp/sql/\select home from users where given like "%d%"\/.cshrc'`", which will print out occurrences of the string "PATH" in the `.cshrc` files of users with a "d" in their given name. Note that both this and the previous example work as shown on SETS.

5.2.2 Mosaic: User Interface Support for Sets

The basic function of a browser is to load and display information from the WWW, and browsers like Mosaic are among the most commonly used search tools on the WWW. To explore the use of dynamic sets in a WWW application, I have modified NCSA Mosaic version 2.6 to use SETS. Mosaic was among the first of the WWW browsers to display information graphically, and at the time the dissertation was initiated Mosaic was the most popular graphical browser whose source code was in the public domain. At the time of this writing, Mosaic is less popular than it was, but is still the most popular of the freely available browsers.

Mosaic displays the contents of a file based on its type. The type is either inferred from the file name, or specified by the server that supplies the file. When accessing the WWW, users can instruct Mosaic to load and display a file by clicking the left mouse button on an *anchor* in the displayed document, by selecting a dialog box and typing in the file's URL, or by filling out a *form*. Forms allow WWW users to enter information (such as a query) to be sent to a search engine. On receipt of such a query, the search engine

[5] This can be disabled either by setting the `noglob` variable in the shell or by enclosing the wildcard notation in quotation marks.

creates and returns an HTML document that contains hypertext links to the WWW objects that satisfied the query. Mosaic then parses and displays the result as it would any other HTML page.

The extensions to Mosaic expose the dynamic sets abstraction to the user, allowing him to create and manipulate sets using widgets in the graphical user interface (GUI). References to WWW objects as members of sets are directed through the file system to the HTTP warden. References to WWW objects independent of SETS use the default mechanism which loads the pages directly into Mosaic's memory.

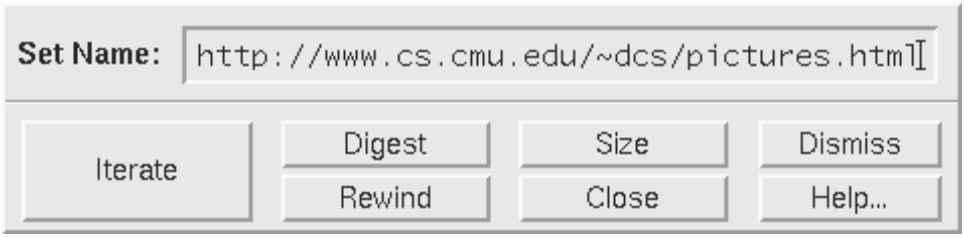
The majority of the modifications to Mosaic involve extensions to the GUI. The changes amount to 2000 new lines of code (out of the 100000 lines in Mosaic 2.6) and took me one month to implement. Although this is significantly more than the changes required for Unix applications, the changes were mostly mechanical. For instance, the code for the widget with which a user opens a set is almost an exact copy of the code for the widget to load a WWW object.

5.2.2.1 Extending Mosaic's GUI to Support Dynamic Sets

The GUI extensions consist of hooks by which SETS functionality can be invoked, and the GUI representations or widgets which are used to manipulate an open set. The first noticeable difference to a user of the modified Mosaic is the addition of a pull-down menu in the menu-bar under the heading of `Sets`. The pull down menu allows the user to toggle the use of SETS for form processing and inlined image retrieval. Using SETS for form processing means that the URL resulting from submitting a form should be used as an http-specific membership specification to open a set. Using SETS for inlined image retrieval means that inlined images should be accessed through the warden (to take advantage of the HTTP warden's prefetching of these images as discussed in Section 5.1.3).

A set is created in one of three ways. First, the user can create a set holding the objects referenced by an HTML document by clicking on the anchor to that document with the right mouse button. Second, the user can request a pop-up window either by clicking on a button on the bottom of the Mosaic frame, typing "g" ("s" and "S" were already used in the interface) in the Mosaic display window, or by selecting "Open a dynamic set" from the Sets pull-down menu. This pop-up window allows a user to manually enter a URL, which is used as an http-specific membership specification. Third, if the user has enabled the use of SETS for form processing, submitting a form will result in an open set.

Open sets are represented in two ways. First, when the set is created, a pop-up window (shown in Figure 5.1) appears which contains a number of buttons. Clicking on these



This window appears when a user opens a set. Clicking on the "Iterate" button causes Mosaic to get the next set element via `setIterate()` and to display it. Clicking on the "Digest" button pops up a window containing the list of the names of objects currently known to be members of the set, obtained via `setDigest()`. "Rewind" reinitializes the digest iterator to allow the member listing to be refreshed, "Size" displays the number of objects known to be members, "Close" closes the set (removing the window from view). "Dismiss" causes the window to disappear without closing the set, and clicking on "Help..." displays a window containing a brief introduction to the use of dynamic sets.

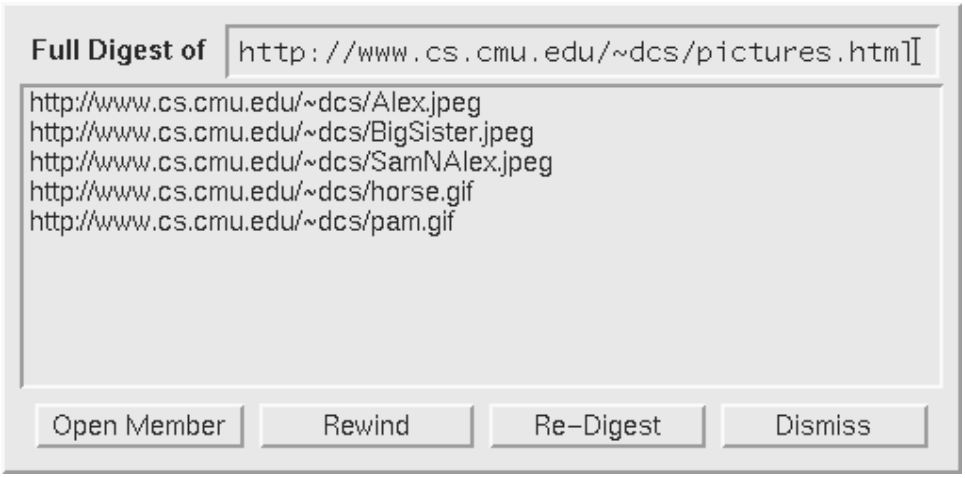
Figure 5.1: Mosaic Window for Managing Open Sets

buttons invokes various set operations. For instance, clicking on the "Iterate" button causes Mosaic to request and display the next element in the set. The caption of Figure 5.1 contains a description of the other buttons in this window frame. Second, the user can display the membership specifications of open sets by expanding the "Select an open set" option of the Sets pull-down menu. Clicking on one of these specifications causes the open set pop-up window to appear.

In addition to iterating on a set, a user can load members directly through use of the digest window, shown in Figure 5.2. The digest window displays a list of the names of the set members. The title of the window indicates whether this digest is complete or partial (SETS returned "...") by including the words "Full Digest" or "Partial Digest" respectively. If a partial digest is indicated, the user can update the digest window by reclicking on the "re-digest" button in the frame below the digest window.

An individual member can be loaded by double-clicking on its name in the digest window. Since loading members in this fashion does not employ the iterator, the user may suffer the latency of fetching the object. One enhancement would be to use a different color for names of objects that have been prefetched, which would inform users of which loads would be quickly satisfied and which would not.

Once a set member has been requested, either through the iterator or selected from the digest, SETS uses Mosaic's internal routines to read in and display it. This means that users can iterate over any type of object that Mosaic can handle, such as HTML, text, postscript, video, audio, and images (although the HTTP warden only will fetch objects accessible via HTTP).



This window appears when a user opens a set's digest. Note that the upper left corner reads "Full Digest", indicating this is a complete list of set members. If the digest were incomplete, this would read "Partial Digest". Double-clicking on one of the names, or clicking on one of the names and then the "Open Member" button will cause Mosaic to fetch and display that object. Clicking on the "Rewind" button resets the iterator to allow Mosaic to read in the membership from the beginning again, for instance to show the current weights on the objects. Clicking on the "Re-Digest" button will further expand a partial digest by calling the `setDigest()` iterator again.

Figure 5.2: Mosaic Window for Managing Set Digests

5.3 Conclusion

Dynamic sets sit at the file system interface between applications and the client subsystems of DFS. This chapter describes extensions that I made to these components in order to test the SETS interfaces described in Chapter 4. The experience, although subjective, does indicate that the dynamic sets programming model is indeed suitable to support search applications and is simple to program. The modifications to the Unix applications were straightforward, and it took only a few hours to modify several applications to use SETS. Modifying the Mosaic browser was more time consuming, primarily due to the complexity of the code and the difficulty of modifying a graphical user interface. The NFS and Coda wardens were each written in a few days. The NFS warden adds or modifies 379 lines of the 6887 lines in the NFS kernel code. Changes to the Coda Venus, although difficult to quantify exactly, represent less than one percent of the code in Venus.

Chapter 6

Prefetching and Resource Management

Up to this point, this dissertation has focused on the design and implementation of dynamic sets, whose value lies in the disclosure of hints of future access to the system. This document now addresses to the problem of how the SETS prefetching engine should exploit these hints. The solution to this problem is both critical to the demonstration of this thesis and sufficiently complicated to warrant further discussion. The complexity stems from trying to balance between aggressiveness and resource conservation. Aggressively prefetching drives up utilization and thus reduces the aggregate time to fetch a set of objects. However, increasing utilization too much can lead to overload, thrashing, and significant degradation of performance. Striking the right balance can be difficult. As an example, a study of prefetching in a parallel file system[44] found a wide range of possible outcomes from prefetching, from near optimal improvements to a reduction in performance equivalent to 20 times the optimal improvement.

The chapter begins with a brief overview of the engine. Then Section 6.2 provides background on the Unix FFS (Fast File System) buffer cache, which figures heavily in the design of the prefetching engine. There follows a discussion of the design rationale for the prefetching engine in Section 6.3, and a detailed description of the engine in Section 6.4. Section 6.5 concludes with a discussion of potential enhancements.

6.1 Overview of the SETS Prefetching Engine

The basic idea behind prefetching is to isolate from the application the work to fetch data in order to decrease the aggregate latency seen by the application to access a set of objects. Performance improvements come from fetching the data in a shorter amount of time, which thus increases the utilization of I/O channels. The dangers of prefetching are that increased utilization may lead to performance degradation, even if all prefetched data is

used by the application (none is wasted). Degradation can come from overloading certain media, such as Ethernet, which shows much lower bandwidth and higher latency as load approaches full utilization. Degradation may also result from increased contention, which can reduce performance by increasing the work that some unit must perform, such as the client CPU, or by delaying synchronous demand operations such as a page-out to disk behind asynchronous prefetch operations.

The most common way to avoid these pitfalls is to schedule prefetch operations carefully[67, 12] . As an example, at every point in time the prefetching engine used in TIP[67] estimates the savings that result from initiating a prefetch operation now, and compares them with the value of the resources that would be consumed by the prefetch operation. If the savings from prefetching exceed the value that is lost from consumption of these resources, the operation is initiated. These solutions rely on the ability to determine two pieces of information. First, they must be able to gauge the cost of I/O operations in order to know the savings, or the reduction in latency from a prefetch operation. Second, they must be able to predict the value that resources such as file cache space or bandwidth will have in the future. Since these prefetching engines have been proven to perform well under a variety of applications on the local disk, they demonstrate that it is possible to obtain these two pieces of information in the local file system.

Unfortunately, these solutions are unlikely to scale well to large distributed systems. As the system grows in size, the variance in the latency grows as well, making it difficult to predict the cost of an operation. Thus it is difficult to know accurately the savings that may result from a prefetch operation. It is also the case that a client in a distributed system has no knowledge of the load on the network or the servers, so it is unable to predict accurately the value of shared resources such as network bandwidth. Although it is conceivable to monitor the network for small LAN-based systems, the cost of doing so increases the overhead of prefetching. Even then, such a mechanism will not scale to GDIS.

In order to avoid these drawbacks of current prefetching engines, and to provide a mechanism that works well on this dissertation's range of target systems, I designed a new prefetching engine for SETS. This design avoids the pitfalls mentioned above by exploiting the semantics of dynamic sets and the I/O properties of search applications.

There are three forms of contention that can affect the application's performance: contention for the client CPU, network (including servers), and the client disk. SETS uses a different strategy to limit the impact that each of these forms of contention has on the application. To avoid increasing the load on the client CPU, SETS uses strategies with low computational overhead, trading optimality for lower complexity. To avoid stalling the application on the network, SETS ensures all of an object's data has been fetched before yielding the object to the application. To avoid stalling demand accesses to the local

disk, SETS carefully manages its use of the Unix FFS buffer cache to avoid unnecessary I/O wherever possible.

In addition, the SETS prefetching engine is designed to be dynamically tuned to adjust its behavior to suit a wide range of circumstances. The engine has a number of parameters which control how aggressively it uses resources. These parameters control the number of outstanding prefetches, the amount of local resources that can be spent on prefetching, and the rate of prefetching, and are described further in Section 6.4.

6.2 Review of Unix File I/O

Since prefetching involves the execution of I/O operations, one must first understand several details of the Unix file system before discussing the SETS prefetching mechanism. Although the Unix file system was discussed earlier in Section 4.1.1, several aspects that pertain to prefetching warrant more detailed discussion. The Unix buffer cache is the most relevant component, and is discussed in Section 6.2.1. Section 6.2.2 presents some other relevant details of the file system.

Files are sequences of logical blocks; logical block 0 contains the first n bytes, block 1 the next n bytes, etc, where n is the file system's block size. Each logical block corresponds to a range of sectors on the disk called a physical block; the physical block's number is the address of the first sector in the range. Readers desiring a complete discussion can read Chapter 7 of "The Design and Implementation of the 4.3BSD Unix Operating System"[47].

6.2.1 The Unix Buffer Cache

The Unix buffer cache, added as part of the 4.3 BSD FFS[53], exists to increase file system performance in three ways. First, it caches file data in physical memory to exploit locality in I/O requests. Second, it allows the use of delayed or asynchronous disk writes by holding written data in physical memory until the write completes. This results in two savings: eliminating synchronous operations from the critical path and enabling overwrites of cached data to eliminate writes of dead data. Third, it buffers access to files which allows many small I/Os from the application to be satisfied by fewer large block I/Os to the disk or network. This amortizes the high overhead of a disk access across multiple small I/Os. The basic operations in the buffer cache are described below and summarized in Figure 6.1.

The buffer cache consists of buffers, a hash table, and several free lists, the number of which depends on the implementation. Each buffer consists of a *buffer header* and a

Routine	Arguments	Description
`rwip()`	file offset, size	Performs file I/O by allocating buffers and reading/writing data from physical blocks.
`getblk()`	physical block number	Return a buffer to hold this block. If the block is cached, mark it busy and return it. If not, grab an unused block or evict the LRU block and use its buffer. Will pause if the buffer is busy or no buffers are available.
`bread()`	physical block number	Gets a buffer, reads the physical block into the buffer and waits for the read to complete.
`breada()`	two physical block numbers	Gets two buffers, reads the first physical block into one, starts asynchronous read of other block. Used to speed sequential reads to a file.
`bwrite()`	buffer	Starts a block write, waits for it to complete.
`bawrite()`	buffer	Starts a block write, but does not wait for it to complete. Buffer will be released by disk driver when the write completes.
`bdwrite()`	buffer	Mark buffer as dirty. This data will be written when this buffer is evicted or flushed.
`brelse()`	buffer	Called whenever an operation completes. Marks buffer as not busy and adds it to a free list.

Figure 6.1: Unix Buffer Cache Routines Relevant to Prefetching.

number of virtual memory pages; usually zero, one or two. Initially, all buffers have one page, but pages can be moved between buffers as needed. The buffers allow application file system I/O to be decoupled from disk or network access: applications read from or write to buffers, and the buffer cache routines move data to and from persistent storage to satisfy the application's I/O. The amount of data that is moved by each operation is called the file system's *block size*, which usually is set to 8KB.[1] Data is cached by holding

[1]The Unix file system also allows smaller blocks called *fragments* to be read and written, but fragments

valid data in buffers so that future accesses to the data can be satisfied without requiring a disk operation. To facilitate the location of a file's data in the cache, all buffers with valid data are placed in the hash table. The free lists hold buffers on which no I/O is currently pending. A *busy* buffer is one that is in use and does not reside on any of the cache's free lists, although it might be present in the hash table.

The number of buffers in the standard cache is fixed at boot time. Thus in steady state an incoming I/O operation must evict some cached data to free a buffer. The buffer cache evicts the *least recently used* (LRU) buffer if no unused buffer is available. The LRU buffer is the one at the head of the `AGE` list, or the head of the `LRU` list if the `AGE` list is empty. If no buffer is available (all have I/O pending on them), the incoming operation will block until a buffer becomes available. More advanced strategies such as those found in Sprite[58] or OSF/1[62] allow the buffer cache to grow or shrink dynamically by taking physical pages from the virtual memory subsystem when I/O activity is high, and releasing them when demand drops.

6.2.1.1 Buffer Cache Operations

When an application invokes a file system operation on a local disk file, the operation calls a low-level routine called `rwip()` to perform the I/O. I/O to files in distributed systems such as Coda use `rwip()` on the local cache container file, while other systems such as NFS[77] provide their own specialized version of `rwip()`. `Rwip()` translates the file offset and request size into a sequence of I/Os on the physical local (or remote in the case of NFS) disk blocks holding the file's data. For each block in the sequence, `rwip()` performs the I/O as described below. When the I/O completes, `rwip()` releases the block by calling `brelse()`, and moves onto the next block in the sequence.

The `getblk()` routine allocates a buffer to hold a physical block. The identity of a physical block consists of the device on which it resides and the reference of the block on that device, I refer to this tuple as the physical block number. `Getblk()` does a hash lookup of the desired block's number to determine if it is already in cache. If not, it gets an unused buffer, possibly evicting the buffer's data in the process. If it is cached, but the buffer holding the block is busy, `getblk()` blocks until the buffer becomes free. When `getblk()` has found a buffer, it removes the buffer from the free list on which the buffer resides, marks the buffer busy, and returns it to the caller.

To read data, `rwip()` calls `bread()` or `breada()`. `Bread()` first calls `getblk()` to get a buffer. If the data is cached, the buffer will contain the desired data and `bread()` is done. If not, `bread()` starts a read of the physical block(s) into the buffer, and then pauses for it to complete. After `bread()` returns, `rwip()` copies the data from the buffer

can be ignored for purposes of this discussion.

into the application's address space and releases the buffer. Since many applications read files sequentially, `rwip()` performs one block *read-ahead* using `breada()` when it detects sequential access to a file. `Breada()` initiates an asynchronous read of the next block in the file in addition to performing the synchronous read of `bread()`. Read-ahead is a form of prefetching, reducing latency to sequentially read a file from disk. However, `rwip()` cannot detect that a file is being read sequentially until the first two blocks have been consecutively read, thus read-ahead is only beneficial for files larger than 16KB (2 blocks).

When writing, `rwip()` first gets a buffer to hold the block. If only a portion of the block is being overwritten, `rwip()` must first read in the block's contents using `bread()`. If the entire block is being overwritten, `rwip()` can use any free buffer because all of the buffer will be overwritten by the operation.

After the application's data has been moved into the buffer, `rwip()` uses either `bwrite()`, `bawrite()`, or `bdwrite()` to copy the data to disk. `Bwrite()` performs a synchronous write: it starts a low-level write operation, and then blocks until the write has completed. When the I/O is completed, `bwrite()` releases the buffer. Synchronous writes are required of any application with strong consistency requirements, such as paging traffic, eviction of dirty buffers, and writes triggered by the `sync()` system call.

If the operation writes an entire block and the block's data has weak consistency requirements, `rwip()` uses `bawrite()` to perform an asynchronous write of the data. `Bawrite()` marks the buffer as asynchronous, starts the low-level write, but does not wait for it to complete. Since the buffer is still busy when `bawrite()` returns, `rwip()` does not release the buffer. Instead, when the I/O completes the disk driver detects that the buffer was marked asynchronous, and calls `brelse()` as a side effect.

When overwriting only a portion of the block, `rwip()` delays the write in the hopes that a future write operation will overwrite the remainder of the block. Applications that do not write whole blocks but do write sequentially will benefit from this batching of requests by reducing the number of disk I/Os that are performed. Delayed writes are performed by `bdwrite()`, which marks the buffer as *dirty* without invoking I/O on it. If the buffer is evicted before this block is written, the data will be forced to disk by a synchronous write and the buffer will be released. Every 30 seconds, or whenever the `sync()` operation is called, all the dirty buffers in the cache are written out to reduce the likelihood of data loss from system failure. Similarly, the `fsync()` operation writes out all dirty buffers associated with a particular file.

Whenever an operation is finished with a buffer, it calls `brelse()` to release the buffer. `Brelse()` marks the buffer to indicate it is no longer busy, and places it on a free list. The free list on which it is placed depends on the buffer's state. Most buffers are added to the tail of the `LRU` list. Buffers which contain garbage data are placed at the head of

the AGE list, and are thus the first to be reused when a new buffer is needed. Garbage data results from erroneous or aborted I/O operations. Buffer which have valid data that is unlikely to be reused are placed at the *tail* of the AGE list, and will be reused before any LRU buffers but after invalid ones. Examples of this occur when buffers that are used to satisfy a page fault are released. After the data has been copied, the data in the buffer is no longer needed since it resides elsewhere in memory.

6.2.2 Prefetching and the Unix File System

In addition to knowing the details of the buffer cache, it is also useful to make three observations about the BSD 4.3 FFS before discussing the SETS prefetching engine.

The first observation is that mapping between logical and physical blocks is expensive. Translation from logical to physical block numbers is done on every I/O by the bmap() operation. Bmap() performs this translation using a table stored in the file's metadata (inode). This table has the physical block number corresponding to the first NDADDR[2] blocks of the file, called *direct blocks*, and the numbers of disk blocks containing the mapping for the rest of the blocks, or *indirect blocks*. If the inode is cached, bmap() can translate direct blocks without needing to perform a disk read. If not, it must read in the inode, possibly evicting data from the buffer cache. If the block is an indirect block, additional I/Os to read in the indirect map might be required.

A second observation is that all file system data passes through the buffer cache as it is read or written. Local I/O uses the buffer cache as described above. Distributed file systems place data in the buffer cache as it is fetched, either storing it there directly as does NFS[77] or as a side effect of writing the data to the cache container file on the local disk as does Coda[82]. This makes the buffer cache the ideal location at which to add support for prefetching. First, no additional copies of data are needed, simplifying movement of data between the prefetching storage area and the buffer cache. Second, the task of locating prefetched data in response to a demand read can use the mechanism already in place to locate cached data. Third, prefetching can be implemented using normal demand operations invoked by the prefetching engine. The drawback of caching prefetched data in the buffer cache is that buffer caches tend to be small, limiting the amount of data that can be prefetched. Further, using the buffer cache to store prefetched data evicts other data, potentially slowing down other applications running concurrently on the same computer.

A third observation is the application's use of the file system API to access data makes it possible to efficiently track an application's progress in consuming data. Knowing an

[2] NDADDR is usually 12, which means the first 96KB of a file are stored in direct blocks for a file system with a block size of 8KB.

application's consumption of data allows the system to infer when a file's data should be prefetched, and when it is safe to release buffers holding prefetched data. Efficient tracking is possible because the file system API limits when and how data can be accessed. A file's data can only be accessed after a file has been opened and before it is closed. Additionally, if the file is read sequentially, progress within an open file can be determined by examining the open file's file pointer. This ability to track accesses is lost if the application were to use memory mapped files, since there is nothing equivalent to the file pointer to which the prefetcher has access.

6.3 Design Rationale

This section presents the design rationale for the SETS prefetching engine. Section 6.3.1 describes the goals of the engine design. Section 6.3.2 lists the assumptions made by the design. The design itself is presented in Section 6.4.

6.3.1 Design Goals

There are three basic goals of the SETS prefetching engine. The first is to offer good performance improvements. The second is to be as general as possible. The third is to minimize the difficulty of implementation.

The primary goal of prefetching in SETS is to decrease the I/O latency seen by applications that use iteration. SETS prefetches the data by loading it from its remote location to the local file system. I/O to the remote data is thereby converted to local file I/O, which should lower the latency seen by the application. It is also important that the engine minimize the performance degradation of applications that do not use SETS but run on a SETS-enhanced operating system.

To defend the claims of the generality of dynamic sets that were made earlier in the dissertation, the engine must also offer these benefits to a wide range of systems. Recall that SETS' target environment spans everything from palm-top to desktop computers, low bandwidth modem to high speed T3 connectivity, and local to world-wide distributed information systems. To perform well over this range of operating conditions, the prefetching engine must be both conservative and flexible.

The third goal is to minimize the implementation complexity. Efficiency calls for tight integration with the file system, but the difficulty of porting rises with the number of modifications to the file system code. Although portability is not a direct goal, tying the benefits of dynamic sets to Mach 2.6 is a strategically poor decision. As a result, the design avoids placing hooks in Mach unless the resulting benefits are overwhelming.

6.3.2 Design Assumptions

The design of the SETS prefetching engine is based on four simplifying assumptions. These assumptions allow a single mechanism to service a wide range of domains, providing these domains satisfy the four assumptions listed below. This section describes these assumptions, and argues why the assumptions are reasonable in the target systems of the dissertation: DFS and GDIS. In addition, it discusses the influence of the assumptions on the design, and where appropriate mentions how these assumptions can be relaxed.

6.3.2.1 Sets Fit on the Local Disk

The first assumption is that *the collective size of a set and its members may be bigger than the amount of available physical memory, but will not be bigger than available space on the local disk.* This assumption is reasonable for two reasons. First, most personal computers have limited amounts of main memory (RAM) due to its cost. Typical personal computers have 8MB to 16MB of memory, and modern operating systems require all of it to achieve reasonable performance. As a result, it is unreasonable to assume that a typical set will fit into RAM. Further, sets can be large: sets containing several megabytes of data are not uncommon. For instance, a set of 10 to 20 high quality images can easily take a megabyte. Second, single user computers have much more disk space than memory, sometimes several orders of magnitude more. This is due to the huge difference in dollars per megabyte of each medium. Given current trends, this difference is unlikely to disappear in the next decade. Although the example set's megabyte may not fit in memory, it is sure to fit on even a reasonably small disk (500MB by today's standards).

The chief implication of this assumption is that SETS should not rely exclusively on the local client's main memory to store prefetched set data. Using the local disk for this purpose will allow SETS to prefetch more information, which will result in lower latencies for SETS applications.

Although it is possible that a set may exceed the local disk, it is likely that the probability is small in practice. The time to fetch enough information to overwhelm the local disk is prohibitive given current network bandwidths.[3] For interactive search, users will probably be unwilling to wait for the data to arrive, and would probably refine the set to some smaller subset before processing the data. In addition, sets that are too large for the local disk can be broken into sufficiently small subsets and processed separately.

[3]Filling up one tenth of a 500MB disk takes 40 seconds on an Ethernet, around 15 minutes at typical Internet speeds (60KB/sec), and around 8 hours on a 14400bps modem.

6.3.2.2 High Variance in Response Time

The second assumption is that *the factors affecting prefetching performance are subject to a high variance*. This assumption is reasonable given SETS's goal of supporting search in a range of operating conditions. One factor is processing time. Different applications have substantially different processing requirements, and even within an application the time required to process an object varies depending on the object's type or construction. For instance, the time to process an `HTML` object is proportional to the number of `HTML` tags it contains, and depends less on the object's size. Another factor is object size. There is a large range of common object sizes, although the average object is relatively small[3, 27]. The range can be substantial even within object types. Other factors such as propagation delays due to the distance the data must travel, congestion on intervening networks, or server load can also greatly affect the time it takes to fetch data. These factors can change over time (sometimes quite rapidly), and may differ from search to search or from platform to platform.

This assumption is a primary reason why the existing informed prefetch methods such as the one used by Patterson et al.[67] are not used by SETS. These systems make prefetching decisions based on observations of current state. To keep overhead low these observations are maintained as averages. Unfortunately, high variances reduce the accuracy of the arithmetic mean as a predictor of the expected value of a distribution, such as the distribution of fetch times. As such, these observation-based estimates may lead to poor performance. SETS's approach is to use a tunable system that may be dynamically controlled, which provides acceptable performance although it may not be optimal.

6.3.2.3 Local Disks Are Faster Than Remote Servers

The third assumption is that *it is faster to access data on a local disk than to access it from a remote server*. Although this assumption is not true in many test environments where the server and network are lightly loaded, it is usually true in practice. First, propagation and protocol processing delays can exceed the delay of a typical disk access if the data is physically or logically distant. For instance, it takes light the same amount of time to cross the United States as it does to perform a disk access; adding protocol overhead and the return trip means accessing data from the other coast takes as long as 10 disk accesses on average[4]. Second, the request will often miss in a busy file server's buffer cache, causing a remote disk operation anyway. If the server is loaded, further delays may be added as the request waits in various free lists. Third, network collisions due to congestion can add latency, particularly when a link or hop on the path is loaded.

[4]Experiments show a disk access takes 15 milliseconds on average, while round trip time between CMU and the University of California at Berkeley takes 200 milliseconds

Fourth, all of these factors weigh more heavily as the load on the system grows. For these four reasons, this assumption is valid for many distributed systems, although it is not as reasonable for tightly coupled environments like parallel computers or clusters of workstations.

Given this assumption, it behooves SETS to continue prefetching from the network even when the amount of prefetched data exceeds the memory allocated for it. The overflow can be written to disk, and reread at a future point when all members of the set have been prefetched. For example, assume a set contains members stored at various WWW servers, and access to these members takes several seconds apiece due to propagation delays. After waiting several seconds, the first members start to arrive. Since the delay is due to propagation and not bandwidth, several of the members can arrive at once and fill the memory allocated to prefetching. When this memory fills, rather than discarding the incoming data and incurring another several seconds of delay to refetch it, SETS should write the incoming data to disk. In addition to lowering the latency, the variance seen by the application to access set members will also decrease.

What is the penalty of this assumption? In the worst case, performance is equivalent to reading information off the local disk, which is not significantly worse than the performance of a DFS in most cases. One can avoid the worst case by carefully managing how the file system handles prefetched data, for instance by reading prefetched data that has been evicted from the buffer cache back into memory before yielding an object to the application. In addition, applications whose processing rate is close to that of the rate of prefetching are likely to have high hit rates in the buffer cache, since prefetched data will be used before it can be evicted. If the performance is still unacceptable, one could modify the warden to refetch data from the server in those cases where server speed exceeds that of the local disk. This is in fact the approach taken by the NFS warden, described in Section 5.1.2.

6.3.2.4 Search Applications Sequentially Access Whole Files

The last assumption is that *applications read set members sequentially.* This means that when an application reads a set member's data it does so from the beginning of a file to its end without explicitly changing the file's file pointer. This assumption is true of most Unix applications[64, 3], and is also true of popular search tools such as WWW browsers like NCSA Mosaic[57]. In fact, the most popular WWW protocols[5] only have operations to read whole files, and so all WWW applications effectively read sequentially.

[5]HTTP and FTP consume the most bandwidth, based on studies of the NSFNET backbone from April, 1995, the last month such statistics were gathered. The statistics are available at `http://www.merit.edu/nsfnet/statistics/1995/nsf-9504.ports`.

Given this assumption, SETS can track an application's progress through a set member by observing the member's file pointer. SETS therefore knows exactly when a prefetched block can be released to free space for other prefetched data. It also means that SETS knows the beginning of a file will be requested first. SETS can prevent the last few blocks of a file from evicting the first few blocks from memory, since the first blocks will be needed before the last blocks are requested.

Fortunately, the cost of this assumption is small for those applications which do not read sequentially, provided that SETS does not aggressively consume buffers to hold objects' prefixes. The number of blocks SETS holds in memory is a tunable parameter, but should never be very large. Although these blocks could be used to better purpose if SETS knew the application's processing order, they cost little and will satisfy demand access if the whole file is read. SETS could alternately release an object's blocks into the `LRU` list when the object is opened to allow its buffers to be reused more quickly. However, this opens the possibility that a prefetched block is evicted before the application gets a chance to read it.

6.4 SETS' Prefetching Engine

The discussion of the design of the prefetching engine has four parts. The first part is an overview of the design, and was presented in Section 6.1. The second part, in Section 6.4.1, presents the key points of the design by answering five important questions. The third part is in Section 6.4.2, which describes a key aspect of the implementation of the prefetching engine: management of Unix's buffer cache for prefetching. The fourth part discusses additional considerations specific to the Mach 2.6 operating system, and is contained in Section 6.4.3.

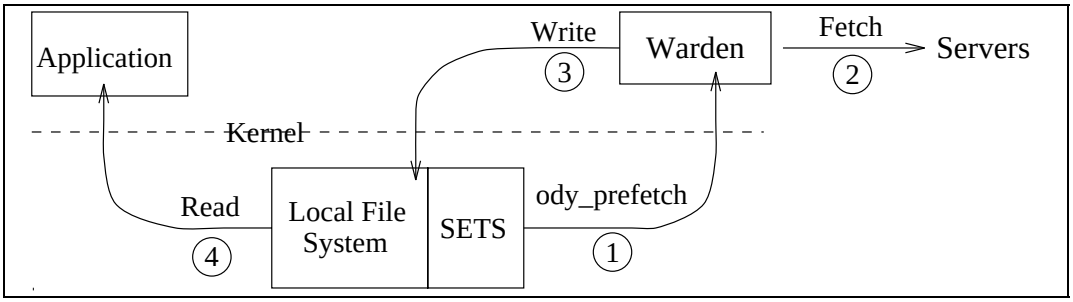
6.4.1 Engine Design

Although dynamic sets disclose hints of future access to the system, the proper way of using these hints to reduce latency is not immediately obvious. For example, the number of members may be large, and prefetching them at once can overwhelm the system. In addition, the application may not process all the members, and therefore prefetching them all may be wasteful. Perhaps the application finds the object for which it is searching before it has examined all members. Or perhaps it never opens a member, and instead creates a smaller subset on which to iterate.

The way in which SETS uses these hints is presented as answers to five critical questions. First, what are the semantics of the prefetch operation `ody_prefetch`, introduced

earlier in Section 4.3.3.1? Second, when should SETS start to prefetch a set's members? Third, how much should sets prefetch at once? Fourth, in what order should members be prefetched? Fifth, which member should be yielded when the application calls `setIterate()`? These questions are answered in the following five subsections.

6.4.1.1 What Does `ody_prefetch` Do?



This figure shows the local actions involved in prefetching. The process starts when SETS initiates a `ody_prefetch` operation. Next, the warden fetches the object from the server, writing the data into a cache container file on the local file system. Applications read the data out of this cache container file using the file system's `read()` operation. Arrow direction indicates the direction of requests. For instance, SETS sends an `ody_prefetch` request to the Warden, and the Warden performs a `Write` to the local file system.

Figure 6.2: Depiction of Local Actions Involved in Prefetching

A prefetch begins when the SETS prefetching engine invokes the `ody_prefetch` operation in the SETS/warden interface. Although the exact functionality of this operation depends on the warden's implementation, it must satisfy two properties. First, it must ensure that the entire object has been fetched before returning. Second, it must tag all local I/O it performs on behalf of the prefetch operation to enable SETS to manage these I/Os separately.

The first property has three benefits. First, it ensures that application accesses to prefetched data will not block on network I/O. Second, it forces remote accesses to be whole-file sequential. This enables the transfer to take advantage of streaming protocols such as those used by TCP[35] or SFTP[79] to reduce the overhead of communication, and is acceptable given the assumptions listed in Section 6.3.2. Third, it simplifies the management of an object's data.

Figure 6.2 shows the basic operations involved in prefetching. SETS first invokes an

`ody_prefetch` operation on an object. The request results in a fetch if the object is not cached. As the warden reads information off the network, it writes the data out to the cache container file for this object. These writes pass through the buffer cache, but are marked as prefetch operations to allow SETS to handle them separately from demand operations. After SETS yields the object to the application, the application can read the prefetched data using the file system's read operation.

6.4.1.2 When to Start Prefetching?

This question arises because some applications may not process a set's members. A search may create a set as an intermediate collection, to be merged or otherwise manipulated before the application processes the set's members. As such, it is not wise to aggressively start prefetching the members of a set as soon as the set is opened, since the prefetched objects may never be used.

By default, SETS conservatively waits to start prefetching until the application has indicated it is interested in processing the set's members. SETS infers the application's interest when the application first calls `setIterate()`. At this point, SETS queues prefetch operations for any files it knows to be members as a result of (partially completed) membership evaluation. If the set has no known members yet, SETS mark the set to indicate that aggressive prefetch is desired, and queues prefetch operations for new members as they are added. Applications like `grep` that begin to iterate on a set immediately after opening it will lose little opportunity to prefetch as a result of this decision.

For applications that open a set sometime before iterating on it, SETS provides an alternative behavior. An application can indicate to SETS that it will be processing the members eventually by setting `SETS_ANTICIPATE_ITERATE` flag in the `mode` argument of `setOpen()`. When SETS sees this flag, it marks the set so that the prefetching engine will begin to prefetch members as soon as members are added to the set.

6.4.1.3 How Much to Prefetch?

There are two reasons why this question needs to be answered. First, SETS needs to limit the bandwidth consumption of a client in order to avoid overloading networks and servers. Second, SETS needs to limit the likelihood of wasting a prefetch by fetching an object that will not be used by the application. Prefetching unused objects incurs costs by consuming resources, but does not offer any offsetting advantages.

SETS limits the bandwidth a client can consume as a side effect of storing prefetched data in the local file system: a client's rate of consumption of network bandwidth cannot

exceed the rate at which it writes the incoming data into the local file system. In steady-state, this rate is bounded by the disk speed. The peak rate is determined by the speed at which a warden can write data into the buffer cache. SETS limits the size of the bursts which can be written at this peak rate by limiting the number of buffers which can be used to hold prefetched data. This limit, how it is implemented, and the implications of it are discussed in Section 6.4.2.2.

SETS limits the number of wasted fetches by limiting the number of outstanding prefetch operations per set, where an outstanding operation is one that has started but whose object has not yet been yielded. This limit is a tunable parameter called `limitOpens`. When SETS starts prefetching, it will initiate up to `limitOpens` prefetch operations, and will then initiate another prefetch every time the application calls `setIterate()`. Since SETS only starts prefetching when it is sure the application will iterate on the set, the only way these outstanding objects will not be processed is if the application does not call `setIterate()` a sufficient number of times.

Limiting prefetching in this way strikes the right balance between aggressiveness and conservatism. By aggressively prefetching initially, SETS gets a number of objects that are ready to be processed, reaping the benefits of parallelism in the I/O subsystem and increasing the tolerance of bursts in the application's processing rate. At the same time, the number of wasted prefetches is bounded. A higher value produces a larger buffer with which to insulate the application from network delays, whereas a lower value reduces the number of objects that might be prefetched unnecessarily. In addition, this allows SETS to tune its prefetching to the rate of the application. If the application runs quickly, it will call `setIterate()` more often, and thus more prefetches will be running concurrently. If the application runs slowly, SETS need not be as aggressive and in fact will prefetch lazily to match the application's rate of member consumption.

Although limiting outstanding prefetches limits the number of wasted operations, it does not limit the amount of prefetched data that is wasted, since objects can vary in size. In practice, this should not be a problem since most objects tend to be small. Object size is self-limiting in many systems since the time to fetch an object is related to the size; people tend to avoid using objects that are too costly to obtain.

6.4.1.4 Which Member to Prefetch?

A chief advantage of using sets as the underlying abstraction is that they have no inherent order. Thus SETS can determine the order in which to prefetch the objects. To minimize the latency seen by the application, SETS should prefetch in order of increasing service time.

Unfortunately, it is difficult to know a priori how long a fetch operation will take, particularly if SETS does not know the attributes of a file other than its name. Obtaining these

attributes, such as the file size, is often nearly as expensive as fetching the file. Once the size is known, it may still be difficult to predict fetch times accurately since network and server performance can change over time. In addition, SETS must also take the weight of objects into account if the application has specified weights using the `setWeight()` operation.

SETS's solution is to prefetch objects in the order that they appear in the set member array, shown in Figure 4.7. Since this array is ordered by weight, the effect of SETS' approach is reasonably close to presenting the objects in the correct order. Objects which take a long time to fetch may be presented out of order, however this is exactly the behavior that `setWeight()` is supposed to exhibit.

Although one can conceive of many algorithms that are more clever than this one, they all have higher computational complexity or require network I/O. As stated earlier, this design favors simplicity in order to reduce competition for local resources. In addition, delaying the decision of what to fetch in order to gather additional attributes reduces the potential benefits of prefetching. Further, this simple approach seems to work well in practice, as born out by the experiments presented later in this dissertation.

6.4.1.5 Which Member to Yield?

A related issue is the order in which members should be yielded by `setIterate()`. Obviously, only local, cached, or prefetched objects should be yielded, since SETS cannot know which of the outstanding prefetches will return first. In addition, objects of greater weight should be yielded before those of lesser weight. But which should be yielded if there are several unyielded prefetched members of equal weight when `setIterate()` is called?

The key is to consider the delays involved in reading an object's data in addition to the delay to yield it. To minimize the read delays, SETS yields the object with the most data in the buffer cache. A secondary benefit of this strategy is that the processing of the selected object causes the most number of buffers to be released from the prefetch pool, creating space to hold other prefetched data. One could alternatively yield the object with the largest percentage of cached data, but doing so would negate this second benefit.

SETS tracks the amount of an object's data that is cached by keeping a per-object counter. When a buffer is allocated to hold the object's data, the count is incremented by the buffer's size. The count is decremented whenever the buffer is evicted. When `setIterate()` is looking for an object, it first locates the objects with the highest weight of any unyielded member, then yields the object of this weight with the most data in memory. This strategy satisfies SETS semantics since order within a weight equivalence

class is undefined, so ties can be broken arbitrarily. If the user has not specified weights, all objects are weighted equally and SETS will yield the member with the most data in memory.

6.4.2 Managing the Buffer Cache

Although SETS assumes that local disk access is faster than remote, forcing all reads of prefetched data to be disk reads would result in mediocre performance at best. Alternatively, putting unrestricted amounts of prefetched data into the buffer cache will force other data out, increasing the cache miss rate and decreasing the performance of all applications on the system.

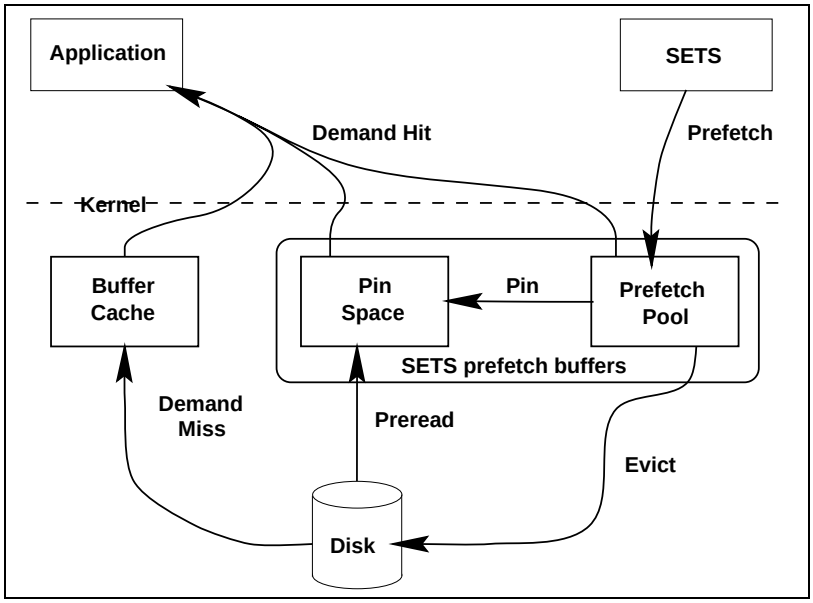
SETS avoids both of these problems by using data in the buffer cache where possible, and by restricting how much of the buffer cache can be used to hold prefetched data. It does this by maintaining a logical pool of prefetch buffers: buffers holding prefetched data. This pool is kept separate from the main buffer cache to prevent prefetched data from being evicted due to demand activity, but is integrated to allow demand reads of set members to find the prefetched data without requiring special hooks. SETS *pins* some of the buffers to prevent them from being evicted by other prefetches, so as to avoid disk I/O when these pages are accessed. To increase the hit rates seen by applications, SETS prereads evicted prefetched data from the disk as pin space becomes available. Figure 6.3 illustrates these actions along with the components involved with prefetching.

The following sections describe how SETS performs these actions in more detail. First I describe how SETS manipulates buffers used to hold prefetched data. Then I describe how SETS restricts the amount of space that can be used for this purpose. Finally, I describe how SETS increases the application's hit rate by exploiting the cache's state. The following sections also discuss the parameters that are used to control SETS' prefetching behavior.

6.4.2.1 Storing Prefetched Data in Buffers

Since prefetched data is stored in the file system until it is needed by the application, the information must pass through the buffer cache at some point. In order to better utilize the buffer cache, prefetched data is written into a logically separate pool of buffers. This allows SETS to keep track of how much memory is being used for prefetching, and to restrict the rate of prefetching by limiting the size of this buffer pool.

In fact, this pool is not separate from the buffer cache itself. Buffers are moved into this pool when they are used to hold prefetched data, and moved back into the main buffer cache when the data is used to service a demand read. Prefetch buffers are stored on a

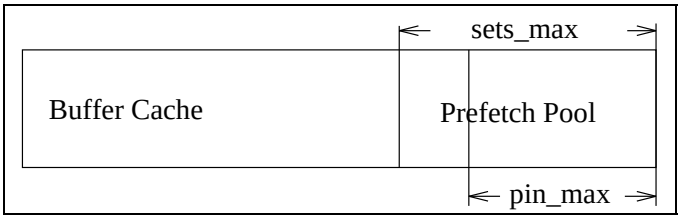


This picture illustrates how data moves through the buffer cache. SETS prefetches information off the network and writes it into buffers in the *prefetch pool*. After an object has been completely prefetched, its pages are *pinned* into the cache to guarantee their presence when requested by the application. If there is not sufficient space in the *pin-space*, the buffers may be evicted by writing their contents to the local disk. If pin-space becomes available, SETS can *preread* information into the pin-space to avoid a cache miss on that data. When the application demands data, it may *hit* in the buffer cache, pin-space, or prefetch pool and avoid a disk read. If the data is not present in the cache, it must be read from disk forcing the application to pause until the disk read completes.

Figure 6.3: Movement of Data in the Buffer Cache.

new free list called `PREFETCH` when not in use, although they remain on the buffer cache's hash table so that demand operations can easily locate them. Keeping the prefetch pool part of the buffer cache in this way allows auxiliary buffer cache operations like `sync()` to function on prefetch buffers without requiring the modification or duplication of their functionality. Figure 6.4 shows the relationship between this prefetch pool and the buffer cache.

When a warden services an `ody_prefetch` operation (described in Section 4.3.3), it opens the cache container file with a flag specifying this file will be used to hold prefetched data. This flag causes `open()` to mark the descriptor with the `IPREFETCH` flag. When an I/O operation on this file needs a buffer, it calls `sets_getblk()`. `Sets_getblk()` allocates a buffer using `getblk()`, but marks the buffer as a prefetch buffer by setting the flag `B_PREFETCH`, and indicates the operation is a prefetch by setting the flag `B_PREIO`.



This figure depicts the meaning of `sets_max` and `pin_max`. SETS limits the amount of the buffer cache that be used to hold prefetched data to the value in `sets_max`. In addition, SETS may chose to hold prefetched data in the cache (prevent eviction), but only up to `pin_max` data may be pinned in this manner. Note that this figure is not drawn to scale, `sets_max` should typically be set to roughly 10% of the buffer cache size to avoid unnecessarily evicting data from the cache.

Figure 6.4: Meaning of `sets_max` and `pin_max`.

Prefetch operations can result in either disk reads or writes. Writes come from wardens storing information fetched off the network to a cache container file on the local disk. When the warden writes to a file opened with `IPREFETCH`, `rwip()` sees this flag and calls `sets_getblk()` instead of `getblk()` so that the I/O will use a prefetch buffer. Reading this open file's data would also use prefetch buffers, although wardens do not read cache container files in practice. Instead, reads happen as a result of SETS prereading evicted data back into the cache. Since SETS initiates these directly, it knows to use `sets_getblk()`.

When the buffer is released at the end of the I/O operation, `brelse()` notices that the `B_PREFETCH` and `B_PREIO` flags are set. To prevent prefetch buffers from being evicted, `brelse()` puts prefetch buffers on the `PREFETCH` free list, which only holds prefetch buffers. Using a separate list prevents prefetch buffers from being evicted until both the `LRU` and `AGE` lists are empty, while allowing these buffers to be accessed by auxiliary buffer cache routines like `sync()`.

When the application reads the data in the buffer, i.e. a demand read, `getblk()` finds the prefetch buffer, and uses it to satisfy the read. `Getblk()` is able to find the prefetch buffer since it is still in the buffer cache's hash table. `Getblk()` clears the `B_PREIO` flag, since the current operation is not the result of a prefetch. When the I/O completes, `brelse()` discovers that B_PREIO is not set, and clears the `B_PREFETCH` flag. The buffer is now no longer a prefetch buffer, and `brelse()` places it on the appropriate free list as described in Section 6.2.1. Alternatively, SETS could force the buffer to be evicted more quickly by placing it at the head of the `LRU` list.

6.4.2.2 Limiting Buffer Usage for Prefetching

`# of workers`	The number of worker threads. Controls the rate at which membership is evaluated and objects are fetched.
`limitOpens`	This is the number of unyielded objects on which prefetch operations can have been started.
`sets_max`	This is the size (in bytes) of the SETS prefetch buffer pool. `sets_max` $\ll$ size of the buffer cache.
`sets_restart`	This is the threshold at which to restart threads that are waiting for free prefetch buffers. `sets_restart` $<$ `sets_max`
`pin_max`	This is the maximum number of bytes that can be pinned in the SETS buffer pool. `pin_max` $<$ `sets_max`.
`sets_count`	This is the count of how many bytes are currently stored in prefetch buffers. `sets_count` $\leq$ `sets_max`
`pin_count`	This is the count of how many bytes are currently pinned. `pin_count` $\leq$ `sets_count`.

Figure 6.5: Parameters and Counters Controlling SETS Prefetching Behavior.

To prevent prefetching operations from using buffers that are needed by the application or by other concurrently running applications, SETS keeps an upper bound on how much memory can be held by prefetch buffers. This limit is enforced by keeping a count of how much space is being used by prefetch buffers, adjusting the count as buffers are moved to the prefetch pool by `sets_getblk()` and are moved back by `brelse()`. This counter is called `sets_count`, and it and the other counters used by SETS are described in Figure 6.5.

When `sets_getblk()` moves a buffer into the prefetch pool, it first ensures that there is enough room in the pool by comparing `sets_count` to `sets_max`, which contains the maximum amount of space that can be used for prefetching. If the sum of `sets_count` and the new buffer's size is greater than `sets_max`, the operation blocks until sufficient space is available. If not, `sets_count` is increased by the new buffer's size.

When `brelse()` moves a prefetch buffer back into the main buffer cache, it decrements

`sets_count` by the size of the buffer. If `sets_count` is lower than the threshold stored in `sets_restart`, `brelse()` will awaken any sleeping threads since space is now available. SETS uses this threshold to keep from waking these sleepers unnecessarily.

Both `sets_max` and `sets_restart` can be adjusted dynamically, although algorithms for doing so are left as a future enhancement. The value of `sets_max` should be a fraction of the total buffer cache size to avoid lowering the cache's hit rate too drastically. The optimal value depends on the application mix that is running. `Sets_restart` should be at least 2 pages (8192 bytes) smaller than `sets_max` to guarantee a sleeping thread will continue. However, it is usually wise to have a larger difference to avoid possible stop-and-go behavior from adversely affecting performance.

6.4.2.3 Pinning Buffers for Faster Reads

SETS pins pages in the buffer cache to have greater control over what is evicted as newly prefetched data arrives. To understand the need, consider what happens when SETS is prefetching a set whose members are collectively larger than the prefetch buffer pool. After the pool has filled, incoming data will evict cached data. References to evicted data will then miss in the buffer cache, forcing the data to be read from disk. In the worst case, all application reads of prefetched data will miss, rendering the buffer cache ineffective.

To overcome this problem, SETS pins a portion of the prefetch buffers, preventing them from being evicted. These pinned buffers remain in the cache until they are required by the application. SETS uses two counters to restrict the amount of data that can be pinned. The counter `pin_count` keeps track of how much data has been pinned into the cache, while `pin_max` holds the maximum amount of data that can be pinned. Since only prefetch buffers may be pinned, and since some prefetch buffers should be reserved for prefetching, the relationships between the counters must obey these invariants:

$$pin\_count \leq sets\_count$$
$$pin\_max < sets\_max$$

Pinning of pages is done by the SETS worker thread just after the prefetch operation has completed. The worker first checks to see if there is sufficient room to hold the object. If there is not sufficient pin space available, the object's buffers remain in the prefetch pool and may be evicted. If there is space, the worker thread updates `pin_count`, grabs the object's buffers, marks them as pinned by setting the `B_PIN` flag, and then puts them on a per-set list. This per-set list holds all pages that have been pinned for that set. Demand access to a pinned buffer will remove the buffer from the per-set list. When the I/O completes, `brelse()` decrements `pin_count` and releases the buffer. To release pinned

pages that are not referenced by the application, SETS releases any pages remaining on a set's list when all references to the set are closed. Since pinned pages are prefetch pages, the release of a pinned page is also the release of a prefetch page, and so `sets_count` is decremented as described above.

Pinning in this manner has two advantages. The first advantage is that SETS pins the pages in the order they appear in the object. Since SETS assumes set members will be read sequentially, it is important that the first pages of a file be in-cache. Once these pages have been read, any out-of-cache pages will be brought into the cache automatically by Unix's read-ahead mechanism (described in Section 6.2.1) as the application reads the file. If other pages than the first were cached, the application would still suffer a cache miss and disk read in order to load the missing first page. A second advantage is that the unpinned portion of the prefetch pool can still be used to prefetch data off the network. One can view the remainder of the prefetch buffers as a pool to be used in steady-state to buffer incoming prefetch data as it is being written to disk. These buffers allow the warden to write small bursts of data without blocking on disk I/O.

To maximize the application's hit rate, the value of `pin_max` should be as large as possible. However, to ensure that network fetches do not stall on lack of available buffers, the difference between `sets_max` and `pin_max` should be large enough to hold a burst of data (TCP window size for instance). On typical machines in our environment, the rate at which data can arrive is not significantly higher than the rate at which it can be written to disk[6] Thus in common circumstances, the unpinned portion of the prefetch pool can be safely set at less than ten pages, to allow for bursts in the rate of incoming data.

6.4.2.4 Increasing the Hit Rate through Prereading

So far I have described how SETS handles data as it is prefetched off the network. But what happens if the data is local when the set is created? Alternatively, what happens to the prefetched data that was evicted from the unpinned portion of the buffer cache? Rather than forcing the application to read this data from disk, SETS *prereads* this data into the buffer cache.

As with any form of prefetching, SETS must take care that this prereading does not impede other I/O, most notably I/O due to network prefetches. Prereading can affect network prefetches in two ways. First, prereads consume disk bandwidth and move the disk head, both of which could slow disk writes of incoming prefetch data. Since dirty unpinned prefetch buffers must be written before they can be reused, this may

[6]Our Ethernet achieves an effective bandwidth which does not usually exceed 400KBps, based on experiments conducted between 2 DECstation 5000/200 connected by 10Base2 Ethernet. Disks can easily achieve similar bandwidths.

block network prefetches until buffers become available. Second, preread data consumes prefetch buffer space that could otherwise be used to hold data that has been prefetched over the network. Since SETS assumes local disk access to be faster than remote access, the network prefetches should take precedence over prereads.

On the other hand, it is important that SETS aggressively preread local objects if there is no incoming prefetching data. Consider a set that has one object which must be fetched over a 14.4Kbaud modem connection, and a number of objects that are on the local disk. The preceding paragraph argues that no local data should be preread until the remote object has been fetched, to avoid interfering with the remote fetch. However, letting the application read the local objects would allow the time to fetch the remote object to be overlapped with the application's processing.

SETS handles these conflicting demands by only prereading data into pinned buffers. This simple heuristic automatically balances these demands. Network prefetches are not slowed because there are more than the minimum $(sets\_max - pin\_max)$ number of buffers to hold prefetch data, insulating the prefetcher from disk stalls. Aggressive reads of local data will occur early in the processing of the set because no data is yet in the cache. In addition, this avoids wasting disk reads by ensuring that preread buffers stay in the cache. If they were evicted, the data would have to be preread again which would waste the first read.

To preread data that has been flushed, `setIterate()` invokes the preread of an uncached member if pin space is available. Hopefully the preread will have completed when the application next calls `setIterate()`, and the object's data will be waiting in the cache. Although SETS could be more aggressive in prereading, I feel this is unnecessary in practice because either the application processing rate is high enough to consume data as it arrives off the network or is low enough that more aggressive prefetching is not warranted.

6.4.3 Design Considerations Specific to Mach 2.6

Although the design just presented should work in any version of BSD, several features needed to be refined for the Mach implementation. In particular, Mach's buffer cache hash table uses a hash function based on physical block number, and provides no direct way of identifying to which file a buffer belongs. Mapping from a block to its file is difficult, and mapping from this file to the set of which it is a member is expensive. Essentially the former can only be done by calling `bmap()` for every logical block of every member of every open set to determine which belongs to the buffer in question. Further, finding the blocks of a set member involves calling `bmap()` and performing a hash lookup for every block in the file. Both of these problems would be alleviated by a file-indexed hash table, such as the one used in NetBSD[59].

The Mach implementation of SETS addresses this problem by only pinning an object's *prefix*, the first `nprefix` blocks of a file, where `nprefix` is a tunable limit. If the file is smaller than `nprefix` blocks, the prefix is equivalent to the whole file. In addition, SETS on Mach decides which object to yield using the number of the object's buffers that are pinned instead of the total amount of its data that is cached.

The main benefit of this approach is that the cost of mapping between members and buffers is substantially reduced by either avoiding the need for mapping or by doing it when it is least expensive. First, SETS only needs to map the blocks in an object's *prefix*, reducing the number of times `bmap()` is called per object. Second, these buffers are pinned and since pinned buffers remain in the cache, SETS must do this mapping at most once. If there is not sufficient pin-space when a member is prefetched, the mapping is deferred until the member's data is preread. Third, the per-member count of pinned buffers need never be decremented. Pinned pages are never flushed unless they are used, and they are never used unless the object has already been yielded. Once an object has been yielded, the count is no longer needed. Fourth, SETS usually pins the prefix immediately after the member has been opened by the prefetch operation, so the file's inode and direct block map should be in memory. Further, if the prefix blocks are all direct blocks (`nprefix` $\leq$ `NDADDR`), the mapping can be satisfied by the data in the inode. Thus the mapping can usually be performed without any disk I/O. (Recall from Section 6.2.2 that `NDADDR` is the number of direct blocks in a file.) If mapping is delayed until preread, the member's inode will be read in at most once (when the first buffer is mapped).

These savings come at a cost of a potentially lower hit rate for application reads. One reason for lower hit rates is the risk of a less than optimal iteration order. For example, SETS may overlook an object with many cached bytes but few pinned bytes in favor of another object with more pinned bytes, but with fewer total bytes in the cache. A second reason is that only an object's prefix is guaranteed to be cached, the other pages may have been evicted. Fortunately, there are also several ameliorating factors. First, the mean object size in file systems and in the WWW is less than two file system blocks[3, 27, 64, 78], so these problems do not exist for many objects if `nprefix` is bigger than two. Second, after the application begins to read the object, Unix's read-ahead mechanism (described in Section 6.2.1) prereads buffers automatically. Third, since there is a limit on how many buffers that can be pinned, pinning the prefix allows more files' data to be pinned. Thus even though the hit rate for a single file may be lower, the hit rate overall should remain high. Fourth, misses from suboptimal iteration can only occur if the buffers are evicted between the point the object would have been yielded and the point it is actually yielded. For reasonably large buffer caches, this is unlikely to be a common event.

However, it should be noted that pinning the prefix is not optimal. It could be the case

that competition for the disk head between the application reading disk blocks beyond the prefix and the prereading of other data can increase the average I/O latency per request. The increase comes if the blocks for the two files are stored on different cylinders, because the disk head is forced to seek on every request. If the reads for one file were performed at once, disk seeks would be avoided and the reads could take advantage of the layout used by the FFS to speed sequential reads.

6.4.3.1 Other Issues

In addition to the introduction of prefixes, the Mach implementation makes two other deviations from the design. The first deviation is that SETS forces all buffers out of the prefetch pool when all sets are deallocated. Flushing the prefetch pool is necessary because SETS cannot easily locate all buffers belonging to a set when it deallocates the set. This aggressive behavior is acceptable because there is no open set to which these buffers belong. This behavior is useful because it allows these useless buffers to be reused before the buffers in the `LRU` and `AGE` lists, allowing the data in those buffers to live longer. Since SETS does know about the pinned buffers belonging to a set, it forces the release of these buffers when the set is deallocated.

The second deviation is that SETS delays the writing of some dirty prefetch buffers until they are released. As described above, dirty prefetch buffers are the result of the warden writing data off the network into a cache container file. Normally, this data would be written asynchronously when the buffer is filled. However, all other accesses to this buffer block until the write completes. In particular, SETS's attempts to pin the buffer as soon as the prefetch has completed tended to block until the write is done. For this reason, SETS on Mach pins a buffer as soon as the write has completed, rather than waiting for the (whole file) prefetch to complete. In addition, SETS delays the writes of pinned buffers to avoid delaying the application should this member be yielded.

The chief disadvantage of delaying the writes is that doing so opens an opportunity for prefetched data to be lost if the system crashes. However, the window of opportunity exists already in Unix file systems, and is bounded by the `update` daemon's period, usually 30 seconds. The `update` daemon wakes once every time period and calls `sync()`, which writes all dirty pages in the buffer cache (prefetch pages included) to the disk.

One implication of delaying these writes, is that they will be batched together by the `update` daemon. The advantage of batching is that the writes can be optimized by scheduling them more efficiently. The disadvantage is that just after `sync()` the disk queues can be quite long if the pin space is large. A synchronous write that is started just after `sync()` completes may be forced to wait some time for service. However, one may be able to lessen this delay though more advanced scheduling techniques, such as

by scheduling synchronous operations in the disk queue before asynchronous operations. Such improvements are left as future work.

6.5 Future Enhancements

There are a number of ways in which the SETS prefetching engine could be improved, but that are left as future enhancements. For instance, one could design feedback mechanisms which dynamically tuned the prefetch engine's behavior to suit different levels of resource availability. The engine has been designed to be tunable by setting the parameters listed in Figure 6.5. Dynamic system adaptability seems like a fruitful area of research, and this prefetching engine is a promising place to start.

One potential drawback of this design is that SETS interleaves reads from concurrent prefetches to the same disk. This interleaving can destroy the benefit of clustering a file's data on the same cylinder, because it may move the disk head on every read. One could solve this problem by rewriting the SETS prefetching engine to reserve buffers and start I/O for a file or a portion of a file en masse. Thus a group of requests can be read without suffering a seek, and may also benefit from track buffering or other preformance enhancing mechanisms supplied by the disk hardware.

A related problem is that SETS does not prevent multiple prefetch operations from accessing the same disk. Although multiple accesses to the same server may be reasonable since they can overlap work on the network, server CPU, and server disk, multiple accesses to the same disk (client or server) can add contention and increase the cost of performing I/O. SETS could instead limit the number of outstanding requests per disk, although this may require additional interaction with the warden to determine where a particular object resides.

Chapter 7

Evaluation: Overview

The key assertion made by the thesis statement is that dynamic sets can reduce the I/O latency for a wide class of applications in a diverse set of domains. This and the three subsequent chapters defend the assertion by presenting experimental results that verify that substantial reductions in latency can be achieved in practice on real systems from using SETS. The chapter begins with a simple model of prefetching, to help the reader better understand the theoretical lower bound on I/O latency. It then describes the experimental methodology used to evaluate the performance benefits of dynamic sets in practice. The following three chapters present the results of experiments in three different domains: GDIS, DFS, and local file systems.

7.1 A Model of Prefetching

This section presents a linear performance model to aid the reader's understanding of the experiments that follow. The model is simple, which both increases its utility as an educational tool and also allows it to be easily modified to describe a range of systems. The model consists of a set of equations over non-negative integers that describe the performance of an application that serially opens a set of files, reads each file sequentially, and spends some amount of time processing each block of data it reads. Another set of equations describes the same application written to use dynamic sets. The equations, taken together, are a means of determining the expected amount of savings that dynamic sets offers to search applications.

The model is based on two sets of parameters: one set describes the system itself while the other describes the application. Figure 7.1 lists the system parameters. *Null* is the time to remotely invoke a null routine (a routine that performs no action), and is a measure of the overhead of making a remote procedure call. *Band* is the effective bandwidth of the

$Null$	Latency of sending a zero-length RPC. Includes CPU overhead on both client and server.
$Band$	The network throughput (bps). All routes to servers have the same bandwidth.
$Open$	Server component of latency to open a remote file.
$Read$	Server component of latency to read one buffer.
$BufSize$	File system buffer size.

Figure 7.1: System Performance Parameters

network connecting the client and servers. *Open* and *Read* are the times for the server to open a file and read a buffer respectively in response to an RPC. This time includes the computational overhead after the RPC processing has been done and any disk I/O latency that is involved. $BufSize$ is the block size of the file system, which is the unit of transfer for systems like NFS.

The second set of parameters, listed in Figure 7.2, describes the application and the set of objects it consumes. N is the cardinality of the set. $Size$ is the average file size. S is the number of servers storing members of the set, and is an indication of the number of files that can be fetched in parallel. Interactive applications have pauses in which the user reads an object's data, $Think$ is a per-file average of this time. For non-interactive applications like grep, $Think = 0$. $Comp$ is the amount of processing per byte performed by the application. Values for $Comp$ are typically small, but are likely to grow in the future due to computationally intensive applications such as image processing. The model assumes that there is no overlap between thinking and computing; computation is performed on a buffer while a file is being read and think time occurs on a file before the next file is requested. For instance, a WWW browser parses the data in an object before displaying it, but sits idle while the user is examining the data.

Equation 7.1 describes the latency of a remote open; Equation 7.2, that of a remote read. These equations describe a system in which an open does not return any file data and each read call returns one buffer's worth of data. One can easily construct similar equations for systems which return a small file's data when the file is opened, or that fetch the entire file as a side effect of the open, as is done in the WWW for example. Equation 7.3 computes the time to process one buffer, $CPUCost$.

N	The number of objects in the set.
$Size$	The size of the files in the set (number of buffers).
S	The number of independent servers storing members of the set.
$Think$	Time spent by user reading each file's data.
$Comp$	Time spent processing the data (per byte).

Figure 7.2: Parameters Describing Factors Which Affect Application Performance

$$OpenCost = Null + Open \tag{7.1}$$

$$ReadCost = Null + Read + \frac{BufSize}{band} \tag{7.2}$$

$$CPUCost = Comp \times BufSize \tag{7.3}$$

The time to process a set of objects without dynamic sets is denoted by $NoSets$, and the equation to derive its value is fairly simple. At the file level, there is no overlap since all file accesses are serialized. Equation 7.4 reflects this by summing the various costs in processing a file: opening the file, reading its contents, and processing the data. Determining the time to read a file ($Reading$) is more complicated because many systems allow asynchronous reads of a file's data to overlap intra-file I/O and processing. An example of an asynchronous reading mechanism is Unix's fast file system (FFS) sequential block read-ahead facility. Read-ahead speeds applications that read files sequentially by asynchronously prefetching block $i + 1$ as a side effect of reading block i. Equation 7.5 shows that the first read is synchronous, but that the system overlaps successive reads with client processing[1]. The $max()$ in Equation 7.5 captures this overlap behavior by only including the longer of the times to read and process data in the sum. Processing the last buffer cannot be overlapped with I/O since all of the file's data has been read, hence the last term ($CPUCost$).

[1]In fact, most Unix implementations require two blocks to be read in sequence before starting to read-ahead (only read-ahead when $i > 2$). To reflect this in Equation 7.5, the model should have 2 $ReadCosts$ in the sum, rather than just one as shown.

$$NoSets = N \times (OpenCost + Reading + Think) \quad (7.4)$$

$$Reading = ReadCost + (Size - 1) \times max(CPUCost, ReadCost) + CPUCost \quad (7.5)$$

SETS extends this concept of read-ahead to more general prefetching of files. As with sequential block read-ahead, SETS prefetching must synchronously fetch the first object, but can overlap successive fetches with application processing and user think time. This offers two advantages: the ability to read a file while another is being processed and the ability to read multiple blocks at once. In addition, SETS can fetch from S servers in parallel, further reducing the amount of I/O latency that is seen by the application. The reason for the S-fold speedup is that almost all of the time to fetch data is spent on the server; fetches to independent servers effectively happen concurrently. As a result, SETS can run S fetches in the time to fetch one file, eliminating the elapsed time for $S - 1$ fetches. However, this assumes that the network has sufficient bandwidth to sustain S fetches at once. For common networks such as Ethernet, this is a reasonable assumption if S is small. The experiments limit the number of concurrent fetches to five, except where noted.

The time to process a set of objects with dynamic sets is denoted by $Sets$, and is derived in Equation 7.6. Because SETS extends the concept of read-ahead, Equation 7.6 is similar in form to Equation 7.5, the equation for non-sets read-ahead. The first term is the cost to load the pipeline, the next term is the cost of executing the pipeline to completion, and the last term is the time to drain the pipe. The middle term of Equation 7.6 shows the result of parallel fetches, an S-fold reduction in latency.[2] Equation 7.7 shows the time to process the file, including user think time and client computation. Equation 7.8 shows the time to fetch the file. Note that SETS can overlap the time to open a file as well as the time to read it.

[2]In reality, SETS cannot perform concurrent I/O with no cost, since each operation does consume some small amount of the client's CPU which cannot be overlapped. However, modeling this effect would complicate the model without a corresponding increase in accuracy. Please remember that both division and subtraction are limited to non-negative integers.

$$Sets = Fetching + max((N-1) \times Thinking, \frac{N-S}{S} \times Fetching) + Thinking \quad (7.6)$$

$$Thinking = Think + (Size \times CPUCost) \quad (7.7)$$

$$Fetching = OpenCost + (Size \times ReadCost) \quad (7.8)$$

There are three effects that the equations do not take into account. First, the variance in some distributed systems is so high that representing some value such as $Band$ or $Null$ as an average introduces considerable error. However, the model is reasonably accurate, and has been validated by comparing it against measurements of a prototype implementation of SETS[92]. Second, SETS does consume some CPU, which increases the amount of computation that must be done while the application is running. However, SETS was designed to minimize this overhead, and fortunately does so: the experiments described below show the client CPU overhead to be small.

The third omission of the model is that it does not account for the fact that overlapping I/O requests can change the average cost of an I/O. On one hand, concurrent requests from different files to the same disk may destroy spatial locality in the request stream. Since the data is laid out on the server's disk for rapid sequential access to a file's data, SETS may see higher I/O latencies due to increased seek times. On the other hand, multiple outstanding requests can result in lower latencies by using server and network resources more efficiently. For instance, keeping the server's input queue full raises the server's utilization which reduces the aggregate latency of a stream of requests. Of course keeping the server fully utilized for long periods might adversely affect other users of the system whose requests are forced to wait in longer queues. One of the motivations for SETS prefetch parameters is to avoid this kind of overutilization of resources.

7.1.1 Implications of the Model

The chief benefit of this performance model is that it illustrates the basic advantages of prefetching. For instance, one can clearly see the benefit of parallel fetches: reducing the cost of latency by a factor of S. In addition, the benefit of overlapping computation and I/O is succinctly stated.

The equations also point out three limitations of SETS. First, the best possible performance SETS can achieve is to eliminate I/O stalls. If $Thinking \gg Fetching$ in Equation 7.6, there is little opportunity to benefit from prefetching; eliminating I/O stalls will have little impact on overall performance. Second, if all the members are remote when

the set is opened, SETS can at best eliminate all latency except the latency to fetch the first item. The optimal speedup occurs when the next file is ready to process as soon as *Thinking* on the previous one has completed. If the next file is not ready, SETS will stall up to an additional $\frac{Fetching}{S} - Thinking$.

The third limitation is that applications can already achieve some of the benefits of prefetching through read-ahead. Thus SETS must do better than read-ahead in order to offer any benefit. In circumstances where *ReadCost* is the most significant factor affecting performance, SETS may only be able to offer minimal improvements. However, the equations tell us when SETS can offer the most additional benefit over read-ahead: when $N > 2$, and when $S > 1$ or $Think > 0$. SETS also can offer benefits in situations where read-ahead is ineffective, such as systems that use whole-file caching. In these systems, all of the cost of fetching is contained in *OpenCost*, and the value of *ReadCost* is close to zero. By comparing Equation 7.5 and Equation 7.8, one can see that SETS obtains benefit from pre-opening files, whereas read-ahead cannot.

7.2 Experimental Methodology

The proof of the performance benefits of dynamic sets is based on the results of three experiments run on real systems. Each experiment consists of repeatedly running a particular search application workload on an existing and interesting information system, both with and without the use of SETS. The results of the runs are then averaged to determine the expected time to perform a search, and the number of runs depends on the experiment at hand. Comparing the elapsed time to run equivalent searches with and without dynamic sets gives a metric with which to determine if dynamic sets reduce the impact of latency on applications in practice. For the purposes of this dissertation, two searches are equivalent if they access the same objects and perform the same amount of (application or user) computation. Equivalent searches can, however, differ in the I/O latency they observe, or the order in which the objects are processed.

The following three chapters each describe an experiment which determines the benefit of dynamic sets in a different system domain. Chapter 8 describes an experiment that used trace-driven analysis to show the benefit of using dynamic sets to search applications on GDIS. The experiment described in Chapter 9 used a synthetic benchmark to examine the benefits of dynamic sets to search on a local-area distributed file system (DFS). Chapter 10 examines the impact of dynamic sets on search over data stored in a parallel disk array. By examining the benefits of dynamic sets in these different domains, the dissertation demonstrates the generality of dynamic sets in addition to their potential for improving the performance of search applications. Taken together, the three domains cover an important range of the kinds of systems on which search is done today.

Although they describe different experiments, all three chapters have the same basic structure. The chapter begins by identifying the domain, why the domain is of interest, and why it is different from the other domains. The chapter then describes the factors that affect search in that domain; the influence of factors may differ in different domains. Each chapter then describes and motivates the specific experimental methodology used by the experiment, and then presents the experimental results.

Each of the three experiments use the same basic experiment design, although the specifics may be different. Each experiment consists of a number of tests. The first test runs the application workload using the *base* factor levels to establish if SETS offers any benefits. The base levels comprise the common case for search in that domain. Successive tests then vary the levels for one of the factors. Collectively, the tests which comprise an experiment demonstrate the benefits of dynamic sets as well as the robustness of these benefits across the factor levels. Except where noted, the experiments flush set members from the cache to ensure independence of runs before running the tests.

7.2.1 SETS Cache Parameter Settings

Figure 7.3 contains the SETS cache parameter settings used by the experiments. In a few cases the experiment used different settings; the text accompanying the test results discusses the rationale and nature of the change for that test. The values were obtained by repeating the experiments for different parameter settings, and then choosing settings which seemed to work well across all three experiments. For parameters such as `nprefix` and `limitOpens`, performance dropped when the parameter was set too high or too low. For `sets_max`, however, the performance of SETS would increase for higher values, but the performance of other applications might drop. Although the experiments did not look at the impact of SETS on other applications in the system, the value of `sets_max` was arbitrarily set to be a small fraction of buffer cache size. The values of `sets_restart` and `pin_max` were chosen based on the value of `sets_max` and experimentation.

`limitOpens`	`sets_max`	`sets_restart`	`pin_max`	`nprefix`	# of Workers
5	2MB	1.93MB	1.75MB	8	5

These are the parameter values used in the experiments. `Sets_max` was $\frac{1}{8}$ of the buffer cache size for the WWW tests, and $\frac{1}{4}$ for the other tests (corresponding to buffer caches of 16MB and 8MB respectively). The values for `pin_max` and `sets_restart` were chosen to be slightly less than `sets_max`. `Nprefix`, `limitOpens`, and the number of workers were chosen to allow SETS to obtain some benefit from prefetching without overwhelming the system.

Figure 7.3: SETS Prefetch Cache Parameters

7.2.2 Metrics Used in the Evaluation

Each of the experiments uses five basic metrics to describe the performance of an application: user think time, client computation time, I/O latency, total elapsed time, and savings due to SETS. For all the tests, total elapsed time (Total) is measured by the application and does not include the time to create the application's process or to load the binary, since these activities are independent of the use of SETS. The manner in which the other metrics are measured depends on whether the experiment uses non-interactive or interactive search as its workload.

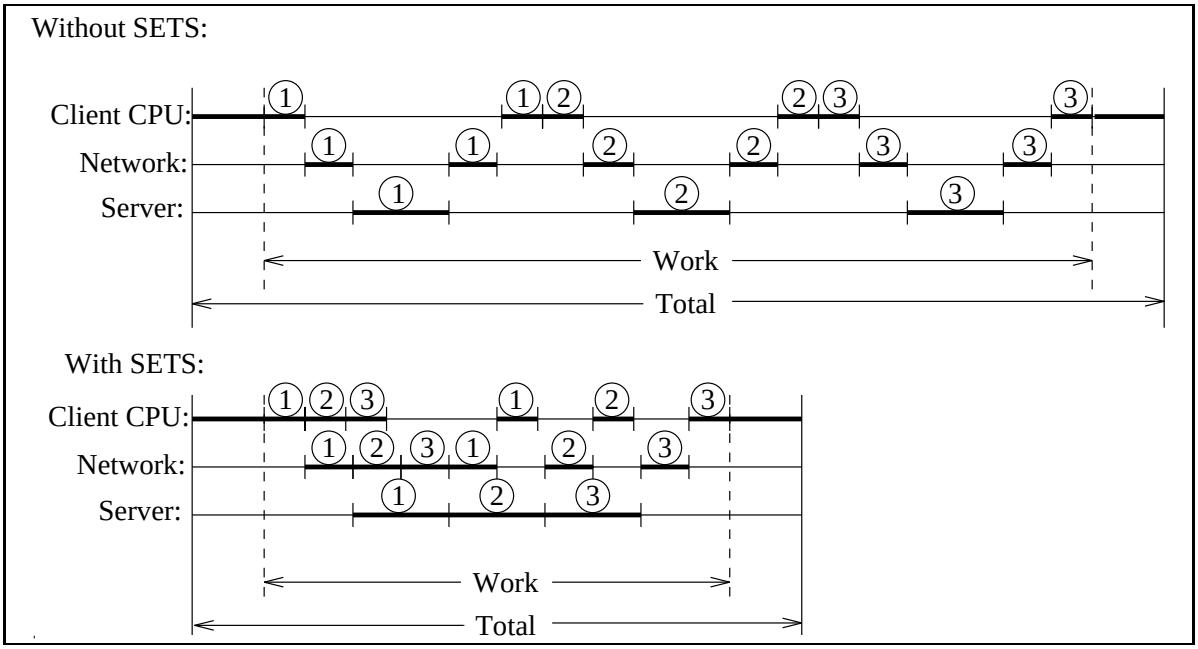
Experiments which use non-interactive search can accurately measure the amount of I/O latency by recording the amount of time spent in the idle loop on the client. The idle loop only executes when no other jobs are pending. For the experiments, most of the load on the client is from the application and SETS, so the idle loop runs whenever the application and all SETS worker threads are blocked. These experiments present 3 metrics in addition to Total: CPU, Stall, and Savings in Total.[3] CPU is the amount of time the client CPU was busy during the run, and is equal to the difference between Total and Stall. Stall is the amount of time spent in the idle thread. Savings in Total reflects the savings in runtime due to SETS; Equation 7.9 contains the formula used to calculate this metric.

$$Savings\ in\ Total \quad = \quad \frac{Total_{W/O\ SETS} - Total_{With\ SETS}}{Total_{W/O\ SETS}} \tag{7.9}$$

The experiments which use interactive search, however, cannot use idle time as a measure since the mechanism which simulates user think time causes the idle loop to run. Thus the Stall metric no longer reflects the amount of time spent waiting for data. Instead of having the system measure idle time, these tests have the application measure the time spent thinking or fetching.

Interactive search tests report four metrics in addition to Total: User, App, Fetch, and Savings in Fetch. User is the amount of time spent simulating user think time. App is the amount of computation performed by the application. Fetch is the amount of time to perform the I/O operations as seen by the application. Fetch thus includes some client CPU processing to setup and send outgoing messages and to handle incoming messages. As such, the theoretical minimum for Fetch for these tests also includes some amount of client processing, and thus will be higher than the potential minimum for non-interactive search. Savings in Fetch shows the percentage reduction in latency due to SETS, and is calculated using Equation 7.10.

[3] $Think = 0$ for non-interactive searches, and so user think time is not reported.



This figure contains two example time lines which show the work to fetch and process three objects with and without the use of SETS. Each time line is broken into three lines, one for the client CPU, one for the network, and one for the server. Thick lines show when the unit is busy, thin lines show when it is idle. Some of the busy periods are labeled 1, 2 or 3, to indicate that they constitute work performed to fetch one of the three objects. The solid vertical lines at either end of the time lines demark the time included in the "Total" metric recorded by the experiments, the dashed vertical lines demark the time included in the "Work" metric. This figure also demonstrates one benefit of SETS, the ability to overlap work on independent units in a distributed system.

Figure 7.4: Time Line of Events to Process 3 Objects With and Without SETS.

$$Savings\ in\ Fetch \;=\; \frac{Fetch_{W/O\ SETS} - Fetch_{With\ SETS}}{Fetch_{W/O\ SETS}} \tag{7.10}$$

There are two reasons why interactive search experiments present the amount of I/O savings due to SETS rather than the savings in total elapsed time. One reason is that Fetch time is a measure of how long the user must wait for information. Reducing fetch time reduces the duration of pauses that users see, thus increasing user productivity and potentially decreasing their annoyance in using the system. A second reason is that user think time is a large fraction of the total elapsed time for interactive search, and cannot be

controlled by SETS. As a result, including User in the I/O savings would underestimate the benefits of SETS.

In addition to these metrics, some DFS and local file system experiments report the time spent performing I/O, to demonstrate the effect (both positive and negative) that prefetching can have on the cost of I/O. Sources of this effect include higher device load and contention for shared resources like the server's disk. When $S = 1$, the Work metric is used for this purpose. Work records the amount of time that the I/O device (NFS server connection or local disk) is busy. Figure 7.4 illustrates the difference between Work, Total, CPU, and Idle. Total and Work are shown directly, CPU is the sum of the times that the client CPU is busy, while Idle is the sum of times it is not. Fetch measures the *apparent* latency (as seen by the application), Work measures the actual amount of time spent performing I/O.

When $S > 1$, a different calculation must be used to determine the amount of work time. Summing the busy times for all S devices overestimates the actual busy time since some of the I/Os run concurrently. Alternatively one could use the largest of the individual device busy times, but doing so would underestimate the actual busy time because not all I/Os are concurrent. Instead, the dissertation presents the measurement of work made by SETS when $S > 1$ (measurement of the time it took SETS to fetch a file). This measurement is less accurate for $S = 1$ because it includes additional computation, and is only valid for runs with SETS.

7.3 Preview of Performance Results

The following three chapters describe the experiments and performance results in detail. The key point of these results is that dynamic sets can reduce the latency seen by search over a range of factors on three very different systems. The GDIS experiment shows a 95% reduction in latency for interactive search is achievable when a set's membership is accurate. Experiments on DFS and local file systems show reductions in runtime for non-interactive search of 50% and 30% respectively, and in some cases, SETS is able to eliminate latency altogether.

In general, the benefit varies with the opportunity to prefetch and reorder. For example, larger values of S in the model allow a higher degree of parallelism, and higher values of $Think$ allow more latency to be hidden by user think time. Alternatively, a set of one member gives almost no opportunity to prefetch, and as a result SETS can provide no benefit. As stated previously, other uses of prefetching such as read-ahead can also limit the benefit one may obtain from SETS.

Chapter 8

Evaluation: Search on GDIS

There are two good reasons for exploring the benefits of dynamic sets to search on Global Distributed Information Systems (GDIS). First, GDIS are widely used and will continue to be important as they provide access to more data and as their user community grows. The most popular GDIS, the WWW, has seen an exponential growth in usage and traffic over the past several years[1], and is currently used by millions of people world-wide. Second, GDIS typically see much higher latencies than other distributed systems. In addition, the variance in latency tends to be high due to external load, differences in sub-network and server capacities, and large propagation delays.

This chapter describes an experiment which examines the benefits of dynamic sets to a particular GDIS, the World Wide Web. WWW searches are usually interactive, using a browser such as NCSA Mosaic to fetch and display candidate objects. One aspect of interactive search is that data consumption is done by humans, so the pause between requests (i.e. the time to process an object) can often last many seconds. Another aspect is the fact that browsers read an object's data sequentially, and process at most one object at a time.

The primary goal of this experiment is to demonstrate that the system can obtain significant reductions in aggregate latency by using dynamic sets. The experiment contains a base test to determine the benefit from sets in a common case, and a separate test to ascertain the sensitivity of these benefits to the primary factors listed in Figure 8.1. The time of day or day of week that a search is run can influence the amount of external load on the system, and thus the magnitude and variance in response time. Automatically loading inlined-images causes the browser to load much more data, and thus effects the aggregate latency seen by a search. Link bandwidth affects the response time of fetching large objects and reduces the opportunity to aggressively prefetch objects in parallel.

[1]From statistics of NSFNET backbone usage collected through early 1995 by Merit Network, Inc., available as ftp://ftp.merit.edu/statistics/nsfnet.

Factors	Name	Description
Primary	Time of search	External load on network and servers depends on when the search is performed.
	Image Loading	Fetch behavior of browser with respect to inlined images.
	Link Bandwidth	Bandwidth of client's link to Internet.
Secondary	Network Costs	Propagation delay, protocol overhead.
	Computation	Amount of computation taken by application.
	Prefetch parameters	Control prefetching and cache management, see Chapter 6.

This table lists the primary and secondary factors affecting the performance of interactive search on the WWW. The impact of the primary factors is examined through experiments, whose results are presented in this chapter. Primary factors have significant impact on search performance and can affect the benefit that dynamic sets offer. Secondary factors have a smaller effect on the elapsed time of the search or are independent of dynamic sets, and so are not directly examined by this experiment.

Figure 8.1: Factors Affecting Search Performance on the WWW

The choice to examine these factors, although arbitrary, is based on the performance model presented in Section 7.1 and an analysis of search on the WWW.

One question not addressed by the experiment is how widely users could employ dynamic sets during search on the WWW. This question arises because current WWW user interfaces only support one mode of interaction: point and click. Thus the WWW has no direct analogy to the use of wildcards in distributed file systems. Fortunately, there are many situations in which one could use sets if the browser were extended to support iteration. Essentially, any page in a hypertext like the WWW can be thought of as a set whose members are the objects referenced by the page. Section 5.2.2 offers an extension to Mosaic that is useful for iterating over most or all the objects a page references (such as a search engine's result page). One could also extend browsers to allow the user to select portions of a page, or use more complicated membership specifications such as executable functions which traverse some portion of the graph rooted at the page in question.

Although these extensions show ways in which dynamic sets could be used, demonstrating that users would employ sets in practice is much more difficult. As a result, this experiment is designed to show what benefits users would receive if they did use dynamic sets. Demonstrating that these benefits can be achieved in practice is left as future work.

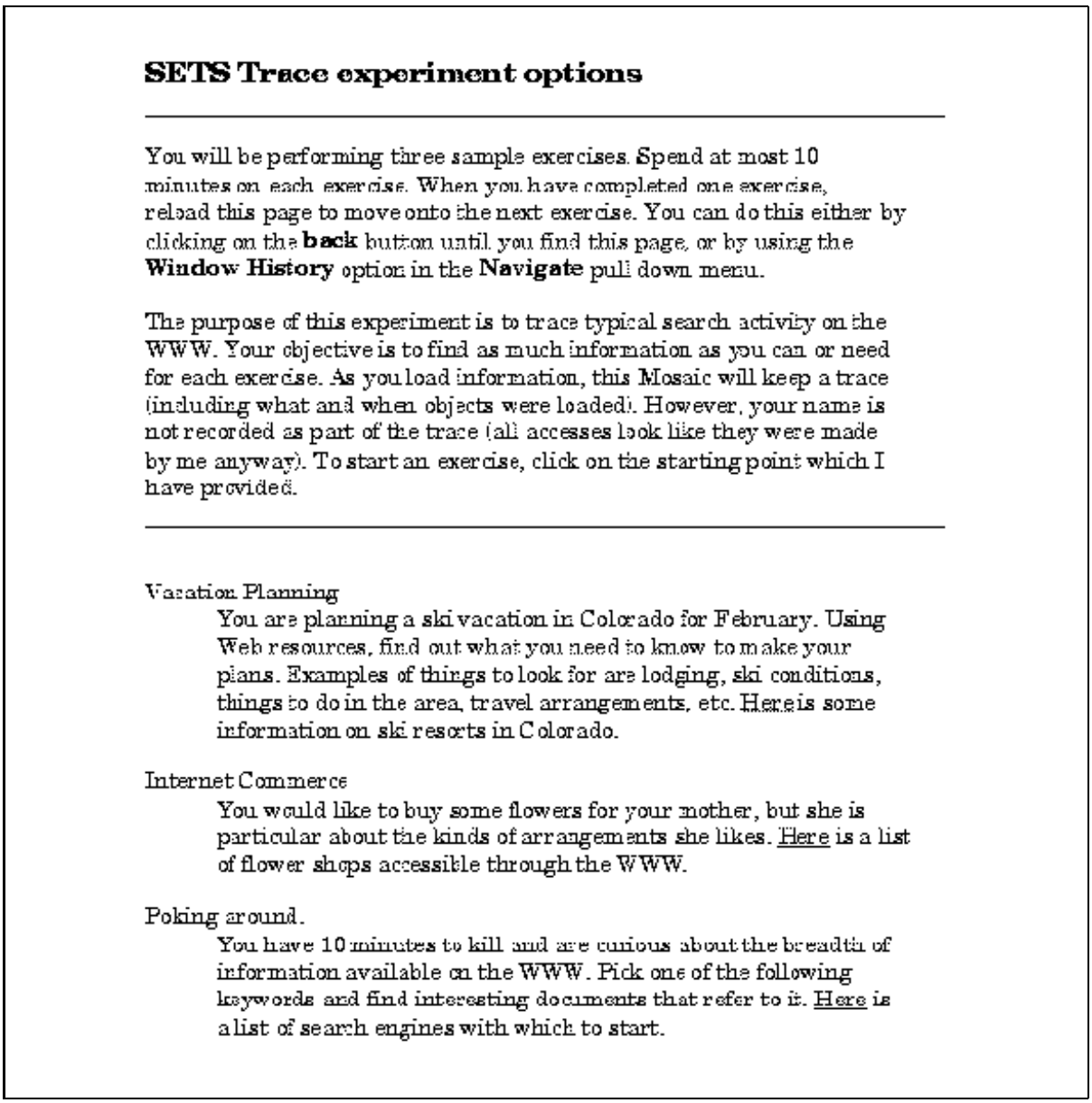
8.1 Test Methodology

In designing this experiment, I balanced between the conflicting goals of reproducibility and realism. Reproducibility is necessary for scientific rigor, but realism is equally important to demonstrate the benefit of SETS in practice. On the WWW, factors that add to latency such as load and non-uniform network topologies also add variance in latency. As a result, equivalent searches that access different objects or servers may see dramatically different performance, even two fetches of the same object may differ substantially in latency. Eliminating these factors to lower the variance reduces the latency and thus the realism of the results. To achieve a balance, this experiment uses traces of real searchers as a workload for reproducibility and replays these traces on the live WWW for realism.

To capture a range of reasonable behaviors and to avoid biasing the results of the tests by my own search pattern, the tests reported below are driven by traces collected from five volunteers. The traces capture roughly 30 minutes of search activity comprising three separate tasks. The volunteers are graduate students at Carnegie Mellon University, and are expert computer users. In addition, each of the volunteers has substantial experience searching for and retrieving information on the WWW, and is familiar with the Mosaic browser.

I informed the volunteers that the purpose of the exercise was to capture traces to be used as a workload to determine the benefit of dynamic sets. I told them that their activity would be logged, but assured them that their identities would remain private. Each volunteer started a modified version of Mosaic 2.6 which displayed the WWW page shown in Figure 8.2, which contained instructions on the three search tasks. The instructions also contained starting points for each task, although the users were free to use other information if they desired. Each user reloaded the instructions only when starting the next task, so a load of a page that contained instructions serves as a boundary between tasks in the traces.

The traces record the names of the objects that were fetched (including inlined images) and the times at which the fetches were requested by the user. By determining the time between the return of one fetch and the start of the next, one can obtain the amount of time the user spent examining the object before moving on. The 5 traces can be viewed as independent samples from the population of directed search activity performed by expert WWW users. Figure 8.3 summarizes the traces to give a better idea of the workload they represent.



SETS Trace experiment options

You will be performing three sample exercises. Spend at most 10 minutes on each exercise. When you have completed one exercise, reload this page to move onto the next exercise. You can do this either by clicking on the **back** button until you find this page, or by using the **Window History** option in the **Navigate** pull down menu.

The purpose of this experiment is to trace typical search activity on the WWW. Your objective is to find as much information as you can or need for each exercise. As you load information, this Mosaic will keep a trace (including what and when objects were loaded). However, your name is not recorded as part of the trace (all accesses look like they were made by me anyway). To start an exercise, click on the starting point which I have provided.

Vacation Planning

You are planning a ski vacation in Colorado for February. Using Web resources, find out what you need to know to make your plans. Examples of things to look for are lodging, ski conditions, things to do in the area, travel arrangements, etc. Here is some information on ski resorts in Colorado.

Internet Commerce

You would like to buy some flowers for your mother, but she is particular about the kinds of arrangements she likes. Here is a list of flower shops accessible through the WWW.

Poking around.

You have 10 minutes to kill and are curious about the breadth of information available on the WWW. Pick one of the following keywords and find interesting documents that refer to it. Here is a list of search engines with which to start.

This figure shows the HTML page that was displayed to users at the beginning of the tracing period. It describes three tasks to be performed by the user, and contains links to relevant pages to help the user get started. The starting points are, respectively, the home page of a Colorado ski index managed by AES Consulting, Inc., a list of 16 florist home pages that I compiled, and a list of links to 12 WWW search engines.

Figure 8.2: Trace Test Instructions

8.1.1 Replaying Traces

The experiments used a version of Mosaic that I modified to both capture and replay the traces. Bill Camargo at Transarc, Inc. designed and implemented the trace capturing

	Trace 1	Trace 2	Trace 3	Trace 4	Trace 5
Think Time (sec)	829.47	794.14	1203.92	997.25	902.64
Fetch Time (sec)	1053.06	1109.94	721.82	758.91	625.89
Total Time (sec)	1882.53	1904.08	1925.74	1756.16	1528.53
Events	351	413	391	1959	496
Fetches	51	39	44	69	52
Errors	1	1	2	1	0
Images	68	83	50	96	73
Object (bytes)	225015	126552	185677	367290	295125
Image (bytes)	414484	302387	254251	546751	300729
Total (bytes)	639499	428939	439928	914041	595854
Servers	19	12	11	24	19

This table characterizes the 5 traces used by the WWW experiment for non-sets workload. Fetch time is the total amount of time spent fetching and displaying objects, think time is the amount of time between requests for objects. An event is the fetch of an object or an inlined-image, or the use of a cached image. The much higher number of events in Trace 4 is due to a few pages which include the same 3 images over 1300 times. Fetches is the number of objects that the user attempted to access, errors indicate the number that resulted in errors such as HTTP error 404, object not found. Byte counts only include successfully fetched objects. The last line indicates the number of different servers that were accessed by the user.

Figure 8.3: Characterization of Traces Used by the Experiment

mechanism, and I extended it to capture additional information and improve its timer granularity. This version of Mosaic also contains extensions to support dynamic sets, as described in Section 5.2.2, and a mechanism to replay the traces.

To replay a trace, Mosaic reads in a trace, fetches the objects listed, and writes a record of each event to a log file. To lower the overhead of replay, these log events are buffered in memory, and the perturbation of the results by logging is minimal. The record includes the type of event, the identities of WWW objects accessed, the elapsed time, and the amount of time Mosaic was forced to block on I/O. Events include fetching WWW objects and inlined images, Mosaic window history operations (e.g. moving forward and backward), and think events. A think event is a pause whose duration approximates the user think time recorded in the trace. Mosaic approximates these pauses by repeatedly sleeping for 100 milliseconds using the Unix `select()` facility until sufficient time has elapsed. Experimentally, this sleep mechanism is accurate to within 1% over the length of the trace. The pauses to sleep during replay do consume some CPU, unlike the pauses for user think time in the original search. This leads the replay to slightly underestimate the benefit of dynamic sets by adding contention for the CPU, but this effect should be

small.

The advantage of this trace replay mechanism is that it induces a repeatable and realistic workload on the system. In addition, the latency seen to fetch the objects named in the trace depends on the state of the system in which the trace is replayed. As a result, I can replay traces on different system configurations to determine the effect of these configurations on the performance of search.

8.1.2 SETS Traces

Because the goal of the experiment is to understand the performance benefits of dynamic sets, the workload for SETS must be equivalent to the workload imposed by the user traces. To achieve this, I created copies of these traces replacing demand load operations with calls to `setIterate()`. Both the original and copy traces induce the same workload – the copies preserve the user think time and fetch the same objects as the original traces. As a result, comparing the results of replaying the modified and unmodified copies of a trace directly show the benefits of using dynamic sets.

In order to induce the same workload, the modified traces access exactly those objects accessed by the original traces. To achieve this effect, I created 15 HTML pages corresponding to the 15 traced tasks (3 per user). Each page contains a link to each object loaded by the trace for that task. The modified traces open one set per task, using the corresponding HTML page to define set membership. The traces then iterate once for every member of the set and close the set before moving on to the next task. Non-task loads of instructional pages such as the one in Figure 8.2 are left as demand loads to represent activity that might occur between searches in real life. The trace events that invoke `setOpen()` are placed immediately before the first call to `setIterate()` and involve no extra user think time so SETS gets no opportunity to reduce the latency of the first fetch through prefetching. Figure 8.4 shows the number and size of the objects for each set used in the experiments.

It is important to realize that since the creation of these SETS traces employed an oracle to determine set membership, this experiment provides an upper bound on the benefit one would expect from using dynamic sets. If one can exactly capture one's near-term future data needs, such as by iterating over the results of a query to a search engine, then one should see performance improvements comparable to the results shown below. However, the benefits from dynamic sets do depend on the user iterating over a set of objects whose membership they define. The benefits shown by the experiments below are only achievable in practice to the degree one sticks to this mode of operation.

		Trace 1		Trace 2		Trace 3		Trace 4		Trace 5	
Set		No.	Bytes	No.	Bytes	No.	Bytes	No.	Bytes	No.	Bytes
1	Obj	21	127583	17	46502	20	121427	10	93930	16	121352
	Img	12	104057	15	36117	9	16648	18	64712	6	18167
2	Obj	13	31086	3	37028	12	28133	39	67379	20	73726
	Img	30	229460	12	233084	22	127575	48	402729	46	107287
3	Obj	12	54702	15	31378	9	24473	16	201178	12	87396
	Img	26	80967	56	33186	19	110028	30	79310	21	175275
-	Obj	5	11644	4	11644	3	11644	4	4803	4	11644

This table shows the cardinality and size of the sets used by the experiment for the SETS workload. Objects accessed by a task were assigned to one set, objects accessed outside of tasks (e.g. the instructional pages such as the one in Figure 8.2) were left as demand loads and are shown in the last row. The sums of the bytes for each trace are equal to the amounts listed in Figure 8.3. Some discrepancies in use of instructional pages between traces are due to a bug in Mach 2.6 that caused the first load from www.cs.cmu.edu to fail. This in turn caused the user to retry the request, which is why the number of loads is sometimes greater than 3. The user in Trace 4 did not load all of the instructions I provided, which is why the demand load byte count is lower than the other traces.

Figure 8.4: Characterization of Traces Modified to Use SETS

8.2 Results of WWW Experiments

The following sections present the results of replaying the 10 traces on the live WWW. The traces were replayed on a DECStation 5000/200 with a 25Mhz R3000A processor and 64MB of RAM. The client software is version 2.6 of NCSA Mosaic modified to replay traces and use dynamic sets, and the client operating system is Mach 2.6. The SETS cache parameters used for these tests are listed in Figure 7.3. All experiments are run on cold caches.

The following sections present the results of the tests that comprise this experiment. The first test examines the benefit of dynamic sets in the base case, successive experiments vary one of the factors while holding the others at their base levels. Different search behaviors are captured in the 5 traces, all of which are used by each of the tests. The base level for load is obtained by running the experiments during weekday afternoons, typically the time of peak load (see Section 8.2.2). The base browser behavior is to automatically fetch and display inlined images. The base network bandwidth is LAN connectivity to the Internet. The client machine was directly connected to CMU's 10Mbps Ethernet, and connected to the Internet via a 45Mbps T3 line which was shared with other machines at Carnegie Mellon University, University of Pittsburgh, and the

Pittsburgh Supercomputing Center. This level of connectivity is typical for universities or medium to large corporations.

The tables in the following sections present the results of replaying the original and modified traces. The tables contain the average elapsed times in seconds (Total) to replay the traces for five runs of each trace, with standard deviations in parenthesis. They also present the portions of time Mosaic waited for data (Fetch), computed to display information (App), and emulated user think time (User). The fetch and think times are directly recorded in the logs, the App time is inferred by subtracting the fetch time for an object from the total time to fetch and display that object. The "W/O SETS" are the times to replay the unmodified traces, the "With SETS" numbers are the times to replay the traces modified to use dynamic sets. Figures 8.6 and 8.13 contain bar graphs which show the results of the four experiments.

8.2.1 Determining the Benefits of Dynamic Sets on WWW

Trace	SETS	User seconds	App seconds	Fetch seconds	Total seconds	Savings in Fetch
1	W/O	826.00 (0.9)	63.40 (4.4)	775.20 (116.9)	1664.60 (121.3)	
	With	826.10 (0.4)	62.40 (11.4)	40.10 (12.2)	928.80 (23.2)	94.83%
2	W/O	787.50 (0.1)	42.30 (4.3)	381.70 (139.5)	1211.50 (141.9)	
	With	787.60 (0.1)	41.80 (1.8)	41.90 (11.4)	871.50 (10.1)	89.02%
3	W/O	1203.30 (0.3)	46.60 (4.3)	509.50 (74.5)	1759.50 (70.9)	
	With	1203.20 (0.4)	48.70 (3.8)	75.80 (19.0)	1327.70 (20.5)	85.12%
4	W/O	957.20 (0.2)	84.30 (1.9)	641.30 (113.3)	1683.00 (112.9)	
	With	952.80 (0.2)	90.00 (3.4)	8.20 (4.2)	1051.10 (4.4)	98.72%
5	W/O	900.20 (0.3)	69.20 (7.6)	681.40 (120.9)	1651.00 (119.3)	
	With	893.10 (0.5)	67.20 (4.6)	29.30 (7.5)	989.70 (5.9)	95.70%

This table shows the tremendous savings potential of using dynamic sets in the base case, and that the savings are entirely due to savings in latency. The table presents the average aggregate elapsed time in seconds to replay the traces on the live WWW. The numbers for each trace are the averages of five runs, the standard deviation is presented in parentheses. Section 7.2.2 describes the metrics User, App, Fetch, Total, and Savings in Fetch, Section 8.3 describes the traces and replay mechanism. The 5 and 7 seconds difference in user time for Traces 4 and 5 are due to a transcription error in the modified traces. Since this error biases against SETS (less time to overlap) and is less than 1% of the overall think time, I chose not to fix the bug and repeat the experiments. Figure 8.6.(a) contains a graph of these results.

Figure 8.5: Results of Replaying Traces on the WWW

The goal of the first test is to establish the benefit of dynamic sets in the WWW. Figure 8.5 presents the results of this test, which are graphed in Figure 8.6.(a). Several observations are apparent from examining the numbers. Foremost is the difference between the elapsed times with and without the use of dynamic sets. Even in the closest case, Trace 2, the difference is greater than 2 times the variance. The benefit from dynamic sets is therefore statistically significant and there is a clear advantage to using dynamic sets, although the benefits depend on the contents of the set and where the objects are stored.

Further examination of the numbers in Figure 8.5 show that almost all of the difference in total elapsed time is due to savings in fetch time from using dynamic sets, the difference for the App and User times are negligible. The difference in App times is due to the competition between the prefetching engine and Mosaic, variance is due to background activity on the client. The difference in think times for Traces 4 and 5 is due to minor transcription errors introduced to the SETS copy of the traces by the author, as explained in the figure caption.

The difference in fetch times is dramatic: the user (in this case emulated by Mosaic) is forced to wait an order of magnitude more time for data to arrive when SETS are not used. The difference is due to prefetching and reordering. Prefetching allows fetching from independent servers in parallel and overlapping I/O with the considerable pauses for user think time; reordering allows SETS to hide longer fetches with the processing and user think time of other objects.

Another observation is that the aggregate fetch times exhibit a high variance. Most traces show a coefficient of variation ($\frac{\sigma}{x}$) in the neighborhood of 30%, although one case is as high as 51%. The high variance comes from Trace 3 with SETS, where $\overline{x} = 8.2$, $\sigma = 4.2$. This higher than normal variance is due to a trial in which a single fetch of an instructional page from CMU's WWW server contributed 12.4 of 15.77 seconds of aggregate load time. Without this trial the mean aggregate load becomes 6.4 and the standard deviation 2.2. Although the variance due to the CMU server is an artifact of the methodology, it is interesting because it demonstrates the variation in fetch time due exclusively to server load (other network factors are negligible in this case), as well as demonstrating the possible range in latency from fetching the same page.

This kind of variance in response time is quite common on the WWW. When one examines the trace replay results in detail, one can see a wide distribution of response times for fetching a single page. The source of this variance is likely to be due to network load in addition to server load. However, there may be other sources for latency as well. One possibility is the presence of bugs in the server which cause connection durations many times normal. Traces with very long aggregate fetch times tended to exhibit a few individual fetches which took hundreds of seconds to complete. Unfortunately, it is very

difficult to locate the source of this abnormality, since it occurs non-deterministically and since I do not have administrative access to the servers used by the test.

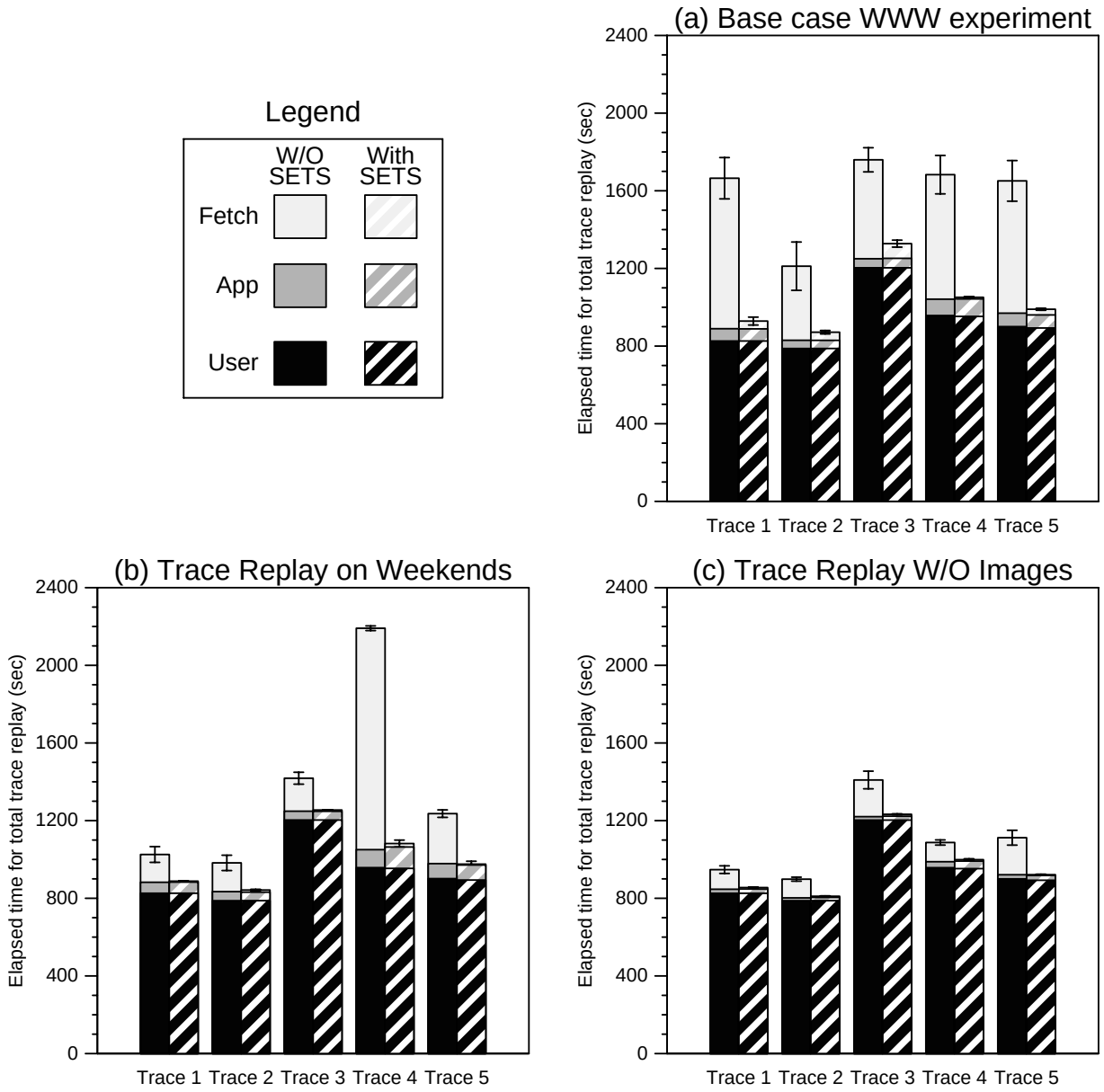
8.2.1.1 Effect of Aborting Long-Running Fetches

One potential source of variance in the results in Figure 8.5 is the presence of fetches which take several hundred seconds to complete. In actual usage, the user is likely to abort these fetches after some period of time. To examine the impact that such aborts might have, I postprocessing the trace replay logs to recalculate the Fetch times in Figure 8.5. Figure 8.7 shows the results of these recalculations, which truncate the time to fetch an object and its images to some maximal value before adding this time to the sum.

For the unmodified traces, the result of post-processing is equivalent to the user aborting the fetch after so many seconds had elapsed. However, this method underestimates the benefit from SETS, since the logs only record the pauses seen by Mosaic. The HTTP warden may have spent more than the maximum number of seconds fetching an object, consuming valuable resources all the while. If the warden had truncated the fetches itself, it could have used these resources to prefetch other data, further lowering the aggregate latency seen by Mosaic.

Figure 8.7 shows how the fetch times in Figure 8.5 would look if fetches had been aborted at 30, 60, 90, and 120 seconds. One effect is a lower average aggregate fetch time, particularly for the non-SETS cases. However, it is important to realize that there is still a significant benefit from using SETS, above an order of magnitude in some cases. A second effect is the substantial reduction in variance which results from aborting excessively long fetches.

The aggregate latencies are lower because the long fetches have been eliminated. However, aborting these fetches does result in loss of data which could affect the outcome of the user's search. Figure 8.8 shows the extent of data loss by indicating the average number of fetches that exceeded the limit and were aborted. One observation is that the number of aborted fetches for SETS is much lower, since prefetching is able to hide the long fetches most of the time. This implies an additional benefit of using SETS: the magnitude of the variance is much lower with SETS, although the coefficient of variation may still be high. Thus the fetch times are much more predictable, and the system less annoying, as a result of prefetching. A second observation, made by comparing the number of aborts in Figure 8.8 with the number of fetches in Figure 8.3, is that the likelihood of seeing a longer than normal fetch on the WWW without SETS is small, but not insignificant. The highest ratio of aborts to fetches is 14% (for Trace 1), the lowest is Trace 2 with 5%. However small the percentage, the value of these missing pages can only be known to the searcher at run time.



These graphs show the results of the first three WWW experiments, and demonstrate that the potential benefit of SETS is robust across a range of factors. Graph (a) shows the base case, described in Section 8.2.1 and shown in Figure 8.5. Graph (b) shows the results of replaying the traces during the weekend, and is discussed in Section 8.2.2 and shown in Figure 8.10. The high load time for Trace 4 is due to timeouts on one of the servers. Graph (c) shows the effect of disabling automatic loading of inlined images by the browser, as described in Section 8.2.3 and shown in Figure 8.11. The App and User components for each trace are equivalent between graphs to within experimental error. Each graph also contains error bars which show the 95% confidence intervals of the Total values.

Figure 8.6: Graphical Depiction of the Benefits of Dynamic Sets to WWW Search

Limit seconds	SETS	Trace 1 seconds	Trace 2 seconds	Trace 3 seconds	Trace 4 seconds	Trace 5 seconds
30	W/O	442.0 (62.0)	219.4 (27.0)	260.2 (19.1)	484.0 (50.5)	410.1 (45.2)
	With	39.8 (12.0)	38.4 (9.3)	72.2 (17.6)	17.7 (20.2)	29.3 (7.5)
60	W/O	608.3 (92.3)	258.2 (37.8)	348.6 (27.9)	570.5 (92.3)	551.9 (44.8)
	With	40.1 (12.2)	41.9 (11.4)	75.8 (19.0)	17.8 (20.4)	29.3 (7.5)
90	W/O	693.4 (113.1)	284.6 (50.7)	406.8 (27.5)	603.5 (103.4)	613.3 (63.5)
	With	40.1 (12.2)	41.9 (11.4)	75.8 (19.0)	17.8 (20.4)	29.3 (7.5)
120	W/O	738.4 (119.7)	302.6 (59.9)	437.7 (32.0)	621.5 (103.6)	630.2 (74.3)
	With	40.1 (12.2)	41.9 (11.4)	75.8 (19.0)	17.8 (20.4)	29.3 (7.5)

The aggregate fetch times in Figure 8.5 contain some fetches which take significantly longer than normal and probably would have been aborted by users in practice. This table shows that there is still a noticeable benefit from dynamic sets even if these fetches are aborted after some period of time. The table presents the result of post-processing the trace replay output, truncating fetches (including the time to fetch inlined images) to limit of 30, 60, 90, or 120 seconds. Since these numbers are the result of post-processing, they may overestimate the time for runs with SETS. In the actual execution a long fetch that tied up SETS' resources, preventing other data from being prefetched, may not have appeared as a long delay to the application. The number of truncated fetches is shown in Figure 8.8.

Figure 8.7: Fetch Times When Limiting Latency Using Post-Processed Replay Results

Limit seconds	SETS	Trace 1	Trace 2	Trace 3	Trace 4	Trace 5
30	W/O	7.0 (1.6)	2.0 (0.8)	3.4 (0.4)	4.2 (2.0)	6.2 (1.3)
	With	0.2 (0.4)	0.4 (0.4)	0.4 (0.4)	0.2 (0.4)	0.0 (0.0)
60	W/O	3.8 (1.3)	1.0 (0.6)	2.6 (0.4)	2.0 (1.6)	3.6 (1.8)
	With	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
90	W/O	2.0 (1.0)	0.6 (0.4)	1.4 (0.4)	0.6 (0.4)	0.6 (0.4)
	With	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
120	W/O	1.4 (0.4)	0.6 (0.4)	0.8 (0.4)	0.6 (0.4)	0.4 (0.4)
	With	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)

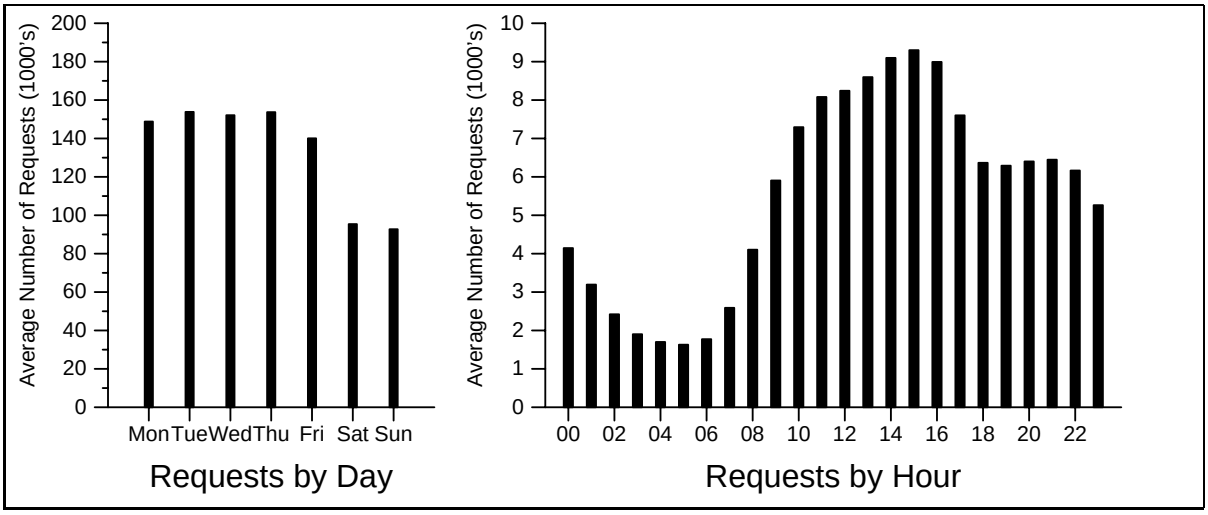
This table shows the number of fetches whose fetch times were truncated to the limit of 30, 60, 90, or 120 seconds. The number is much lower with the use of SETS than without, since prefetching hides most of the latency from the application. The effect of these truncations on average aggregate fetch time is shown in Figure 8.7.

Figure 8.8: Average Number of Aborted Fetches in Post-Processed Replay Results

8.2.2 Effect of the Time of Search

This test examines the influence of the first primary factor, the time a search is run, on the performance improvements offered by SETS. This factor is relevant because the time

of search indicates the amount of external load generated by other users of the system that the search is likely to encounter. Load is interesting because it tends to increase fetch latencies and increases the variance in response time, and is commonplace on the WWW. Although one might be tempted to eliminate load from the experiments, for instance by using a simulator, doing so would take away a significant component of latency on the WWW.



These graphs show the average number of requests by day and by hour seen by the Library of Congress WWW server (http://lcweb.loc.gov/stats) during the first half of 1996. The average number of requests on Saturday or Sunday (94,057) is much lower than the weekday average (156,898). The graph on the right shows the number of requests by hour, averaged across the 183 days included in the results. Requests seem to peak during the afternoon (US Eastern). Statistics from other WWW servers hold similar results, but do not record statistics as thoroughly as does the Library of Congress. For American servers, load seems to be dominated by American clients which means peak loads during the afternoon hours in North America. Although it is unclear whether this relationship also holds in other parts of the world, the servers accessed by the traces are all within North America and the Caribbean.

Figure 8.9: Usage Statistics for the Library of Congress WWW Server

Fortunately, the load on the network and servers is reasonably predictable at a large granularity. Figure 8.9 shows that loads tend to be higher during the day than at night, and higher during the week than on weekends. This experiment exploits this fact by replaying the traces at a time with different load characteristics than the base test to see how load affects the benefits of using SETS.

Figure 8.10 shows the results of replaying the five traces during the weekend, when loads are typically much lower. These results are also graphed in Figure 8.6.(b). The first observation to make is the significant reduction in average fetch time for the non-SETS case as compared with the results of the mid-week test shown in Figure 8.5. The reduction is most likely due to the lower load on the networks and servers. A second observation is that SETS can benefit from the lower load as well, and still manage to offer significant reductions in latency. For most of the traces, the fetch time with SETS is hardly noticeable, and is only a fraction of the CPU time taken to display the WWW objects.

Trace	SETS	User seconds	App seconds	Fetch seconds	Total seconds	Savings in Fetch
1	W/O	825.90 (0.2)	56.50 (1.3)	143.10 (47.5)	1025.50 (46.2)	
	With	826.00 (0.3)	57.30 (1.0)	4.80 (1.0)	888.10 (0.5)	96.65%
2	W/O	787.40 (0.1)	46.80 (1.8)	148.00 (45.3)	982.30 (45.0)	
	With	787.70 (0.3)	43.60 (1.6)	11.00 (3.8)	842.40 (4.3)	92.57%
3	W/O	1203.10 (0.1)	45.40 (2.0)	169.80 (33.7)	1418.40 (34.9)	
	With	1203.10 (0.4)	44.90 (3.1)	6.50 (2.4)	1254.60 (1.6)	96.17%
4	W/O	957.83 (0.7)	92.63 (4.9)	1140.52 (9.0)	2190.99 (13.7)	
	With	954.02 (0.6)	110.16 (6.5)	17.80 (16.4)	1081.98 (20.3)	98.44%
5	W/O	901.32 (0.7)	77.24 (6.3)	257.50 (28.6)	1236.07 (22.0)	
	With	893.97 (0.9)	77.41 (16.1)	4.43 (0.6)	975.82 (17.1)	98.28%

This table shows that the benefits from SETS are also achievable during times of low load by presenting the results of replaying the traces on the WWW during the weekend. The table presents the average aggregate elapsed time in seconds to replay the traces on the live WWW. The numbers for each trace are the averages of 5 runs, the standard deviation is presented in parentheses. Section 7.2.2 describes the metrics User, App, Fetch, Total, and Savings in Fetch, Section 8.3 describes the traces and replay mechanism. Comparing the results in this table with those in Figure 8.5, one can see the effect of external load on the performance of search in the WWW. Figure 8.6.(b) contains a graph of these results.

Figure 8.10: Results of Weekend Replays of WWW Traces

One anomaly does exist: the high Fetch time for Trace 4 both with and without SETS. During the weekend in which this trace was replayed, a particular server deterministically timed out on images, but would serve HTML pages without a problem. Documents that contained many images stored on that server thus suffered timeouts of 75 seconds for each image. Although repeating this experiment on a different day would likely produce

results more in line with the other traces, it does demonstrate an advantage of SETS: the ability to reorder the members allows SETS to hide such long delays behind the processing of other objects.

8.2.3 Effect of Inlined Images

Another factor affecting the time to fetch and display data is whether or not the browser automatically loads inlined images. Inlined images are pictures that are displayed as part of the document, but that are stored as separate objects. Before Mosaic displays a document, it must first fetch all of the images the document contains. Thus the amount of time a user must wait for the information to be displayed is equal to the sum of the time it takes Mosaic to fetch the document and the images it contains, plus the time to process the document to find the images and the time to display them. To reduce latency, Mosaic caches images in its memory. Because some documents contain the same image many times, the image cache can be very effective at eliminating fetches. However, documents from different authors or servers tend not to share images, limiting the effectiveness of the image cache in general.

Because inlined images add to the latency of fetching and displaying information, the Netscape Navigator displays the text in the document as soon as it can, and fetches and displays the images asynchronously. Unlike Mosaic, a user of Netscape does not have to wait for the complete document to arrive before examining its contents. Another approach to reducing image-induced latency is taken by users who disable the automatic fetching of inlined images. A study of a commercial WWW site found that 20% of accesses to their server were from people who were using a text-based browser or who had disabled the automatic loading feature[24]. In cases where bandwidth is limited, such as a personal computer connected via a 14.4 Kbaud modem, disabling inlined images can substantially improve the response time of search on the WWW.

These approaches to avoiding delays due to inlined images raise the question of whether dynamic sets will offer reduced latency to users of Netscape or to those who do not fetch the inlined images by default. To answer this question, this test replayed the traces, but disabled loading of inlined images in Mosaic and the prefetching of the images by the HTTP warden. Runs without SETS precisely capture the behavior of users of Mosaic who disable automatic loading of inlined images. They also exhibit fetch times at least as long as those Netscape would exhibit because loading the image adds some delay to the display of text, and also delays the user if the images are necessary to understand the text.

The only difference between this test and the base test described in Section 8.2.1 is the treatment of images. Figure 8.11 presents the results of this test. The first observation to

Trace	SETS	User seconds	App seconds	Fetch seconds	Total seconds	Savings in Fetch
1	W/O	825.80 (0.4)	20.60 (2.9)	100.90 (21.5)	947.50 (22.3)	
	With	825.80 (0.2)	21.90 (0.8)	7.40 (1.8)	855.10 (1.8)	92.67%
2	W/O	787.50 (0.2)	14.50 (1.4)	96.40 (10.9)	898.40 (11.7)	
	With	787.80 (0.5)	16.30 (1.3)	6.20 (2.5)	810.40 (2.3)	93.57%
3	W/O	1202.60 (0.2)	17.80 (1.2)	188.90 (51.9)	1409.40 (51.8)	
	With	1202.70 (0.2)	20.10 (1.4)	9.70 (2.2)	1232.60 (2.8)	94.87%
4	W/O	957.50 (0.4)	30.80 (3.1)	98.90 (13.0)	1087.30 (15.2)	
	With	952.80 (0.1)	38.70 (2.0)	7.30 (6.3)	998.90 (5.2)	92.62%
5	W/O	899.90 (0.3)	21.20 (0.9)	190.80 (43.9)	1112.00 (43.5)	
	With	892.80 (0.1)	24.60 (2.8)	4.00 (0.8)	921.50 (2.7)	97.90%

This table shows the benefit of SETS when inlined-images are not automatically loaded as would be seen by, for example, users of text-based browsers or who disable image loading by default. The numbers are the average of 5 replays of the same traces (with standard deviations in parenthesis), but with automatic loading of inlined-images disabled in the browser. Section 7.2.2 describes the metrics User, App, Fetch, Total, and Savings in Fetch, Section 8.3 describes the traces and replay mechanism. The high variance in Fetch time for Trace 4 with SETS is due to problems with the SCS CMU WWW server. The variance in latency for runs without SETS results from running on the live WWW. Figure 8.6.(c) contains a graph of these results.

Figure 8.11: Results for Trace Replay Without Fetching Inlined Images

make is that disabling the loading of images reduces the fetch times for the unmodified (W/O SETS) traces, compared with the results in Figure 8.5. An implication of this observation is that Mosaic's image cache is not sufficient to eliminate image-induced latency. The second observation is that SETS still offers substantial savings, indicating that they would also be useful to users of Netscape or other more sophisticated browsers. A third observation is that the amount of computation (App) has dropped significantly, indicating that a large portion of the computation to display a page is spent processing images.

The high variance for Fetch time for Trace 4 ($\overline{x} = 7.3$, $\sigma = 6.3$) with SETS is due to a single demand load of an instructional page from CMU's WWW server in a single trial. 13.4 of the 19.2 seconds in the 4th replay of the trace resulted from a single load. If the load had taken the .6 seconds average instead of 13.4 seconds, the average load for this trace would be 4.4 with a variance of 2.5. As mentioned previously, these demand loads often added substantial overheads to the SETS replay. This points out both the high variance of response from WWW servers (there is virtually no network delay to access CMU's WWW server from the client used in the experiment), and the benefit of

prefetching to reducing the impact of that variance.

8.2.4 Effect of Low Bandwidth Network Connections

The final primary factor to consider is the bandwidth of the connection that the client machine has to the Internet. Many Internet end-users are connected via very low bandwidth links such as phone lines. For example, consider the thousands of America Online users, or users of other "Internet service providers" who access the Internet from home PCs via 14400 or 28800 baud modems. In such cases, the low bandwidth can be a significant contributor to the latency seen by these users.

To asses the benefit of dynamic sets in such a setup, this test replayed the traces on a laptop computer connected to the Internet via Serial Line IP (SLIP) over a 14400 baud modem. The laptop was dialed into a shared terminal server; the network connection from the terminal server to the Internet is the same as that used by the client in the other tests. Thus the only difference in network topology is the use of the SLIP line. The laptop is a 25Mhz 486 with 32MB of memory. Typical disk-to-disk transmissions over SLIP achieve anywhere from 1300B/s to 2500B/s, depending on the modem's rate of compression. I chose to use the laptop for this experiment because it ran the necessary software (Mach 2.6 with SETS modifications) and had a modem. Since users frequently have different hardware at home than on their desk, I felt this test did not need to share the same environment as the others. In addition, I felt that the alternative of using a laptop to route between the DECStation and the Internet added unnecessary complexity to the experiment. Further, the performance of the two client machines is roughly comparable.

Figure 8.12 shows the results of this experiment. Because the client machine for this test is different from the client used in the other tests, these results should not be directly compared with those of the other experiments. In addition, the laptop's timer is only accurate to 15 milliseconds. Although this does not affect the accuracy of the averages, it explains the zero variances in User think time.

The table shows that SETS offers substantial performance improvements even over very slow networks. Since the network bandwidth is extremely limited, the HTTP warden was limited to fetch up to 2 objects concurrently, as opposed to the limit of 6 in the other tests.[2] Setting this lower limit benefited SETS by reducing the likelihood of overloading the SLIP connection from too many incoming bytes streams vying for service. However,

[2]From SETS perspective, a document and its images count as 1 object. Thus `limitOpens` = 5 will restrict SETS to prefetching 5 objects, but the number of open connections is limited by the number of HTTP worker threads, which is 6 for the other tests and 2 for this test.

Trace	SETS	User seconds	App seconds	Fetch seconds	Total seconds	Savings in Fetch
1	W/O	826.00 (0.0)	82.40 (1.2)	1038.84 (34.2)	1947.24 (35.1)	
	With	826.00 (0.0)	75.20 (2.5)	299.85 (24.6)	1201.06 (25.8)	71.14%
2	W/O	781.00 (0.0)	66.13 (1.7)	912.28 (70.1)	1759.42 (69.3)	
	With	781.00 (0.0)	66.69 (1.4)	204.13 (32.5)	1051.83 (32.1)	77.62%
3	W/O	1198.00 (0.0)	48.41 (1.5)	733.85 (59.4)	1980.26 (60.4)	
	With	1198.00 (0.0)	54.25 (2.8)	250.06 (21.8)	1502.31 (19.4)	65.92%
4	W/O	945.00 (0.0)	67.32 (9.6)	1394.69 (33.5)	2407.01 (36.3)	
	With	941.00 (0.0)	50.98 (2.6)	424.84 (40.3)	1416.83 (42.5)	69.54%
5	W/O	898.00 (0.0)	64.03 (2.0)	1378.63 (60.6)	2340.67 (60.7)	
	With	892.00 (0.0)	63.01 (2.4)	398.95 (76.9)	1353.96 (74.9)	71.06%

This table shows the potential for performance improvements from SETS even when bandwidth is limited, although the benefit is smaller than for the other tests. The table presents the average aggregate elapsed time in seconds to replay the traces on the live WWW over a 14400baud SLIP connection, from a DEC 425SL laptop. The numbers for each trace are the averages of 5 runs, the standard deviation is presented in parentheses. Section 7.2.2 describes the metrics User, App, Fetch, Total, and Savings in Fetch, Section 8.3 describes the traces and replay mechanism. The experiment restricted prefetching to at most 2 objects at a time in order to prevent thrashing on the SLIP connection. Figure 8.13 contains a graph of these results.

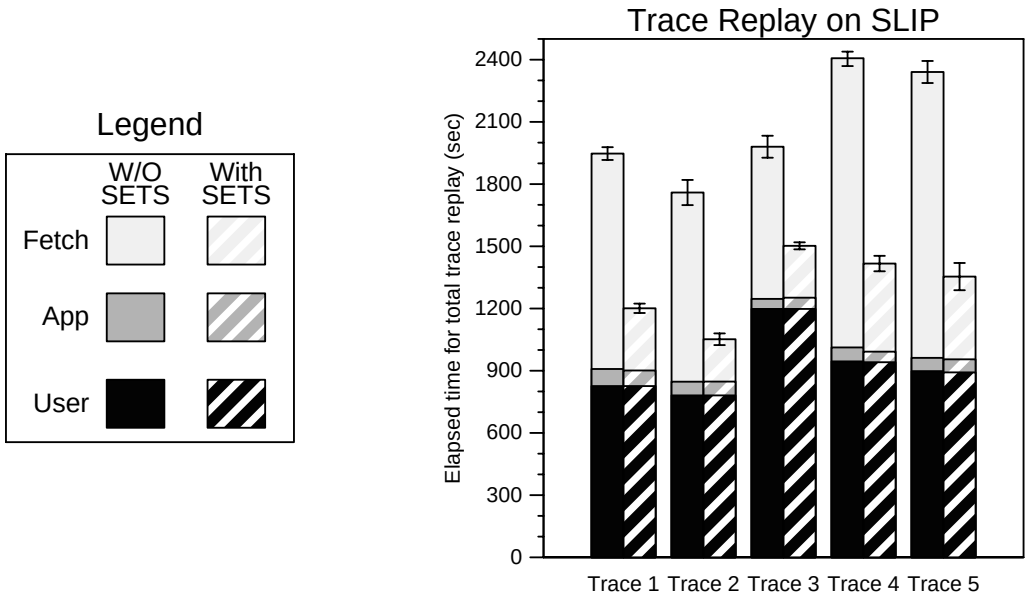
Figure 8.12: Results for Trace Replay over SLIP from a Laptop Computer

this lower limit also penalizes SETS by limiting its ability to fetch objects in parallel in order to overlap long latencies to remote servers.

The extent of this penalty can be seen by comparing the time it actually took SETS to fetch the objects to the time it (theoretically) would take to transfer the data over the SLIP line. If the transfer time is close to the actual time, the performance is fundamentally bandwidth limited by the SLIP link. However, the transfer time is much lower than the total time to fetch the objects. As an example, in one run of Trace 3, the HTTP warden fetched 428284 bytes of sets data[3] in 821 seconds, but it would have only taken 329.4 seconds at 1300B/s (these times are not shown in the tables). Most of the difference is likely to be due to the limit on parallel fetches.

This penalty results from the tension between bandwidth and latency, reducing parallel fetches reduces bandwidth but increases the aggregate latency. I conjecture that one could solve this problem by adding a WWW proxy on the far side of the SLIP link.

[3]This does not include instructional pages, and is the result of summing the first 6 rows for Trace 3 in Figure 8.4.



These graphs show the results of the fourth WWW experiment. The bar graph shows the results of replaying the traces over a phone line, described in Section 8.2.4 and shown in Figure 8.12. Each graph also contains error bars which show the 95% confidence intervals of the Total values. This graph is displayed separately from the ones in Figure 8.6 because it shows test results from a different client machine than the one used in the other tests.

Figure 8.13: Graph of Search Performance on WWW over SLIP.

The laptop would disclose the set's membership to the proxy when the set was opened. The proxy would aggressively prefetch the objects to its local disk in parallel, in the same manner as that employed by SETS. The laptop would then fetch the objects from the proxy over the SLIP line. As a result, the laptop could drive the SLIP line to full utilization to lower the latency seen by the client, without sacrificing the benefits of parallel fetches.

8.3 Conclusion

In summary, SETS offers tremendous benefits to WWW searchers who iterate over sets of objects they specify at the beginning of the search. By examining the Fetch numbers for runs with SETS, one can see that SETS was able to greatly reduce the I/O latency seen by the application and user, and was able to virtually eliminate I/O from the performance of the search in some cases. For these traces, it was never the case that runs with SETS ran slower than those without, and it was always the case that the benefit from SETS was statistically significant. Another benefit of dynamic sets is that they reduce the magnitude of variance that would be seen by the user. Both of these benefits would result in raising the usability of the WWW and lowering the frustration of the user if dynamic sets were widely employed. Although the WWW is egregious in the magnitude and variance of latencies it exhibits, it is an important case to consider due to its popularity and scale.

The benefits result from overlapping I/O and computation, in particular the pauses for user think time that are captured in the traces. SETS also benefits by fetching from independent servers in parallel. I did not directly examine the impact of these parallel fetches on the WWW, but anecdotally the impact was small since neither server administrators nor other users of the local network were sufficiently disturbed to report a problem. In addition, I occasionally monitored network response times from another client on the same local area network to gauge the impact of prefetching on other users, and found no serious performance degradation. This anecdotal evidence indicates that one can indeed achieve substantial performance improvements without significantly overloading the network.

As stated above, these tests do have one caveat. The modified traces do not necessarily demonstrate that one can use dynamic sets in such a way as to achieve these performance benefits. The question of whether users of browsers can perform searches using sets remains to be answered, as does the question of whether or not users can precisely specify the objects they wish to examine in terms of sets. However, given the dramatic benefits demonstrated by these experiments, it is worth investing the effort to deploy dynamic sets and build a user community.

Chapter 9

Evaluation: Search on local-area DFS

The second experiment examines search on local-area distributed file systems (DFS). Distributed file systems are a common paradigm for read/write access to persistent data, and are in widespread use in government, industry, and university settings. DFS also provide a domain that is distinct from the domain of GDIS in several ways. First, DFS are typically owned and managed by a single organization such as a school or department, whereas GDIS often span many organizations, companies, and even countries. As a result, DFS tend to be much smaller in scope than a GDIS, but can still store thousands of objects and be accessed by hundreds of users. Second, fetch times in DFS tend to be much lower than those in GDIS. One reason is that DFS span a much smaller geographic area (often within a single building), so propagation latencies are smaller. Another reason is that load on DFS servers tends to be manageable. The user community of a particular DFS is typically known, so it is easier to purchase equipment sufficiently powerful to support them most of the time. Third, the variance in response time is also much lower, although it is still not trivial: use of caches and specialized hardware can make some accesses much faster than others. Because of these differences and because they serve a different goal, supporting the persistent data needs of a small to medium sized organization[83], DFS will continue to be interesting systems for the foreseeable future.

The domain representative examined by this experiment is Sun's Network File System (NFS)[77]. NFS is a widely used distributed file system, and is often employed as a point of comparison when describing the performance of new distributed file systems. An NFS server exports a portion of its name space (subtree) to NFS clients. A client mounts a subtree into its name space, allowing applications on that client to access data from the server. Application reads and writes of NFS file data are redirected by the client to the server storing the data. Reads and writes are in block-sized chunks, where a typical block size is 8KB. The data is cached in the client's in-memory buffer cache for fast access to recently referenced pages.

Factors	Name	Description
Primary	Cardinality	Number of objects in a set
	File size	Average size of candidate objects
	Servers	Number of servers exporting candidate objects
	Think	User or CPU processing time
	Cache	Effects of incidental cache hits
	Bandwidth	Bandwidth of network
Secondary	Server speed	OpenCost, ReadCost, disk speed
	File system	Block size, number of buffers
	Prefetch parameters	Described in Chapter 6
	Cache Size	Number of in-memory buffers and inodes

This table lists the primary and secondary factors affecting the performance of search on NFS. The impact of the primary factors is examined through experiments, whose results are presented below. The secondary factors have a smaller effect on the elapsed time of the search, or are independent of dynamic sets, and are not directly examined by this experiment.

Figure 9.1: Factors Affecting Search Performance on NFS

The goal of this experiment is twofold. The first goal is to determine if SETS offers performance improvements in this domain. The related question of whether users can effectively employ sets on NFS is more easily answered than it was for WWW users. Unix users are familiar with and frequently use set notation in their daily work: `csh` wildcard notation. Wildcard notation lets a user simply and easily define a set of objects to process. Adding dynamic sets to this scenario is trivial: instead of the shell expanding the notation into a list of names, the application passes the notation to `setOpen()` as the specification of set membership. The program must be rewritten to iterate over the set instead of stepping through the list, but these changes are straightforward and were shown in Figure 3.2.

The second goal of the experiment is to gain a better understanding of the low-level factors affecting the performance of SETS applications. Because the variance in latency is much less egregious in a DFS like NFS than on the WWW, one can capture reasonably realistic results even when running experiments on an isolated network. This lowers the likelihood of some other user influencing the results of the experiment, which in turn lowers the variance. A lower variance allows one to see the influence of low-level factors more clearly, because smaller differences can be recognized as statistically significant.

The primary factors examined by this experiment are listed in Figure 9.1. Cardinality (N in the model, Figure 7.2) is the number of objects in the set, file size ($Size$) is the size of the objects. Servers (S) is the number of servers storing members of the set, and

therefore is an indication of the parallelism that can be achieved. Bandwidth (*Band*) indicates the peak available network bandwidth, although the achievable bandwidth may be limited by other factors such as disk or server bus bottlenecks. Cache effects explore the performance of a search when some of the candidate objects are already cached. Think time includes both user think time (*Think*) for interactive searches as well as processing time (*Comp*) for non-interactive applications.

9.1 Test Methodology

Search in a DFS like NFS can be either interactive like search in GDIS, or non-interactive. Examples of non-interactive search applications are Unix utilities like `grep` and `find`, examples of interactive search tools are browsers, file managers, and the Unix utility `more`. Interactive searches involve both processing and think time, while non-interactive searches only involve processing time, since the application requests the next object as soon as the current one has been fully read and processed.

In order to capture both behaviors, the tests in this experiment use a synthetic benchmark as an application workload. The benchmark program is derived from the Unix `grep` utility, preserving the I/O pattern of `grep` (whole file sequential, process a block before reading the next), but providing two parameters to control the amount of think time. The first parameter, *Comp*, is the amount of processing to be done, expressed in terms of microseconds/byte. It controls the number of instructions executed by the benchmark program between file system reads. The second parameter, *Think* is the amount of user think time, expressed in terms of seconds/object. This parameter controls the duration of the pause after all the data in an object has been processed and before the next object is requested. The benchmark program performs these pauses by calling the Unix `select()` facility, which causes the program to sleep for approximately the desired period of time.

The basic test consists of running the synthetic benchmark program on a set of uncached NFS files using the basic file system operations and again using dynamic sets. The test script flushes both the client's and the servers' buffer caches to eliminate co-dependencies between runs before running the benchmark. It runs the benchmark using a list of NFS file names expressed in wildcard semantics, for instance "`/mnt/nfs/16K{a,b,c,d}`". Using kernel modifications, the benchmark program can accurately capture the number of hits and misses in the buffer cache, the number of NFS operations that were performed, the aggregate amount of time spent performing NFS operations, the amount of client CPU idle time, and the length and amount of disk activity on the servers.

9.2 Results of NFS Experiments

The client and server machines for these experiments are DECStation 5000/200s with 32 MB of RAM running the Mach 2.6 operating system, which includes an in-kernel NFS version 2 client and server. The machines have a hardware cycle counter with which the kernel can accurately time events to within a few microseconds. The tests were run on an isolated Ethernet. The machines were lightly loaded: only the user running tests was logged in during the tests, although the machines were not booted single user. Since the machines are normally shared among several users, they were rebooted before each series of tests to ensure a clean test environment. In addition, four unessential daemons (`at`, `afsd`, `zhm`, and `ticketd`) were killed before running the experiments to reduce contention for the network.

Each experiment consists of varying one of the primary factors and repeating the basic test 15 times. The numbers presented in the tables and graphs below are the averages of the first 10 runs. On occasion, the NFS fetches for a run would take 50% to 100% longer than the average of the other runs. This happened both with and without SETS. Since this behavior seemed to be due to a bug in the implementation of NFS unrelated to SETS or this dissertation, these runs were eliminated from the results and replaced by the 11th run. There was never a case in which two runs within a test showed this higher than normal latency.

The factor levels chosen for the base test were picked to reflect common usage where possible. The base test consists of runs on sets with cardinality of 12 16KB files stored on 1 server, 10base2 Ethernet (10Mbps), $Think = 0$ seconds of think time per file (non-interactive search), $Comp = 1$ microsecond of processing/byte, and a cold cache (no objects cached). The tests used the same SETS cache parameters as the WWW tests, shown in Figure 7.3, except where noted. The test of the base levels is actually repeated in each of the tests below, for example the data point for $N = 12$ in the cardinality test shown below in Figure 9.2.

9.2.1 Cardinality

Figure 9.2 shows the results of running the benchmark on different sized sets of uncached 16KB files. As expected, the results show that dynamic sets reduce the running time of the application for $N > 1$, and the amount of reduction grows with the size of the set. For $N = 1$ there is no statistical difference in the run times. The reduction in run time is a result of lower idle times: the application spends less time waiting for data and more time working. The tradeoff is that more computation has to be done in order to prefetch the files.

N	SETS	CPU milliseconds	Stall milliseconds	Total milliseconds	Savings in Total
1	W/O	27.46 (0.4)	37.91 (1.2)	65.38 (1.5)	
	With	29.66 (0.5)	37.51 (1.1)	67.17 (1.1)	-2.74%
2	W/O	50.26 (0.3)	123.88 (8.7)	174.14 (8.8)	
	With	54.87 (1.4)	55.26 (11.0)	110.13 (10.6)	36.76%
4	W/O	95.17 (0.7)	244.82 (10.3)	339.99 (10.5)	
	With	105.45 (3.5)	109.96 (6.3)	215.41 (4.7)	36.64%
8	W/O	185.02 (1.1)	481.73 (7.3)	666.75 (7.9)	
	With	215.70 (5.6)	153.64 (11.2)	369.34 (8.7)	44.61%
12	W/O	274.33 (1.4)	725.71 (15.4)	1000.03 (15.4)	
	With	326.28 (9.1)	215.33 (19.7)	541.61 (13.3)	45.84%
16	W/O	364.59 (2.1)	944.55 (16.5)	1309.14 (16.0)	
	With	434.59 (6.7)	282.38 (15.7)	716.97 (12.7)	45.23%

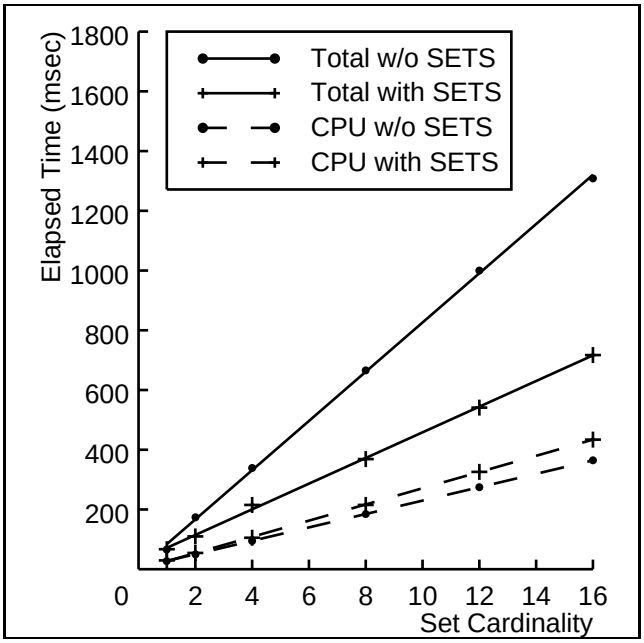
This table shows the effect of set cardinality on the benefit from dynamic sets. The left column (the N parameter of the model in Section 7.1) shows the number of 16KB files that were processed by the synthetic benchmark, holding the other factors at their base levels. The metrics in the other columns are described in Section 7.2.2. The numbers are the mean of 10 runs, standard deviations are presented in parenthesis.

Figure 9.2: Effect of Set Cardinality on Benefit from SETS

This increase in computation can be plainly seen in Figure 9.3, which plots the average CPU and Total times from Figure 9.2. The dashed lines show that the amount of CPU is greater for SETS, and that the difference in computation grows with the size of the set. Fortunately, this increase in computation is small, and in particular is smaller than the decrease in latency that SETS offers. As a result, SETS provides a net savings in elapsed time to run the benchmark, which can be seen by comparing the solid lines in the graph.

From where is SETS getting the reduction in latency? One source is clearly the ability to overlap computation and I/O. As shown in Figure 7.4, one can consume some of the idle time with computation to send and receive other messages, reducing the amount of idle time with legitimate work. The second source of reduction stems from an additional benefit of prefetching: a higher I/O efficiency. This efficiency can be seen in Figure 9.4, which shows the amount of time the system spent performing I/O. SETS gains an advantage here by overlapping multiple requests. Since queues are full more often, the server and network sit idle for smaller periods of time resulting in higher utilization.

These tests also show a third benefit of sets. Because SETS knows all members will be read sequentially, it can start pre-reading a member's data immediately, as opposed to



This graph shows the cost and benefit of SETS vs the set cardinality. The points are the experimental results from Figure 9.2, with lines fitted via regression with a correlation coefficient of greater than .9995 in all cases. The dots show the results without SETS, the pluses those with SETS. The solid lines show the total elapsed time and the dashed lines show the amount of CPU, the difference between the solid and dashed lines is the stall time. From the graph, one can see the increase in CPU usage due to SETS, but also the larger reduction from overlapping computation and I/O. The result is that SETS can reduce the run time for every file in the set, and thus get more benefit for larger sets.

Figure 9.3: Benefit of SETS vs Cardinality

	N					
SETS	1	2	4	8	12	16
W/O	43.18 (1.46)	133.79 (8.64)	264.03 (10.32)	518.80 (7.71)	781.17 (15.84)	1018.75 (15.82)
With	42.86 (1.02)	86.72 (10.80)	191.36 (4.77)	344.35 (8.55)	515.99 (13.37)	689.22 (14.14)
Savings	0.74%	35.18%	27.52%	33.63%	33.95%	32.35%

This table shows the Work metric for the cardinality test. Work measures the actual amount of time spent by the system fetching files, as opposed to the apparent fetch times seen by the application. When the Work times with SETS are less than those without SETS, the I/O for SETS is more efficient.

Figure 9.4: Amount of Work Done by System to Fetch Objects for Cardinality Test

waiting until a sequential access pattern has been established. The Unix FFS on the

server's disk, for instance, does not read-ahead until after the first two blocks have been read. The files in this test are only 2 blocks long, so the runs without SETS do not benefit from read-ahead.

9.2.2 File Size

Size KB	SETS	CPU milliseconds	Stall milliseconds	Total milliseconds	Savings in Total
1	W/O	45.05 (0.5)	136.64 (13.5)	181.69 (13.6)	
	With	58.63 (0.3)	37.04 (3.5)	95.68 (3.6)	47.34%
4	W/O	89.20 (1.3)	247.24 (16.1)	336.44 (15.7)	
	With	110.04 (1.9)	62.27 (11.1)	172.31 (10.6)	48.78%
16	W/O	275.44 (1.0)	720.00 (17.3)	995.44 (17.6)	
	With	323.12 (4.7)	227.23 (14.7)	550.35 (12.6)	44.71%
64	W/O	1058.55 (27.5)	1734.84 (22.8)	2793.39 (43.1)	
	With	1337.57 (10.1)	294.67 (45.0)	1632.24 (35.9)	41.57%
256	W/O	4249.74 (102.8)	6541.34 (78.1)	10791.08 (78.5)	
	With	4850.37 (118.0)	4924.11 (167.4)	9774.47 (150.8)	9.42%
1024	W/O	16754.25 (146.4)	23138.94 (836.6)	39893.18 (849.0)	
	With	20743.26 (284.9)	15647.30 (776.2)	36390.56 (800.4)	8.78%

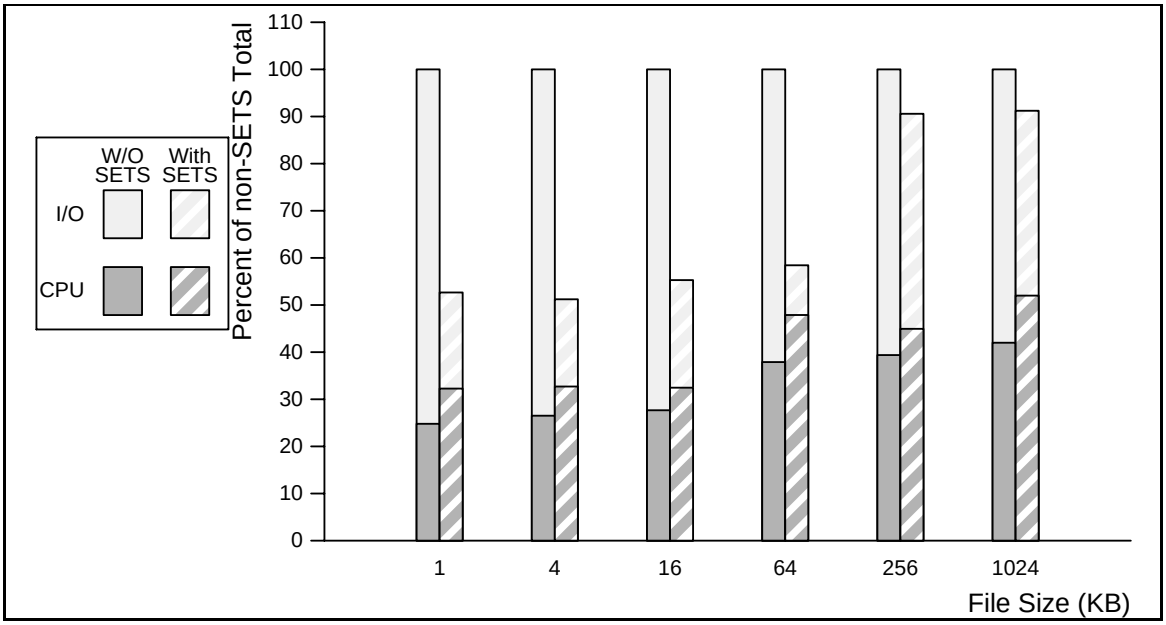
This table shows the effect of member size on the benefit from dynamic sets. The left column shows the member size in KB for each test (*Size* in the model in Section 7.1), the other factors were set to their base levels. The numbers are the mean of 10 runs, standard deviations are presented in parenthesis. The metrics in the other columns are described in Section 7.2.2. The chief result is that SETS performs very well for typical NFS files, but the performance drops off as files get large.

Figure 9.5: Effect of Member Size on Benefit from SETS

Figure 9.5 shows the benefits of dynamic sets relative to file size. The experiment runs the benchmark on sets of identically sized files, increasing the file size on repeated tests from 1KB to 1MB in multiples of 4. The interesting result of this benchmark is that the benefit from SETS decreases as the file size increases. However, dynamic sets do offer some benefit for these larger files and still offer substantial benefit for files smaller than 64KB. Fortunately, the range of sizes under which dynamic sets offer greatest performance improvements covers most files in a typical Unix environment. Studies have shown median

file sizes between 10KB and 16KB, and 80% to 90% of files are less than 50KB in size[3, 64, 78].

Why do SETS fare worse as the file size grows? As predicted by the model in Section 7.1.1, SETS may offer little benefit in situations in which performance is dominated by the cost of reading data, because read-ahead will obtain most of the benefit of prefetching. Reading large NFS files from a single server with no user think time is such a situation. Essentially, the relative benefit SETS gets by prefetching decreases as the performance improvement from read-ahead increases. Fortunately, SETS can still overlap processing and I/O to retain some benefit.



This figure graphs the results in Figure 9.5, normalizing the values for each file size to the average total execution time without SETS. Striped bars show the benchmark times with SETS, solid bars show the results without the use of SETS. Light colors show the idle stall time, dark colors the amount of CPU. Of interest is the benefit from SETS for smaller files and the relative drop in benefit from SETS for larger files. Fortunately, 80% of files on typical Unix systems are smaller than 64KB.

Figure 9.6: Normalized Execution Times vs Member Size

Figure 9.6 contains a bar graph with the experimental results, normalized to the average total execution time without SETS for each file size. Normalization allows for comparison between different file sizes. Comparing the difference between the SETS and non-SETS

cases for both small and large files, one can see how the relative benefit from SETS drops for bigger files as the percentage of time spent in the idle thread increases. This drop off in performance for large files is sensitive to the primary factors. With larger values of $Comp$ or $Think$, SETS can overlap more of the I/O costs with processing, reducing the latency that is apparent to the user. In addition, if the files were stored on multiple servers, SETS could perform the fetches in parallel. As shown in Equation 7.8, parallel fetches reduce the latency seen by the application by up to a factor of S.

One change could be made to improve the performance of SETS for large files. SETS destroys the sequentiality of access at the disk head by reading from multiple files concurrently. This can increase the average seek time, as the head must move back and forth between the locations of the different files on disk[1]. This increase can been seen in the work times for larger files in Figure 9.7, and can also be seen in the measurements of server disk work time which are not shown here. These numbers show that the I/O efficiency demonstrated by SETS on small files is reversed, and SETS sees less efficient I/O when reading large files.

To avoid destroying sequentiality, the SETS prefetching engine could be rewritten to submit requests to read some or all blocks of a file at once. Currently, SETS reads each block synchronously, so concurrent reads from multiple worker threads get interleaved. If the data files accessed in the experiment were stored on different servers, SETS could overlap requests to files on different devices without incurring this overhead.

	Size in KB					
SETS	1	4	16	64	256	1024
W/O	154.97 (13.5)	272.23 (15.9)	775.79 (17.2)	1935.80 (40.7)	7349.71 (73.7)	26220.31 (879.7)
With	86.44 (3.6)	160.01 (10.5)	524.75 (12.6)	1500.21 (53.6)	9293.73 (139.3)	35027.05 (817.5)
Savings	44.22%	41.22%	32.36%	22.50%	-26.45%	-33.59%

This table shows the Work metric for the size test. Work measures the actual amount of time spent by the system fetching files, as opposed to the apparent fetch times seen by the application. Where the Work times with SETS are less than those without SETS, the I/O for SETS is more efficient. This table is further evidence that SETS performs well for small files but not as well for large files. Fortunately, the increase in work time for large files is hidden by prefetching and thus SETS can still offer benefit to sets of large files.

Figure 9.7: Amount of Work Done by System to Fetch Objects for Size Test

[1]The Unix FFS clusters files on disk to avoid seeks when sequentially reading from a file.

9.2.3 Number of Servers

S	SETS	CPU milliseconds	Stall milliseconds	Total milliseconds	Savings in Total
1	W/O	271.82 (3.5)	719.13 (17.0)	990.95 (16.6)	
	With	320.45 (7.8)	221.82 (21.4)	542.27 (14.1)	45.28%
2	W/O	276.95 (6.5)	657.41 (14.4)	934.35 (12.6)	
	With	349.17 (6.1)	68.80 (13.1)	417.96 (11.2)	55.27%
3	W/O	274.89 (2.2)	672.26 (14.1)	947.15 (14.4)	
	With	352.80 (4.2)	55.56 (10.9)	408.36 (8.0)	56.89%

This table shows the effects of parallel fetches by presenting the results for runs of the benchmark program on sets of 16KB files stored on multiple servers. The left column (S in the model in Section 7.1) contains the number of servers on which members of the set were stored, the other factors were set to their base levels. The metrics in the other columns in the table are described in Section 7.2.2. For $S = 1$, all twelve members were stored on one server, for $S = 2$, 6 members were stored on each server, etc. Fetching in parallel has two benefits. First, S blocks can be read in the time it takes to read one. Second, offloading the 1 server disk reduces the extra seek time introduced by SETS. An anomaly can be seen which results from a small difference in the speed of the server disks: the slight reduction in stall times for runs W/O SETS for larger values of S.

Figure 9.8: Benefits of SETS vs S for 16KB NFS files

Figure 9.8 shows the results of running the base case on multiple servers. As one would expect, the amount of idle time for SETS drops as more servers are involved. SETS can fetch multiple files in parallel from independent servers with little overhead, allowing S files to be fetched simultaneously. There is also a second benefit: competition for the disk head is reduced since the load is spread over three disks. This can be seen by comparing the amount of time SETS spent prefetching objects. Figure 9.9 shows the average elapsed time (in milliseconds) to prefetch one file, with standard deviations in parenthesis. As the load is spread across more servers and disks, the cost of I/O drops. This in turn reduces the penalty for using dynamic sets on large files as seen in Figure 9.6. Figure 9.10 contains the results of the same test run on 1MB files, and shows the benefits from parallel fetches to be even more dramatic for large files.

There are two limits to the reduction in latency that can be achieved by exploiting parallelism. First, fetching files in parallel requires more bandwidth. At some point, increasing S will drive the network to thrash, resulting in substantial performance degradation. Fortunately, most of the benefit from parallel fetches can be obtained for small values of S reducing the dependence of SETS on the availability of high bandwidth. In the future,

Size	*S*		
KB	1	2	3
16	182.08 (7.08)	100.33 (2.42)	98.64 (3.42)
64	502.24 (26.59)	281.65 (12.93)	274.33 (30.53)
256	3606.57 (56.21)	1194.03 (107.38)	1000.04 (29.02)
1024	12610.76 (200.43)	6834.87 (131.87)	6711.92 (67.70)

This table shows the average time in milliseconds for SETS to prefetch a file. Section 7.2.2 explains why this metric is appropriate for $S > 1$ while Work is not. By comparing the times for different values of S, one can see how distributing the load over multiple servers reduces the I/O penalty SETS incurs by increasing the average seek time per I/O.

Figure 9.9: SETS Prefetch Time vs Degree of Parallelism

restricting prefetching to reduce bandwidth consumption may be less of a concern since network bandwidth is increasing: 100Mbps local area networks are commonplace and gigabit networks are on the horizon.

The second limitation is that a file must be wholly fetched before the iterator will yield it. As a result, none of the time to fetch the first file can be overlapped with computation. Unfortunately this cost is slightly more than one might otherwise expect, because the time to fetch a file is increased because of contention introduced by prefetching. Whereas the latency to fetch a file with no contention (Figure 9.2, $N = 1$) is 37.51, the lowest idle time here was 55.56 (Figure 9.8, $S = 3$). Some of the 18 milliseconds difference may be due to contention for the network (or contention for servers since SETS can fetch 5 files at once, but from at most 3 servers). However, not all of the difference is necessarily due to contention. Additional latency may also result from each file processed since the time to process S files is shorter than the time to fetch the next S files. Figure 9.15 shows that the cost to fetch a file when $S = 3$ is equivalent to the processing when $Comp = 3$.

Figure 9.11 shows the results of this test run on 16KB, 64KB, 256KB, and 1MB files. Each test is presented as a cluster, with numbers in the cluster expressed as a percentage of the non-sets run time. This graph is interesting as it illustrates several points. First, SETS has cut the idle time substantially when $S > 1$. Second, the relative decrease in benefit from SETS for large files when $S = 1$ is apparent. Third, this relative decrease for large files is not apparent when $S > 1$ because SETS can obtain benefit over read-ahead by exploiting parallelism. This effect was in fact predicted by the model as discussed in Section 7.1.1. Fourth, one can see that parallel fetches introduce some additional CPU overhead, but that this overhead is less than 10% of overall run time.

S	SETS	CPU milliseconds	Stall milliseconds	Total milliseconds	Savings in Total
1	W/O	16693.06 (164.5)	23214.82 (727.1)	39907.89 (740.4)	
	With	20735.93 (273.7)	15546.30 (518.0)	36282.23 (457.5)	9.09%
2	W/O	16737.57 (132.2)	22439.24 (370.9)	39176.81 (415.6)	
	With	21026.54 (121.4)	2858.06 (47.9)	23884.61 (112.6)	39.03%
3	W/O	16813.24 (118.5)	22222.25 (641.9)	39035.50 (633.7)	
	With	21048.67 (115.1)	2531.37 (102.2)	23580.04 (129.6)	39.59%

This table shows the results of performing the multiple server test on 1MB files, leaving the other factors at their base level. The left column shows the number of servers, other metrics are explained in Section 7.2.2. Since the I/O cost of fetching large files is high, the benefits of parallel fetches are more apparent than for smaller files. When $S = 3$, the idle time with SETS is very close to its theoretical minimum, and so the benefit from additional servers is likely to be small.

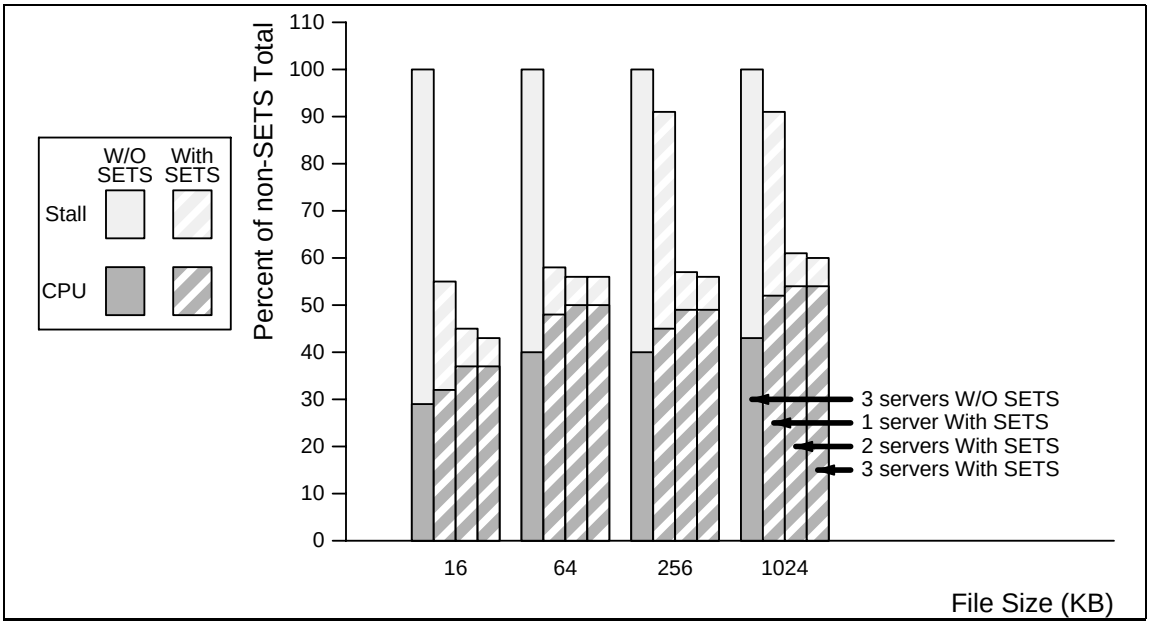
Figure 9.10: Benefits of SETS vs S for 1MB Files

9.2.4 Processing Time

The next experiment considers the effect of processing time on the benefit from SETS. As one would expect from the model in Chapter 7.1, larger processing times offer greater opportunity to overlap computation and I/O which allows SETS to further reduce the idle time. The trade off is that I/O contributes a smaller percentage to overall run time as the processing increases. Thus the potential benefit of dynamic sets to overall run time is diminished.

Figure 9.12 shows the results of an experiment in which the benchmark's processing per byte was increased to emulate more computationally intensive applications. One interesting result is that SETS gets no additional benefit for $Comp > 6$, and most of the benefit when $Comp > 3$. One can see this result by comparing the stall times for runs with SETS in Figure 9.12. When $Comp = 3$, the cost of processing a file is as great as the cost to fetch the next one, so the only stall time SETS sees is the time to fetch the first file. I conjecture that the small benefit SETS gets between $Comp = 3$ and $Comp = 6$ is due to processing being greater than the average I/O plus the variance. Figure 9.13 graphs the results of the same experiment run on 1MB files, in which one can see the same result by comparing the size of the stall component for runs with SETS.

In addition, Figure 9.13 shows a case in which SETS increases the benchmark's run time: 1MB files when $Comp = 0$. With the reduction in application processing, there is less of an opportunity to overlap processing and I/O. In addition, SETS gets little additional benefit from prefetching than from FFS read-ahead, and increases the seek time by



This graph shows the results of the multi-server test for 16KB, 64KB, 256KB, and 1MB files. Each cluster shows the results for the non-sets case with 3 servers and the sets case for 1, 2, and 3 servers, with one cluster for each file size. The values in the cluster are expressed as a percentage of the non-sets run time to allow comparisons between file sizes. Notice that idle times for 2 or 3 servers is approaching the theoretical limit; idle time is close to that of fetching the first time. More servers would not significantly improve the performance. The difference between the times for 1, 2, and 3 servers in the runs without SETS are not statistically significant, so only the results for runs on 3 servers are shown.

Figure 9.11: Benefit of SETS vs S for 16KB, 64KB, 256KB, and 1MB Files

introducing contention for the disk head from prefetching multiple files concurrently. As stated in Section 9.2.2, SETS could reduce the seek time by queueing all blocks from one file before starting to fetch another file from the same server or disk.

9.2.5 User Think Time

Figure 9.14 shows the benefit of SETS for interactive search tools running on NFS files. To emulate interactive tools, SETS uses the select system call to pause for $Think$ seconds between each file. Since these long pause times contribute to idle time, this experiment uses an elapsed time profiler to measure the amount of work and I/O performed by the benchmark, as discussed in Section 7.2.2. Note that tests of interactive search use different metrics than tests of non-interactive search. Except for the Total values, the

Comp $\mu sec/byte$	SETS	CPU milliseconds	Stall milliseconds	Total milliseconds	Savings in Total
0	W/O	81.34 (4.2)	707.80 (11.1)	789.14 (11.0)	
	With	95.12 (1.6)	432.84 (12.3)	527.96 (11.9)	33.10%
1	W/O	280.69 (3.2)	722.76 (8.7)	1003.45 (8.4)	
	With	327.41 (6.5)	226.46 (9.6)	553.87 (7.2)	44.80%
2	W/O	474.54 (3.4)	736.82 (11.7)	1211.36 (12.7)	
	With	552.72 (8.1)	90.19 (9.5)	642.91 (8.7)	46.93%
3	W/O	666.19 (1.7)	718.14 (11.9)	1384.32 (12.2)	
	With	746.07 (0.9)	86.06 (2.8)	832.14 (3.2)	39.89%
4	W/O	861.36 (4.9)	726.88 (7.7)	1588.24 (7.6)	
	With	939.13 (1.3)	83.07 (5.1)	1022.20 (4.7)	35.64%
6	W/O	1268.57 (84.8)	711.90 (13.5)	1980.47 (81.7)	
	With	1325.93 (0.8)	80.55 (4.6)	1406.48 (4.1)	28.98%
8	W/O	1626.12 (3.3)	691.20 (12.1)	2317.32 (11.4)	
	With	1707.23 (4.0)	80.68 (2.2)	1787.91 (4.0)	22.85%
10	W/O	2019.33 (29.3)	680.06 (11.8)	2699.38 (26.6)	
	With	2093.01 (3.3)	86.93 (5.2)	2179.94 (5.4)	19.24%

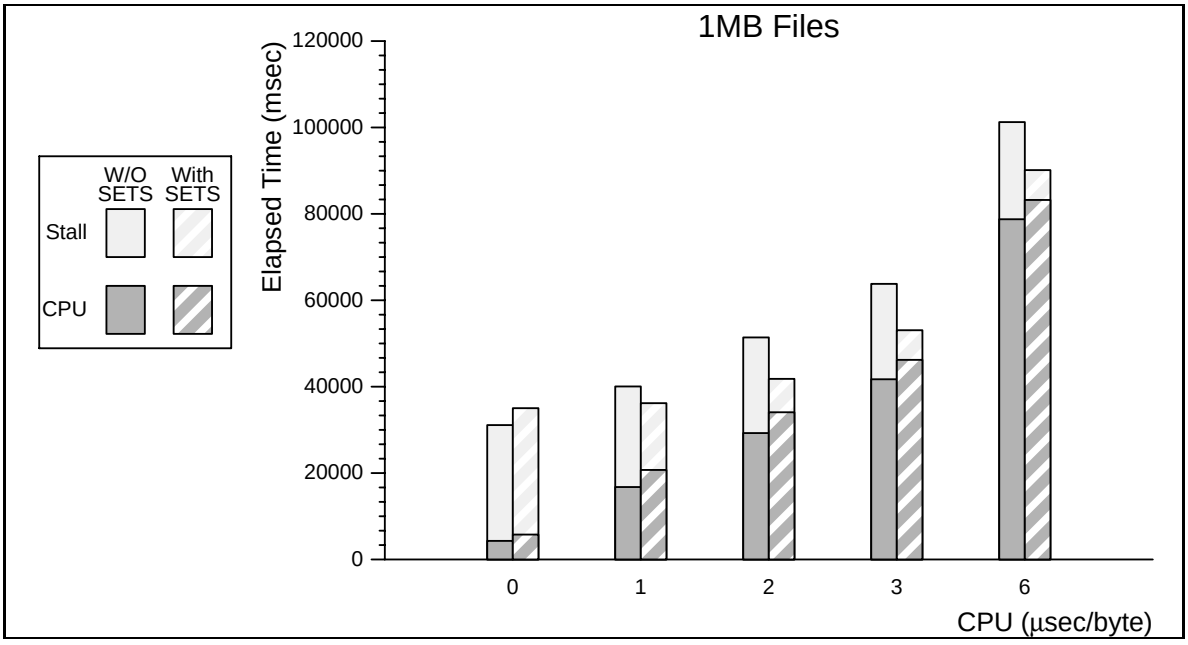
This table presents the results of running the benchmark for different values of application processing per byte, *Comp* in the model in Section 7.1, leaving the other factors at their base level. The metrics in the other columns are described in Section 7.2.2. With additional processing, SETS is able to overlap more of the I/O, reducing the amount of I/O stall time (Stall). When the time to process a file exceeds the time to fetch it, SETS can overlap all I/O except for the time to fetch the first file.

Figure 9.12: Benefits of SETS vs *Comp* for 16KB Files

metrics should not be directly compared with the results of the other NFS tests.

The chief observation is that SETS can exploit user think time by overlapping it with I/O. Thus runs with SETS can make progress while the user is thinking. Runs without SETS cannot, since the system does not know a file will be processed until the file is opened. SETS does not get additional benefit from larger *Think* once the pause covers the fetch of the next file. The time to fetch one 16KB, 64KB, or 256KB file is less than 1 second, 1MB files take less than 2 seconds to fetch. As such, the results for longer values of *Think* are not shown. SETS faces the same limit here as it does with the other tests: it can eliminate all but the time to fetch the first file.

9.2.6 Cache Effects



This graph shows the results of an experiment to test the effect of application processing on the benefit from SETS on 1MB files, with the other factors at their base level. The application was run with different values of computation (*Comp* in the model in Section 7.1, the amount of time (expressed in μsec/byte) to process data. For values of $Comp > 3$, SETS gets no additional benefit since it is already overlapping as much I/O as it can. This can be seen in the graph since the size of the light shadowed portion corresponding to Stall does not decrease for higher values of *Comp*. When $Comp = 0$, SETS is worse since Unix read-ahead achieves most of the benefit of prefetching and SETS introduces additional seek time by introducing contention for the disk head from prefetching multiple files concurrently.

Figure 9.13: Benefit of SETS vs *Comp* for 1MB Files

So far the NFS experiments have not demonstrated one benefit of dynamic sets: the ability to reorder processing of set members. This ability is advantageous when some members are less expensive to obtain than others. As an example, consider a set in which some of the members are cached and others are not. Because it determines the order in which members are yielded to the application and knows the state of the cache, SETS can yield the cached members immediately. One advantage is that the cost of processing these cached objects can be used to hide the I/O of fetching other members, allowing SETS to potentially eliminate all I/O stalls. In earlier experiments, the best SETS could do was to eliminate all stalls but those to fetch the first member's data. A second advantage is that by yielding the cached member immediately, SETS avoids a situation in which the cached data is evicted before it can be used by the application.

Size KB	*T* sec	SETS	User seconds	App milliseconds	Fetch milliseconds	Total milliseconds	Saving in Fetch
16	0	W/O	0.00 (0.0)	195.48 (1.3)	810.69 (29.8)	1006.54 (28.7)	
		With	0.00 (0.0)	240.61 (3.6)	305.60 (14.5)	553.61 (12.3)	62.30%
	1	W/O	12.41 (0.0)	196.60 (1.6)	819.22 (42.2)	13324.74 (66.5)	
		With	12.42 (0.0)	200.43 (0.6)	133.19 (11.8)	12678.10 (11.0)	83.74%
64	0	W/O	0.00 (0.0)	777.93 (3.0)	1995.67 (24.9)	2769.79 (21.2)	
		With	0.00 (0.0)	1181.76 (77.0)	467.47 (32.8)	1650.89 (99.1)	76.58%
	1	W/O	12.37 (0.0)	792.45 (33.7)	2030.94 (34.6)	15094.44 (67.5)	
		With	12.34 (0.0)	819.31 (3.4)	350.59 (5.9)	13443.64 (7.5)	82.74%
256	0	W/O	0.00 (0.0)	3147.70 (46.3)	7809.95 (244.2)	10924.49 (255.3)	
		With	0.00 (0.0)	3968.43 (100.4)	5724.63 (153.0)	9683.92 (109.8)	26.70%
	1	W/O	12.29 (0.0)	3136.72 (45.5)	7775.77 (283.6)	23194.20 (332.9)	
		With	12.39 (0.0)	3625.50 (93.1)	2760.45 (254.5)	18684.45 (288.2)	64.50%
1024	0	W/O	0.00 (0.0)	12498.27 (73.8)	27460.45 (984.0)	39910.29 (950.4)	
		With	0.00 (0.0)	17004.33 (445.0)	19289.79 (271.9)	36173.30 (353.1)	29.75%
	1	W/O	12.37 (0.0)	12598.86 (108.2)	27379.74 (754.0)	52079.85 (792.5)	
		With	12.36 (0.0)	16076.83 (231.9)	11657.49 (841.9)	39945.81 (704.8)	57.42%
	2	W/O	24.91 (0.0)	12528.87 (124.7)	27129.60 (455.5)	64457.43 (420.1)	
		With	24.91 (0.2)	15359.40 (188.1)	9757.72 (270.6)	49970.51 (276.1)	64.03%

This table shows the results of running the benchmark with different values of user think time (*Think* in the model) on NFS files, with the other factors at their base levels. The left column shows the size of the set members, the next column shows the desired duration of the user pause times in seconds (*T* is an abbreviation for *Think*). The metrics in the other columns are explained in Section 7.2.2. The result of this test is that interactive searches give greater opportunity for SETS to hide latency, which one would expect from the model in Section 7.1.

Figure 9.14: Effect of User Think Time on Benefit from SETS for NFS

In order to demonstrate the benefits of reordering to greatest effect, the experiment uses $S = 3$ and $Comp = 3$ in order to ensure that the cost of processing a file completely overlaps the cost to fetch the next file. As a result, the only stalls in runs with SETS result from fetching the first file. In addition, the experiment caches (the data of) one of the members of the set after flushing the buffer caches but before running the application. Because SETS can reorder the accesses to set members, it can use this cached file instead of forcing the application to wait for a fetch, regardless of the file's position in the order of expansion.

Figure 9.15 shows the experimental results. The table shows results for 16KB, 64KB,

Size KB	# in Cache	SETS	CPU milliseconds	Stall milliseconds	Total milliseconds	Savings in Total
16	0	W/O	664.05 (3.8)	656.64 (9.3)	1320.69 (10.9)	
		With	742.22 (2.4)	40.37 (5.1)	782.59 (4.0)	40.74%
	1	W/O	681.16 (55.1)	610.41 (9.8)	1291.57 (52.3)	
		With	745.94 (1.2)	0.27 (0.4)	746.21 (1.1)	42.22%
64	0	W/O	2616.87 (49.9)	1578.78 (13.5)	4195.64 (38.8)	
		With	2904.36 (69.0)	138.88 (14.2)	3043.24 (67.7)	27.47%
	1	W/O	2592.66 (23.3)	1446.98 (20.7)	4039.64 (41.0)	
		With	2907.42 (54.3)	0.56 (0.7)	2907.98 (54.4)	28.01%
256	0	W/O	10463.93 (90.9)	6201.83 (207.2)	16665.76 (236.5)	
		With	11426.54 (102.8)	616.77 (47.8)	12043.31 (105.0)	27.74%
	1	W/O	10357.23 (87.4)	5663.04 (119.1)	16020.27 (135.9)	
		With	11480.31 (88.5)	0.36 (0.4)	11480.68 (88.5)	28.34%
1024	0	W/O	41477.78 (125.9)	21788.86 (448.0)	63266.63 (513.1)	
		With	45727.53 (106.9)	2303.37 (57.4)	48030.90 (124.6)	24.08%
	1	W/O	41454.71 (142.5)	22086.78 (652.0)	63541.49 (719.9)	
		With	46041.39 (94.7)	0.18 (0.3)	46041.57 (94.7)	27.54%

These results show that SETS can exploit the presence of a set member in the cache to eliminate I/O stalls. The test ran with $Comp = 3$ and $S = 3$ and with the other factors at their base levels. The leftmost column shows the size of the set members, the next column indicates whether or not a member was cached. The metrics in the other columns are explained in Section 7.2.2. Note that with no file cached, the SETS idle time is equivalent to the time to fetch one file. If a file is cached SETS can use this file without stalling on I/O, and hide the cost of fetching the other files with application processing. For the runs without SETS, the presence of a cached member reduces the number of fetches and thus the idle time. With 1MB files, however, the cached object is flushed to make room for other set members before the file's data can be used. As a result, the stall time for runs on 1MB files without SETS are not lower when a member is cached.

Figure 9.15: Exploiting Cached Files to Reduce I/O Stalls

256KB, and 1MB files, with and without a pre-cached file. As predicted, the presence of just one member in the cache allows SETS to effectively eliminate I/O stalls. It is not important which file is pre-cached, as long as SETS tests for its presence in the cache before allocating all worker threads to fetching uncached members. The algorithm SETS currently uses fetches the first *limitOpen* members in the order determined by the expansion of the membership specification. As long as the cached member is one of these *limitOpen* members, SETS can use it to avoid the I/O stall.

The 1MB results show the second advantage of reordering: increasing the likelihood of

using cached data before it is evicted. SETS is able to determine the member is cached and yield the object while its data is still in the cache. Without SETS the system does not know the data will be needed and inadvertently flushes the blocks from the cache, since the members are collectively larger than the buffer cache. The system must then re-fetch the evicted data blocks, and as a result does not benefit from the presence of a member in the cache at the start of the test. This problem does not arise for the smaller files, since the entire set fits into the buffer cache. One can see this effect in Figure 9.15 since the stall time for 1MB files without SETS does not decrease by the amount of time to fetch one file as it does for smaller files.

Figure 9.16 shows the results of this experiment run for a different set of factor levels, which allow a more direct comparison with the other experimental results. These tests used 1 server, $Comp = 3$, and sets of 12 1MB files. Because all accesses are going to the same server disk, prefetching several files increases the average cost of an I/O. As a result, the cost of fetching a file is larger than the multi-server case, and in particular is longer than the processing of a file when $Comp = 3$. Caching one file does not eliminate idle time, since the application still blocks on the fetch of the second file. Caching both the first and second files, however, does allow SETS to effectively eliminates idle time, as shown in Figure 9.16[2] Alternatively, runs of the experiment on a single disk, with one object cached, and with values of $Comp > 6$ also allow SETS to eliminate I/O stalls by overlapping the fetch of the second file with the processing of the first. However, the results of such tests are not shown here.

9.2.7 Bandwidth

This section describes the results of experiments to see how the benefits of SETS are affected by lower bandwidth connections. There are two reasons for exploring the benefits of SETS on low bandwidth networks. First, one of the principle benefits of prefetching, use of parallel fetches, assumes that there is sufficient bandwidth to support concurrent reads. As one's bandwidth consumption approaches the network's peak rate, performance will drop off rapidly due to collisions and overruns. Second, many users access data in their DFS remotely; either from home or from a mobile client. It is not immediately obvious how dynamic sets will perform for these types of situations.

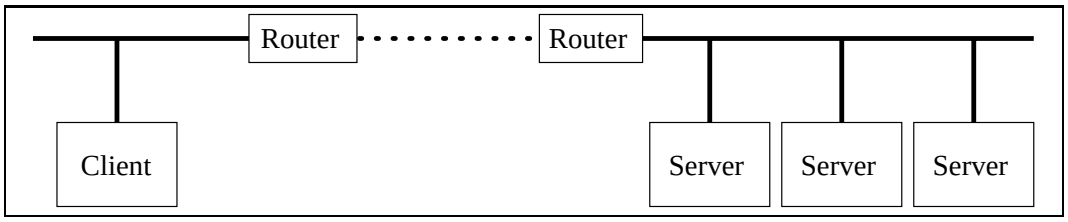
This experiment consists of tests on 2 slow networks, Wavelan and SLIP. The topology for these tests is shown in Figure 9.17. The client and servers were connected by two routers which communicated via a low bandwidth link. The link for the Wavelan tests was a 2Mbps wireless network and the routers were NCR WavePoint access points set

[2]In this second experiment, I selected which members to pre-cache in order to ensure that the runs without SETS would utilize the data before it was evicted.

# in Cache	SETS	CPU milliseconds	Stall milliseconds	Total milliseconds	Savings in Total
0	W/O	41488.74 (149.6)	22152.17 (644.8)	63640.91 (607.7)	
	With	46028.64 (110.6)	7344.31 (233.4)	53372.95 (253.9)	16.13%
1	W/O	41227.96 (140.5)	20484.02 (896.7)	61711.97 (904.6)	
	With	45915.63 (143.0)	3963.64 (209.9)	49879.28 (211.8)	19.17%
2	W/O	40977.32 (90.6)	18472.38 (381.1)	59449.70 (373.8)	
	With	45384.06 (112.1)	1.52 (1.0)	45385.57 (112.5)	23.66%
3	W/O	40707.46 (70.7)	16795.09 (494.9)	57502.55 (485.9)	
	With	44704.84 (81.8)	1.64 (0.6)	44706.48 (81.9)	22.25%

This table shows how SETS can take advantage of cached set members to effectively eliminate I/O stalls. The left column shows the number of members that were cached prior to starting the application ($Size$ = 1MB, $Comp$ = 3, and the other factors are set to their base level). The metrics in other columns are described in Section 7.2.2. Although the overall savings are not impressive, the idle times drop to zero within experimental error. The actual values are less than one hundredth of a percent of the idle times for the runs without SETS.

Figure 9.16: Benefits of SETS When Some Members Are Cached



This figure shows the network topology used for the low bandwidth test. The solid lines are 10Mbps 10Base2 Ethernet links. The client and servers are on different Ethernet segments. Connecting the two segments are two routers which forward packets from the Ethernet over some slower medium (the dashed line) to the other Ethernet segment. For Wavelan, the routers are NCR WavePoint Wavelan routers. For SLIP, the routers are 2 laptops running Mach 2.6 connected via 14400baud modems.

Figure 9.17: Network Topology for Bandwidth Experiments

up to minimize outside interference. The routers for the SLIP test were two laptop computers connected via 14400bps modems and a phone line. Although the laptops may have higher overhead than specialized router hardware, they are able to route packets faster than the SLIP connection can carry them, and the added overhead should not disturb the test results. Although the modem does compress data, it still takes over 8

seconds to transmit 16KB files.

Band	*S*	SETS	CPU milliseconds	Stall milliseconds	Total milliseconds	Savings in Total
10Mbps	1	W/O	271.82 (3.5)	719.13 (17.0)	990.95 (16.6)	
	1	With	320.45 (7.8)	221.82 (21.4)	542.27 (14.1)	45.28%
	2	With	349.17 (6.1)	68.80 (13.1)	417.96 (11.2)	55.27%
	3	With	352.80 (4.2)	55.56 (10.9)	408.36 (8.0)	56.89%
2Mbps	1	W/O	285.42 (18.8)	1682.38 (36.7)	1967.79 (52.8)	
	1	With	301.32 (3.3)	965.65 (103.1)	1266.97 (100.8)	35.61%
	2	With	306.59 (5.7)	931.55 (117.4)	1238.14 (117.6)	37.08%
	3	With	311.59 (19.5)	895.58 (56.2)	1207.17 (56.4)	38.65%
14Kbaud	1	W/O	1190.85 (330.9)	114501.29 (4618.2)	115692.13 (4892.0)	
	1	With	990.04 (348.6)	107108.14 (5406.2)	108098.18 (5728.8)	6.56%
	2	With	785.87 (209.5)	105376.11 (3998.5)	106161.98 (4134.2)	8.24%
	3	With	876.47 (272.4)	105448.80 (3918.6)	106325.27 (4103.1)	8.10%

This table shows the results of multi-server tests run on networks with different bandwidths, with the other factors at base levels. The left column shows the peak bandwidth (*Band*) of the link connecting the client and servers. Other columns show the number of servers (*S*), use of SETS, amount of processing (CPU), client CPU idle time (Stall), and total elapsed times in milliseconds. The rightmost column shows the savings in total elapsed time due to SETS, calculated with Equation 7.9. Although the magnitude of the variance in fetch times for SLIP is much higher than for the Wavelan numbers, the coefficient of variation is less than 10%. The runs on different networks use different values of *limitOpens* than the other tests to avoid overruning the low bandwidth networks. The numbers for 10Mbps (Ethernet) are pulled from Figure 9.8 for comparison as an aid to the reader.

Figure 9.18: Benefits of SETS for Different Bandwidth Links

Figure 9.18 shows the results of the multi-server experiment run on Wavelan and SLIP. Because the experiment is concerned with the impact of lower bandwidth on the benefit of concurrent fetching, the figure only shows the results for SETS when $S > 1$. The difference between runs without SETS for different values of S was nominal. In order to avoid overrunning the slow link, the Wavelan tests used $limitOpens = 3$ and the SLIP tests used $limitOpens = 1$. The numbers for 10Mbps (Ethernet) are pulled from Figure 9.8 for comparison as an aid to the reader.

The basic result of the tests is that dynamic sets does offer a small benefit even on low bandwidth links. SETS reduces idle time by over 40% for the Wavelan tests, and elapsed

time by close to 36%. The difference for the SLIP tests is much smaller, just over 6%. Although the numbers are close, they are still statistically different at a significance level of 5%.

A more disturbing trend is that SETS does not seem to benefit from parallel fetches. On SLIP this is not surprising, since the network is so much slower than any other component in the system. However, even runs on the Wavelan network get relatively little benefit from concurrent fetches. The reason for this is that one disk can produce enough data to consume all of the network bandwidth. The server's disk bandwidth is measured at around 400KBps, while the network's peak bandwidth is 512KBps. Thus there is little opportunity to decrease latency by running concurrent fetches.

The low performance benefits from SETS for the SLIP runs is due to the overwhelming contribution of the network to I/O latency. The disk takes around 20 milliseconds to read 16KB, but the network takes over 8 seconds to transmit it at 14.4Kbaud (with compression enabled). Although there is little benefit from concurrent fetches, SETS can benefit by overlapping I/O and computation, particularly user think time. Figure 9.19 shows the results of interactive search run on SLIP. Recall that interactive searches have pauses between each file during which the user reads the file's data. Since SETS can prefetch a file before it is requested, it can overlap I/O latency with these pauses to reduce the amount of time a user must wait for data. The table shows that for even a modest pause of 6 seconds per file, SETS is able to substantially reduce I/O times and thus the runtime of the application.

Figure 9.19 also shows a puzzling trend: runs without SETS benefit from higher values of $Think$. It may be the case that spreading out requests reduces the load on the SLIP connection. Unfortunately, the test environment does not permit finer analysis of these results, and I am unable to determine a better explanation for this phenomenon at this time.

9.3 Conclusion

In summary, these experiments demonstrate the SETS do offer significantly improved performance for search on NFS. Further, SETS is able to realize all three of the potential performance improvements of prefetching in a variety of situations: exploitation of parallelism via concurrent fetches, overlapping computation and I/O, and increasing the utilization of servers and the network. In many cases, SETS is able to offer near-optimal reductions in latency (up to limits of the implementation).

This experiment also shows the benefits of reordering. By using the system's knowledge of the cache, SETS can detect a member in the cache and yield that member immediately. By doing so, SETS can eliminate I/O stalls in many cases, even for 1MB files.

Think seconds	SETS	User seconds	App milliseconds	Fetch milliseconds	Total milliseconds	Savings in Fetch
0	W/O	0.00 (0.0)	194.80 (0.7)	106702.74 (5904.3)	106467.61 (5978.2)	
	With	0.00 (0.0)	194.55 (1.3)	108380.45 (5961.1)	108299.15 (6065.6)	-1.57%
1	W/O	12.35 (0.0)	195.16 (1.1)	92078.29 (4529.4)	104004.91 (4409.0)	
	With	12.34 (0.0)	194.84 (1.2)	93174.21 (2747.7)	105374.40 (2909.7)	-1.19%
2	W/O	24.91 (0.1)	194.25 (0.9)	79149.41 (4094.4)	103991.97 (4169.7)	
	With	24.97 (0.1)	194.99 (0.9)	82618.17 (4086.1)	107262.11 (4134.6)	-4.38%
4	W/O	49.92 (0.4)	194.29 (1.6)	67873.44 (2056.7)	117664.90 (1858.8)	
	With	50.12 (0.2)	196.17 (1.2)	61881.91 (5956.1)	111437.19 (5850.7)	8.83%
6	W/O	75.46 (0.3)	194.85 (0.9)	69560.18 (3371.1)	144297.56 (3526.4)	
	With	76.32 (3.1)	195.40 (1.6)	39732.98 (5009.3)	115596.14 (4312.3)	42.88%

This table shows the results of running the benchmark with different values of *Think* on 16KB NFS files accessed over a SLIP link, with the other factors at their base level. The left column shows the values of user think time, the metrics in the other columns are explained in Section 7.2.2. The chief observation is that SETS can offer substantial benefit for interactive searches even with very little bandwidth by overlapping I/O latency with user think time.

Figure 9.19: Benefits of SETS vs *Think* over SLIP

However, these experiments do point out a deficiency in the implementation. As shown in Figure 9.7, SETS can be less efficient at I/O because it sometimes does not access data sequentially. Sequentiality is lost because of the way SETS prefetches files: one thread per file means accesses from different files are interleaved. A small change to the SETS prefetching engine should address this problem, but the change is left as future work.

Chapter 10

Evaluation: Search on local file systems

The third experiment examines search on the local file system (LFS). There are two goals in determining the benefit of SETS in this domain. The first goal is to evaluate the behavior of the SETS prefetching engine under a different domain than that for which it was designed. The prefetching engine was designed to prefetch remote objects in a distributed system efficiently, and prioritizes remote accesses (prefetching) over local accesses (prereading). In particular, SETS only prereads data from disk into the cache's pin-space. As a result, a set of local files is a worst case for SETS. Second, it may be the case in practice that some portion of the set is either stored on the local file system or already cached when the set is opened. As a result, the performance of SETS on local data is relevant to determining the performance of SETS in general.

The LFS domain is important because many users store their personal data on their local file system even when a DFS is available. In addition, at the heart of every WWW, NFS, or other server is a local file system which stores the data. This domain is different from both GDIS and DFS because it involves no network communication. As a result, all I/O requests are known to the file system, and the latency to access data is often smaller than that in GDIS and DFS (depending on load and the hit rate in the server's in-memory cache). Further, since there are fewer components involved, the performance of the application is more directly tied to the performance of the disk. Thus low level issues such as data layout and the disk's parameters are likely to have a greater impact.

The representative example of an LFS examined here is the Unix Fast File System (FFS) running on an array of independent parallel disks (Just A Bunch Of Disks, JABOD). Unix FFS and variants are commonly used in Unix systems, and are similar to other PC and workstation file systems. JABOD is the simplest form of a disk array. Data is stored in files; each file resides on exactly one disk, but different files can be stored on different disks. Examples of potential uses of FFS on JABOD are the local file system for GDIS and DFS servers, workstations or PCs with multiple disks, and object repositories.

Factors	Name	Description
Primary	Cardinality	Number of objects in a set
	File size	Average size of candidate objects
	Disks	Number of disks holding candidate objects
	Cache	Effects of incidental cache hits
	Think	Inter-block and inter-file processing time
Secondary	Disk speed	Bandwidth, latency, seek time
	Bus Bandwidth	Usually sufficient to support up to 6 disks
	File system	Block size, number of buffers
	Prefetch parameters	Described in Chapter 6
	Cache Size	Number of in-memory buffers and inodes

This table lists the primary and secondary factors affecting the performance of search on the FFS. The impact of the primary factors is examined through experiments, whose results are presented below. The secondary factors have a smaller effect on the elapsed time of the search, or are independent of dynamic sets, and are not directly examined in this dissertation.

Figure 10.1: Factors Affecting Search Performance on the FFS

An example of this type of repository is the QBIC image database[61]. QBIC stores the images as file system objects. A query to QBIC returns the names of the files containing the desired images, and the system then returns the files themselves to the client for processing on demand. One can imagine modifying QBIC to use SETS: forming a dynamic set using the names returned from the query, and having SETS prefetch the files on behalf of the application. Other servers which provide indexed access to technical reports, for instance, or video libraries could use SETS in a similar fashion.

Figure 10.1 describes the factors affecting the performance of search in an LFS. These factors are similar to those affecting search on NFS, with the obvious exception of the network parameters. Bandwidth for JABOD refers to bus bandwidth, which is typically sufficient to support parallel access to a small number of disks. Since the tests presented below limit concurrent accesses to at most 3 disks, bus bandwidth is not a significant factor in determining system performance for this experiment.

10.1 Test Methodology

Although the performance characteristics of FFS differ from NFS, the application mix is roughly identical. In addition, FFS on JABOD provides another good domain in which to perform low level analysis to better understand the performance of SETS. For these

reasons, it makes sense to use the same test methodology used in the NFS experiments. A benefit of this strategy is that one can compare the results for similar tests on both systems for further insights into the benefit from SETS.

10.2 Results of FFS Experiments

The machine on which these tests were run is a DECStation 5000/200 with 32 MB of RAM running the Mach 2.6 operating system. The disks were 3 Digital RZ24 3.5" hard drives, connected to one SCSI controller. The machines have a hardware cycle counter with which the kernel can accurately time events to within a few microseconds. The machine was lightly loaded: only the user running tests was logged in during the tests. Although the machine was not booted single user, it was isolated from the network to avoid interference. Since this machine is normally shared among several users, it was rebooted before each series of tests to ensure a clean test environment.

This experiment uses the same benchmark as the NFS experiment, but accesses the files off local disks instead of off NFS servers. The numbers are the averages of 10 runs.[1] The factor levels for the base test used sets of 12 16KB files stored on 1 disk, 0 seconds of think time per file (non-interactive search), 1 microsecond of processing/byte, and a cold buffer cache (no objects cached), which are similar to the base levels used in the NFS experiment. The tests used the same SETS cache parameters as the WWW and NFS tests, shown in Figure 7.3. The file system block size was 8KB.

10.2.1 Cardinality

Figure 10.2 shows the results from running the benchmark on different sized sets of 16KB files. The results show the increase in CPU overhead from SETS and prefetching, the significant decrease in idle time as a result of prefetching for $N \geq 4$, and the resulting decrease in run time from using SETS. These results are also graphed in Figure 10.3.

Compared with the NFS results in Figure 9.2 and Figure 9.3, prefetching small files from disk does not produce as big of a benefit. Since the I/O latencies are much smaller, there is less of an opportunity to benefit from prefetching. The CPU times are also lower, which is likely due to the smaller overhead to setup a SCSI disk request as compared with an NFS request. A related observation is that the difference in CPU times between

[1] Although the machine was isolated for the tests, occasionally some background activity from Unix daemons such as `update` would consume CPU or disk bandwidth, resulting in significantly higher numbers. No more than two out of 15 runs were anomalous; more often all 15 runs had low variance. Since these anomalies happened both with and without SETS, these abnormal runs were discarded from the results.

N	SETS	CPU milliseconds	Stall milliseconds	Total milliseconds	Savings in Total
1	W/O	21.58 (0.3)	9.37 (0.0)	30.95 (0.3)	
	With	23.56 (0.4)	8.97 (0.2)	32.53 (0.4)	-5.11%
2	W/O	39.27 (0.6)	18.59 (0.4)	57.86 (0.5)	
	With	43.48 (0.2)	12.93 (0.0)	56.42 (0.2)	2.49%
4	W/O	75.69 (1.7)	36.64 (1.2)	112.32 (0.6)	
	With	84.33 (0.6)	16.70 (0.2)	101.03 (0.7)	10.05%
8	W/O	145.81 (0.4)	130.92 (1.9)	276.73 (1.9)	
	With	171.28 (0.8)	50.75 (5.2)	222.04 (5.4)	19.76%
12	W/O	217.47 (1.4)	186.92 (5.9)	404.39 (6.1)	
	With	255.98 (1.4)	60.00 (4.1)	315.99 (3.7)	21.86%
16	W/O	287.70 (1.0)	258.21 (6.6)	545.91 (6.3)	
	With	341.16 (2.0)	64.22 (4.0)	405.39 (4.1)	25.74%

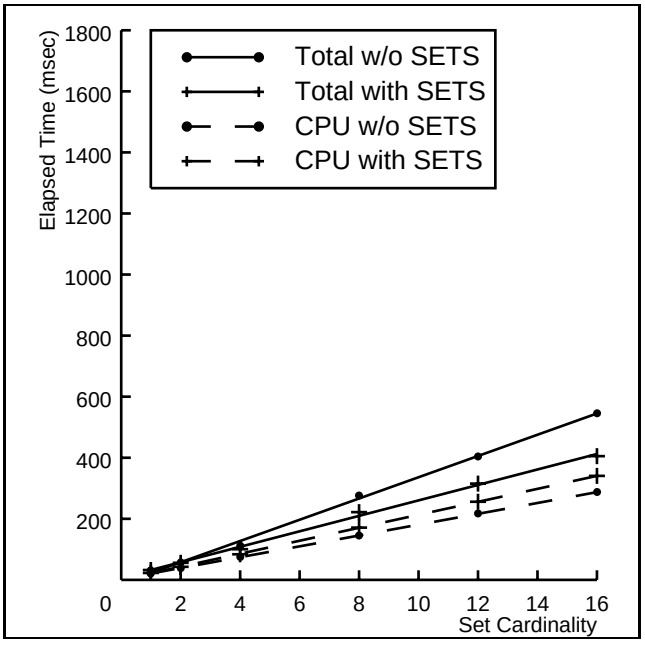
This table shows the effect of set cardinality on the benefit from dynamic sets. The left column (N) shows the number of 16KB files processed by the benchmark program, the other factors were set to their base level. The metrics in the other columns are described in Section 7.2.2. The numbers are the mean of 10 runs, standard deviations are presented in parenthesis. Negative savings mean SETS degraded performance. The chief result of this experiment is that SETS does offer benefit to this domain, but the magnitude of the savings are smaller than in DFS.

Figure 10.2: Effect of Set Cardinality on Benefit from SETS for FFS

runs with and without SETS is smaller. Based on this observation I conjecture that some of the increase in CPU for the runs with SETS is due to context switching and prefetching overhead, which is endemic to any prefetching mechanism. Thus the overhead due exclusively to SETS is smaller than the difference in CPU between runs with and without SETS.

The benefit from SETS in this experiment has two sources, the overlap of CPU and I/O and the ability to aggressively prefetch. Overlapping CPU and I/O produces the chief benefit for this test; there is little opportunity to exploit parallelism or pipelines in the I/O channel. SETS also provides benefit by prefetching aggressively, which means it can queue reads for a file's data before the file is opened. FFS only begins to read-ahead after the application requests the first 2 blocks, and thus gets no benefit from read-ahead for 16KB files.

One interesting observation that I made while running the experiment is that the cost of I/O, and thus the benefit of SETS in this situation, seems to depend on the layout of the files on disk. For instance, I reran this test using identical 16KB files stored in a different directory, allowing the file system to determine placement of the files' inodes and data



This graph shows the cost and benefit of SETS for different sized sets of 16KB disk files. The points are the experimental results from Figure 10.2, with lines fitted via regression with a correlation coefficient of greater than .998 in all cases. The dots show the results without SETS, the pluses those with SETS. The solid lines show the total elapsed time and the dashed lines show the amount of CPU, the difference between the solid and dashed lines is the idle time. From the graph, one can see the increase in CPU usage due to SETS, but also the larger reduction from overlapping computation and I/O. The result is that SETS can reduce the run time for every file in the set, and thus get more benefit as the set grows in size. To allow for comparison, this graph uses the same scale as the graph of the NFS cardinality test results, shown in Figure 9.3.

Figure 10.3: Benefit of SETS vs Cardinality on FFS

blocks. The only difference between the two sets of 16KB files was the cylinder group in which the files were stored, and the layout of the blocks within the group. In this second test, the benefits from SETS were uniformly better than shown here, particularly for $N < 8$, although the cost of I/O was higher both with and without SETS. As an example, the base case ($N = 12$) for runs on this second set of files saw stall times 358.36 (14.58) without SETS and 174.87 (8.55) with SETS, while the total savings were 27.05%. The CPU costs were equivalent between the tests.

Unfortunately, the influence of data layout is larger on the cost of I/O as captured by the Work metric than is the influence of SETS. As a result, it is difficult to determine the effect of prefetching in this domain on the cost of I/O. To a first approximation, however, it appears that the chief effect of SETS' prefetching on the cost of I/O is the

lack of sequentiality of access, which introduces extra seeks.

10.2.2 File Size

Size KB	SETS	CPU milliseconds	Stall milliseconds	Total milliseconds	Savings in Total
1	W/O	25.40 (0.6)	78.66 (5.7)	104.05 (5.8)	
	With	36.18 (1.1)	37.70 (5.7)	73.88 (5.3)	29.00%
4	W/O	62.23 (0.2)	80.03 (2.3)	142.27 (2.2)	
	With	74.67 (0.6)	46.22 (4.1)	120.88 (4.3)	15.03%
16	W/O	217.09 (1.3)	192.82 (2.0)	409.90 (2.4)	
	With	255.53 (1.2)	58.13 (2.8)	313.66 (3.0)	23.48%
64	W/O	849.25 (1.8)	1037.24 (2.3)	1886.50 (1.5)	
	With	941.57 (4.7)	1014.38 (26.4)	1955.95 (24.7)	-3.68%
256	W/O	3410.52 (59.9)	4768.95 (49.0)	8179.47 (107.8)	
	With	3463.56 (42.5)	5559.59 (65.6)	9023.14 (83.6)	-10.31%
1024	W/O	13706.57 (53.8)	15737.23 (47.1)	29443.80 (41.6)	
	With	13770.13 (51.3)	16300.27 (74.1)	30070.39 (53.0)	-2.13%

This table shows the effect of member size on the benefit from dynamic sets. The left column shows the member size in KB for each test (*Size* in the model in Section 7.1), the other factors were set to their base levels. The numbers are the mean of 10 runs, standard deviations are presented in parenthesis. The metrics in the other columns are described in Section 7.2.2. This table shows that SETS does improve performance for small files typical in Unix, but degrades performance for large files.

Figure 10.4: Effect of Member Size on Benefit from SETS for FFS on JABOD

Figure 10.4 shows the results of a test to determine the effect of file size on the benefit from SETS. The results show two different trends, depending on file size. For small files, those 16KB in size and smaller, SETS reduces the idle time by aggressive prefetching. This results in overall reduction in runtime of greater than 15%. The primary reason for the reduction in idle time is that SETS anticipates access to the file before it is opened, and can prefetch. FFS cannot know the file will be accessed, and must read its data synchronously.

For large files, those larger than 64KB, SETS increases the idle time seen by the benchmark in addition to increasing the amount of computation. As a result, SETS increases

runtime by prefetching! There are two reasons why SETS performs poorly for large disk files. One reason, as has been mentioned several times previously, is the lack of sequentiality resulting from the way SETS queues concurrent reads. A more influential reason, however, is that SETS only reads a prefix of `nprefix` blocks into the cache for these large files, and for this experiment `nprefix` = 8. When an application processes a file, it will miss on blocks 9 through n, triggering reads by the file system. Thus a majority of reads for large files use the same code with or without SETS, and SETS will obtain little benefit from prefetching over read-ahead. As a result, any positive or negative impact of SETS on the first `nprefix` reads will be dwarfed by the cost to read the other blocks, and the effect of SETS on performance will asymptotically approach zero as the file size increases.

The behavior for files larger than 16KB and no larger than 64KB is slightly different than either of these two groups (represented by 64KB files in Table 10.4). For these files, SETS does offer some reduction in idle time, but not enough to overcome the overhead in CPU and seek time that it introduces. The savings in idle time is lower than that for smaller files because FFS is prefetching 6 of the 8 data blocks per-file through read-ahead during runs without SETS. FFS is thus getting the benefit of prefetching for most of the reads, while SETS is losing some of the benefit of prefetching by increasing the cost of performing I/O. However, with additional opportunities for prefetching, such as $Think > 0$, SETS will be able to further decrease latency for these files while simple read-ahead may not.

10.2.3 SETS on Small and Medium-Sized Files

Because SETS exhibits such different behaviors on small and large files, the remainder of the chapter will discuss each case separately. As mentioned above, all of the benefits from SETS apply to the first `nprefix` buffers of a file. For files with fewer than `nprefix` buffers, use of SETS results in performance improvements in many situations. The following subsections show the effect of parallel fetches, increased computation, user think time, and cache state on the benefit from sets. These subsections present the results for 16KB and 64KB files; smaller files should do as well as these since the FFS does not use read-ahead for files smaller than 16KB.

10.2.3.1 Multiple Disks

Figure 10.5 shows the results of parallel fetches on the benefit of SETS for 16KB and 64KB files. As predicted, SETS can utilize all the disks to reduce idle and elapsed times by fetching multiple files in parallel. As with NFS, fetching from multiple disks allows S reads to be performed in close to the time to perform one, and reduces the load on

Size KB	*S*	SETS	CPU milliseconds	Stall milliseconds	Total milliseconds	Savings in Total
16	1	W/O	217.00 (0.6)	186.47 (6.0)	403.47 (6.2)	
		With	254.66 (1.0)	63.35 (3.7)	318.01 (3.4)	21.18%
	2	W/O	217.99 (2.1)	270.60 (6.5)	488.59 (7.3)	
		With	256.01 (0.8)	46.14 (1.9)	302.15 (1.3)	38.16%
	3	W/O	219.32 (3.2)	248.68 (10.0)	468.00 (12.8)	
		With	257.35 (1.3)	19.82 (0.3)	277.17 (1.2)	40.78%
64	1	W/O	848.88 (1.3)	1044.83 (9.5)	1893.71 (9.1)	
		With	948.40 (4.6)	950.26 (27.1)	1898.67 (24.8)	-0.26%
	2	W/O	852.40 (3.6)	1029.61 (18.5)	1882.01 (17.9)	
		With	976.75 (7.8)	291.57 (56.1)	1268.32 (49.2)	32.61%
	3	W/O	849.29 (2.8)	1099.01 (9.1)	1948.30 (9.2)	
		With	1006.44 (76.3)	243.89 (20.4)	1250.33 (66.0)	35.82%

This table shows the results of running the benchmark on sets of 16KB and 64KB files stored on multiple disks ($S \geq 1$), with the other factors at their base level. The left column shows the size of the file, the next column shows the number of disks on which the files were stored. The metrics in other columns are described in Section 7.2.2. The numbers are the mean of 10 runs, standard deviations are presented in parenthesis. The table shows that SETS can reduce latency through parallel disk reads, and thus produce substantial benefit even for 64KB files.

Figure 10.5: Effect of Parallelism on Benefit from SETS for FFS on JABOD

Size	*S*		
KB	1	2	3
16	89.38 (2.4)	72.65 (3.6)	60.99 (2.0)
64	615.24 (12.7)	344.09 (17.3)	328.76 (27.9)

This table shows the average time in milliseconds for SETS to read a file from disk. By comparing the times for different values of S, one can see how distributing the load over multiple disks reduces the I/O penalty SETS incurs from increasing the average seek time per I/O.

Figure 10.6: SETS Prefetch Time vs Degree of Parallelism

and therefore the contention for a single disk head. Figure 10.6 shows the average time it took SETS to read a file from disk for different values of S. As one can see, spreading the load onto multiple disks lowers the cost of I/O.

The savings which result from fetching from 3 disks is little more than that from two disks. Thus it is likely that SETS would not derive significant benefit from more than 3

disks. This means SETS can achieve near optimal performance without requiring special high-bandwidth bus hardware. However, it may be the case that reducing the prefetching parameter `limitOpens` to equal the number of disks would reduce the overhead in seek time without limiting the ability of SETS to reduce idle time. Initial experiments would indicate that this is the case, and exploring a mechanism to dynamically adjust `limitOpens` may prove fruitful.

10.2.3.2 Computation

Size KB	*Comp*	SETS	CPU milliseconds	Stall milliseconds	Total milliseconds	Savings in Total
16	0	W/O	23.43 (0.8)	322.05 (9.0)	345.49 (8.7)	
		With	35.33 (0.9)	257.88 (9.0)	293.21 (9.0)	15.13%
	1	W/O	218.82 (3.5)	187.36 (7.0)	406.17 (8.2)	
		With	255.13 (1.2)	55.18 (3.1)	310.30 (2.5)	23.60%
	2	W/O	410.21 (0.7)	171.15 (5.4)	581.36 (5.1)	
		With	466.43 (38.1)	44.04 (3.6)	510.46 (39.5)	12.20%
64	0	W/O	63.63 (1.3)	1729.73 (19.0)	1793.36 (18.3)	
		With	108.77 (2.9)	1588.36 (10.9)	1697.13 (11.8)	5.37%
	1	W/O	849.45 (1.6)	1038.31 (8.1)	1887.76 (8.2)	
		With	943.51 (7.6)	982.95 (36.7)	1926.46 (30.5)	-2.05%
	2	W/O	1693.98 (3.3)	404.10 (24.7)	2098.08 (24.6)	
		With	1775.61 (1.9)	343.32 (17.8)	2118.93 (16.3)	-0.99%
	6	W/O	4785.39 (3.8)	285.08 (7.5)	5070.48 (7.2)	
		With	4870.59 (3.1)	287.78 (3.2)	5158.37 (5.6)	-1.73%

This table shows the effect of application processing (*Comp*) on the benefit from dynamic sets. The left column shows the size of the set members. The next column shows *Comp*, the amount of processing per byte performed by the application (in μsec/byte). The metrics in the other columns are described in Section 7.2.2. The numbers are the mean of 10 runs, standard deviations are presented in parenthesis. Although larger values of *Comp* allow SETS to hide more latency, it also increases the overall runtime resulting in smaller savings. Since FFS's read-ahead also benefits in this way, SETS does not provide additional savings for larger values of *Comp* for 64KB files.

Figure 10.7: Effect of Application Processing on Benefit from SETS for FFS on JABOD

Figure 10.7 shows the results of running the benchmark with different amounts of computation per byte for 16KB and 64KB files. With larger values of *Comp*, SETS can further

decrease the idle time by overlapping I/O with longer amounts of application processing in the 16KB case. However, spending more time processing data diminishes the relative contribution of I/O to runtime, reducing the benefit of SETS as a percentage of elapsed time.

For the runs without SETS on 64KB files, increasing the application's computation reduces the likelihood of stalling on read-ahead buffers, which in turn reduces the idle time seen by the application. As a result SETS gets little or no benefit over read-ahead for large values of *Comp*, and as a result of overhead actually runs a little slower (although the difference is not statistically significant).

As stated before, the chief reason SETS does not perform better for 64KB files is that the benefit it offers from prefetching is outweighed by the cost it introduces by interleaving reads of different files. It is likely that fixing this behavior and reducing `limitOpens` to the number of disks will result in lowering the overhead, and thus decreasing the latency seen by applications that use SETS.

10.2.3.3 User Think Time

Size KB	*Think* seconds	SETS	User seconds	App milliseconds	Fetch milliseconds	Total milliseconds	Savings in Fetch
16	0	W/O	0.00 (0.0)	195.37 (2.2)	213.89 (15.8)	411.86 (18.5)	
		With	0.00 (0.0)	218.58 (2.0)	92.43 (6.9)	315.23 (6.5)	56.79%
	1	W/O	12.33 (0.0)	205.56 (32.2)	245.04 (32.3)	12727.97 (30.6)	
		With	12.38 (0.0)	197.16 (0.8)	74.84 (3.7)	12616.22 (5.3)	69.46%
64	0	W/O	0.00 (0.0)	791.20 (3.3)	1104.01 (6.1)	1888.17 (6.8)	
		With	0.00 (0.0)	838.62 (7.6)	1086.03 (26.9)	1922.67 (21.5)	1.63%
	1	W/O	12.35 (0.0)	792.24 (3.0)	1117.81 (75.2)	14199.60 (81.7)	
		With	12.33 (0.0)	796.38 (37.3)	376.97 (4.4)	13448.16 (33.4)	66.28%

This table shows the results of running the benchmark with different values of *Think* on 16KB and 64KB files, with the other factors at their base level. The metrics in the other columns are described in Section 7.2.2. Note that the savings is in terms of Fetch, not overall runtime, since this test uses interactive search. As pointed out in Section 7.1.1, SETS can take advantage of larger values of *Think* while read-ahead cannot, and so greatly reduces latency.

Figure 10.8: Effect of User Think Time on Benefit from SETS for FFS

Figure 10.8 shows the benefit of SETS for interactive search tools running on local disk

files. This experiment shows that SETS can overlap I/O with user think times to reduce the aggregate latency of accessing the set members. As with the NFS experiments, the time for I/O is bounded below by the time to fetch one member. In addition, SETS does not derive additional benefit for $Think > 1$ since it takes less than one second for SETS to read a file off the disk.

Size KB	# in Cache	SETS	CPU milliseconds	Stall milliseconds	Total milliseconds	Savings in Total
16	0	W/O	602.77 (0.8)	168.99 (7.5)	771.76 (7.3)	
		With	646.55 (1.0)	50.10 (7.8)	696.66 (8.0)	9.73%
	1	W/O	603.29 (0.6)	130.54 (7.0)	733.83 (7.2)	
		With	647.13 (0.7)	15.42 (5.5)	662.56 (5.2)	9.71%
	2	W/O	604.84 (4.6)	137.52 (19.4)	742.36 (19.7)	
		With	647.13 (3.2)	0.00 (0.0)	647.13 (3.2)	12.83%
64	0	W/O	2465.96 (2.1)	331.90 (15.6)	2797.86 (15.7)	
		With	2552.64 (3.8)	307.13 (16.8)	2859.77 (14.8)	-2.21%
	1	W/O	2464.43 (2.7)	295.09 (11.3)	2759.52 (11.0)	
		With	2554.99 (5.3)	33.99 (38.0)	2588.98 (33.0)	6.18%
	2	W/O	2460.10 (7.5)	271.64 (17.5)	2731.74 (21.9)	
		With	2547.78 (1.4)	0.00 (0.0)	2547.78 (1.4)	6.73%

This table shows how SETS can take advantage of cached set members to effectively eliminate I/O stalls when $Comp = 3$ through reordering. The left column shows the size of the members, the test was run on 16KB and 64KB files with the other factors at their base level. The next column is the number of members that were cached prior to starting the application (out of 12). The metrics in the other columns are described in Section 7.2.2. Although the overall savings are not impressive, the idle times for runs with SETS drop to zero when more than one member is cached.

Figure 10.9: Effect of Warm Cache on Benefit from SETS for FFS on JABOD

One anomaly, higher App time for runs with SETS when $Think = 0$, results from measuring the amount of application processing (App) via elapsed time. When $Think = 0$, the computation to perform I/O directly competes for the CPU with the application, resulting in longer elapsed times to perform the same amount of application computation. When $Think > 0$, some of the computation of prefetching is performed while the application is sleeping, reducing the competition for the CPU and thus the elapsed time measured by App. Although both cases do overlap computation and I/O to some extent, this anomaly is more pronounced for SETS which prefetches more aggressively. This ef-

fect can also be seen in the NFS interactive search tests, shown in Figures 9.14 and 9.19. This anomaly is only apparent because the experiment used an elapsed time profiler to measure App, as opposed to counting the number of application instruction that were executed.

10.2.3.4 Cache State

Figure 10.9 shows the results of warming the cache with some of the set members before running the application, and using $Comp = 3$. Two observations should be made. First, SETS is able to eliminate I/O stalls when more than one file is in the cache when the benchmark runs. Second, as a result of SETS ability to reorder, the benchmark runs faster with SETS than without, even for 64KB files. However, it should be noted that since the stall time for these sets is small, the overall savings that result from eliminating I/O stalls is modest.

10.2.4 SETS on Large Files

The results discussed in Section 10.2.2 showed that SETS offers no benefit for sets of large files stored on one local disk for the base factor levels. One of the problems is that SETS only prefetches `nprefix` blocks from a file, where `nprefix` = 8 for these experiments. Thus SETS only prefetches a fraction of the blocks: 256KB files have 33 blocks in addition to the inode (32 data blocks and one indirect block), 1MB files have 129. As a result, SETS cannot offer any benefit to a majority of the reads of a large file's data.

This point is made by Figure 10.10, which shows the results of running the benchmark on sets of files stored on multiple disks. Whereas SETS could achieve large benefit by prefetching NFS files in parallel from different servers, here the gain is minimal. Fortunately, the savings are sufficient for SETS to offer a small improvement in performance. Similarly, SETS achieves either modest savings or no savings when $Think > 0$, or the cached is warmed with some of the members, although the results of these tests are not presented here.

The reason for restricting prefetching to a file's prefix is to overcome a limitation of Mach's buffer cache. As discussed in Section 6.4.3, the mapping between a buffer and the file to which it belongs can be expensive in Mach, which uses a hash of the block's physical address to locate the block. The decision to limit prefetching to the prefix of a file reduces the cost of this mapping by avoiding the need for it or by doing it when it is least expensive. Other operating systems, like NetBSD, which use a logical block hash do not have this problem. As a result, a port of SETS to one of these operating systems

S	SETS	CPU milliseconds	Stall milliseconds	Total milliseconds	Savings in Total
1	W/O	13695.76 (57.0)	15743.05 (74.9)	29438.81 (42.1)	
	With	13781.14 (49.4)	16253.53 (88.3)	30034.67 (64.1)	-2.02%
2	W/O	13687.83 (44.1)	15821.73 (47.2)	29509.56 (23.9)	
	With	13794.64 (42.3)	15332.80 (120.6)	29127.44 (112.3)	1.29%
3	W/O	13635.17 (102.4)	15801.39 (52.4)	29436.57 (132.5)	
	With	13739.64 (106.4)	15206.56 (77.5)	28946.19 (163.2)	1.67%

This table shows the results of the benchmark run on 1MB files stored on multiple disks ($S \geq 1$), with the other factors set to their base level. The left column shows the number of disks storing set members. The metrics in the other columns are described in Section 7.2.2. As described in Section 10.2.2, SETS can offer minimal benefit to sets of large files due to limitations in the implementation. In particular, SETS can only prefetch the first 8 blocks of 1MB files, so a majority of the accesses use the same path with or without the use of SETS.

Figure 10.10: Test Results for Sets of 1MB FFS Files Stored on Multiple Disks

will likely perform better for large files in situations where SETS can beat read-ahead, such as when $S > 1$, $Think > 0$, or some of the members are cached.

S	SETS	CPU milliseconds	Stall milliseconds	Total milliseconds	Savings in Total
1	W/O	13679.76 (70.5)	15760.92 (108.2)	29440.68 (69.3)	
	With	14775.47 (101.2)	28345.67 (201.9)	43121.14 (186.5)	-46.47%
2	W/O	13638.62 (69.5)	15849.80 (85.0)	29488.43 (96.6)	
	With	15470.39 (153.5)	8088.52 (142.4)	23558.91 (179.4)	20.11%
3	W/O	13687.15 (69.9)	15865.24 (82.7)	29552.39 (28.8)	
	With	15910.47 (132.8)	2886.68 (30.2)	18797.15 (148.2)	36.39%

The results in this table indicate that removing the `nprefix` limitation would allow SETS to reduce the latency over sets of large files. This table shows the results of the same test as Figure 10.10, but with different SETS prefetching parameters: `nprefix` = 128, `sets_max` = 3.125MB, `pin_max` = 3MB, and `limitOpens` = 3. These parameters were chosen to allow SETS to prefetch whole files from disk. The left column shows the number of disks storing set members (S), the metrics in the other columns are discussed in Section 7.2.2.

Figure 10.11: Tuning SETS Parameters for 1MB Files on Multiple Disks

In order to validate my assertion that the poor performance of SETS on large files resulted from the `nprefix` limit, I repeated this experiment using different values for the SETS prefetching parameters. These new values allow SETS to prefetch an entire 1MB file from disk. Table 10.11 shows the results of this experiment, which uses `nprefix` = 128[2], `sets_max = 3.125MB`, `pin_max = 3MB`, and `limitOpens = 3`.

This test shows two results. First, SETS is able to substantially reduce I/O by prefetching from different disks in parallel. The result is nearly 40% reduction in overall runtime for 3 disks. Second, SETS pays a big penalty for one disk because it must wait until the entire file has been read before yielding it to the application. One of the reasons this pause is so large is the lack of sequential access from SETS. Another reason is that SETS waits until the entire prefix has been cached before yielding the file to the application. When `nprefix` is small, this is a reasonable approach; for large `nprefix` it clearly is not. However, a port of SETS to another operating system which does not need `nprefix` should not suffer this large penalty.

10.3 Conclusion

These experiments demonstrate that SETS do offer improved performance for search on FFS on JABOD, particularly when read-ahead is ineffective. Read-ahead only benefits files larger than 2 blocks (16KB), and cannot take advantage of user think time or concurrency between files, and cannot reorder access to files in a set to make use of cached data before it is evicted.

As stated in the beginning of this chapter, SETS was designed to prefetch from a distributed system, and not from the local disk. As a result, this experiment stresses SETS in different ways than it was intended to support. Still, SETS performs quite well in this environment, providing substantial benefits for small and medium-sized files across a wide range of factors. Since 80 to 90% of files on Unix systems are smaller than 64KB, SETS provides benefit in a majority of cases[3, 64, 78]. Although the benefits for large files are not as great, SETS does offer some reductions in latency over a more restricted range of factors.

The experimental results also show three opportunities for improving the performance of SETS in this domain. First, a simple change to the prefetching engine would allow SETS to ensure that requests for blocks on the same disk are submitted sequentially. This would decrease a penalty SETS incurs, and greatly reduce the cost of I/O when prefetching from the local disk. A second change would be to add more complicated prefetching algorithms

[2]The indirect block is read in as a side effect of prefetching the data blocks, so setting `nprefix` to 128 allows the entire 1MB file to be preread.

which dynamically adapt to suit the current system and application parameters. As an example, SETS could limit the number of opens to no more than S when reading from the local disk. Third, removing the `nprefix` restriction will allow SETS to offer more benefits to search on large files.

Chapter 11

Related Work

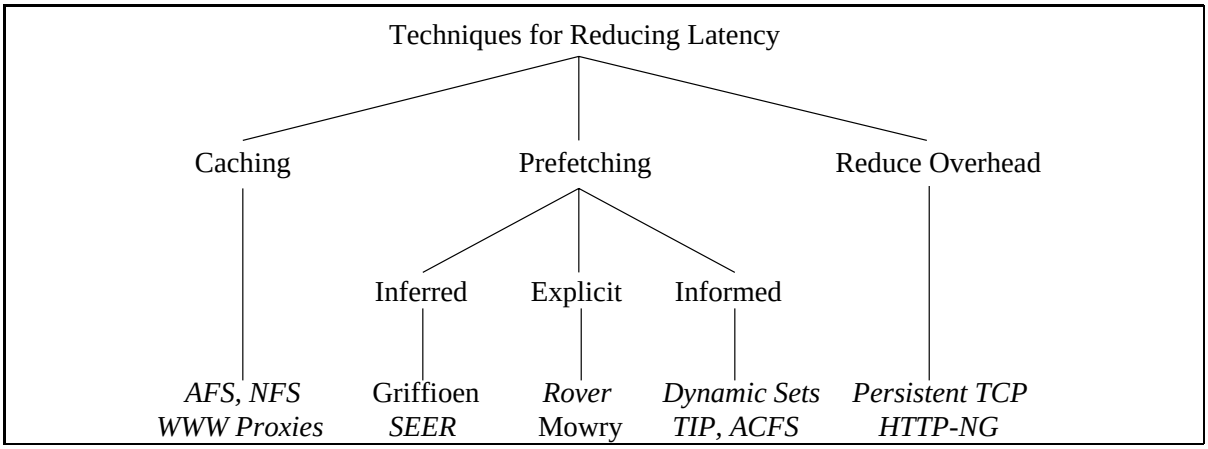
Dynamic sets are an interface extension designed to reduce the aggregate latency of search applications through informed prefetching. There are two basic bodies of work related to dynamic sets: systems attempting to reduce the effect of I/O latency and systems that attempt to improve support for search by changing the file system interface. Relevant work in these areas is discussed in Section 11.1 and Section 11.2. Section 11.3 discusses other work that is related to specific aspects of dynamic sets or SETS.

11.1 Addressing Latency

In most computer systems, the memory system architecture is structured as a hierarchy in which small, fast, and expensive memory is used to hold data from a larger, slower, and cheaper form of memory at the next level. Misses at level i incur a high latency to fetch data from level $i+1$, relative to the cost of a hit at level i. Because there are a number of levels, each with different characteristics and design tradeoffs, several bodies of work focus on how best to overcome latency for some particular layer. For example, both VLSI and distributed file system designers use caches to reduce I/O latency, although the cache structure and policies differ.

Of particular interest to this dissertation are efforts to reduce the latency to read persistent data into local memory from its place on a local or remote disk. Figure 11.1 shows the basic approaches to eliminating latency in file systems, along with example systems for each approach. There are effectively three alternatives: cache to increase the hit rate, prefetch to lower the effect of missing, and reduce the overhead and thus the cost of a miss. Systems that prefetch can be further classified into those that use inferred, explicit, and informed prefetching, based on how the system decides what to prefetch and which level of the system controls I/O. The following sections discuss each of these

five alternatives and how they relate to dynamic sets. Relevant or interesting examples of each class are also presented. In addition, Section 11.1.6 discusses approaches which use a similar philosophy to SETS in their attempts to overcome I/O latency.



This figure shows a hierarchy of techniques to reduce latency. The bottom 2 rows contain examples of a particular class of techniques. The names in the bottom row are either the names of representative systems or the first author of papers describing those systems which were not explicitly named. Dynamic sets are an example of informed prefetching.

Figure 11.1: Hierarchy of Techniques to Reduce Latency

11.1.1 Caching

Caching has been widely used to overcome read latency in distributed systems. Systems such as AFS[34], NFS[77], and Sprite[58] rely on caches to provide good performance, and most WWW browsers employ some form of caching as well. As described in Section 2.1.2.1, caches only work well when an application exhibits good temporal locality. Search applications tend to exhibit poor locality, and so caching does not perform well for searches. In addition, the data accessed by a search can flush the cache, and thus harm the performance of other concurrently running applications. This effect is shown by one study of a WWW proxy cache shared by several thousand users. The cache showed a hit rate of roughly 33%, even though all accesses passed through the cache and the cache's size was unlimited (e.g. no object was evicted to make space for another object)[27]. The low hit rate was due to first references of objects as well as invalidations of cache entries for objects that were no longer valid. Since this study is in many ways optimal, it is a

demonstration of the limited benefit caches offer to applications on a GDIS.

Another use of caching proxies, called geographic push-caching[31], offloads servers by *pushing* objects to a proxy that is geographically closer to the source of the requests. For instance, if a large number of CMU students access WWW pages from Harvard, the Harvard server may choose to push these pages to a caching proxy at CMU. This would result in lower latencies for CMU students and a lower load on Harvard's server. However, this approach assumes that one can dynamically find proxies that are near a cluster of clients on the network and are willing to serve these objects. It also assumes that the popularity of these objects will continue long enough to justify pushing them out to a proxy, and that issues involving access control and ownership of intellectual property will be solved shortly.

11.1.2 Inferred Prefetching

Many systems attempt to reduce latency by prefetching data based on guesses of future accesses [45, 19, 93, 44, 43]. In some cases, these guesses can be reasonably accurate and result in performance improvements. However, the penalty of inaccurate predictions is overutilization of the I/O channel. Since the I/O channel is likely to be the bottleneck of systems that would benefit from prefetching, these faulty predictions can lead to significant performance degradation.

A common form of inferred prefetching is the one-block read-ahead mechanism employed by the Unix local file system[2, 87]. The system detects that a file is being read sequentially and automatically queues a read for block $i + 1$ when block i and $i - 1$ have been read. Results from the NFS and FFS experiments discussed in Chapters 9 and 10 indicate read-ahead is effective at reducing I/O when a file is read sequentially. However, the benefit of this approach is limited to applications that read files sequentially, and read files that are large enough to allow the system to infer sequentiality. For instance, most Unix files are too small (less than 16KB) to benefit from read-ahead[3, 64, 78].

The benefits of more aggressive inferred prefetching are less clear. Korner proposed the use of an expert system to analyze I/O traces and build a model which can predict the I/O needs of an application. The system would then use these predictions to manage the cache and prefetch information. Although the paper describes a simulation study which shows impressive benefits from this approach, it is unclear whether these benefits would carry to a real system. In addition, it would require repeatedly running an application "to fully exercise its file behavior" (p 222) before any benefits could be realized. Although this may be useful in determining how a search application will process a file, it is unlikely to determine accurately what files it will access.

Griffioen and Appleton suggest the use of a probability graph to make prefetch and cache

decisions[29, 28]. The graph, built over time, shows the number of times that some file (A) is accessed after some other file (B). One can infer a semantic dependency between the two files if the probability that B is accessed when A is accessed (i.e. $P(B|A)$) is high. Similarly, Curewitz, Krishnan, and Vitter study the use of a compression algorithm to drive prefetching[19]. Intuitively, the act of predicting future references in a reference stream is similar to a compression algorithm's predictions of bit patterns in a data stream. The algorithm eliminates I/Os in the reference stream by prefetching or smarter eviction policies just as it eliminates bits in the data stream. The drawback of both of these techniques is that they require reference patterns that are regular and have been exhibited before. Search applications do not exhibit either property to a great degree, and any small gains are likely to be overwhelmed by the increased load from inaccurate predictions.

Mogul and Padmanabhan are currently exploring Griffioen and Appleton's technique for reducing the latency of WWW access[65]. In their approach, a WWW server builds the probability graph, and informs clients of likely objects to prefetch when responding to a fetch request. The client can then chose to prefetch the objects speculatively. Simulations based on traces of their WWW server show a small benefit. The technique can reduce latency, but does increase the bandwidth consumption due to inaccurate predictions. They have not deployed this technique on the WWW at this point, and have no experience on this approach in practice.

11.1.3 Explicit Prefetching

As an alternative to speculative prefetching, some systems expose asynchronous I/O operations directly to applications, and applications manage prefetching themselves by explicitly invoking these operations. Since applications know their own data needs, they can often initiate fetches early enough that the operation to fetch some piece of data completes before the data is needed. The two drawbacks of this approach are that it places the burden of efficient I/O on the application programmer and that applications lack information such as the state of the local cache necessary to utilize resources efficiently. In addition, applications that manage I/O themselves are highly sensitive to changes in CPU or I/O speed, and are thus difficult to port or maintain.

One recent system that advocates this approach is the Rover Toolkit, which offers *queued remote procedure calls* as its communications paradigm[37]. Since QRPC do not block the calling thread, the application programmer must manage asynchrony directly, such as polling Rover to find out when an operation has completed. In addition, the application must preserve the context of the call in order to respond appropriately if the operation ever fails. Although they do not study this effect, conversations with the authors indicate that programming Rover is non-trivial even for expert programmers.

A recent paper by Mowry, Demke, and Krieger describes a similar approach which uses compiler inserted hints to pre-page in a virtual memory system[55]. Rather than having the application writer generate I/O directives, their compiler generates prefetch requests by analyzing program loops to determine near-future data accesses in virtual memory. Similar data-flow analysis allows the compiler to insert hints to release pages as well.[1] One drawback of their approach is that the compiler cannot know the state of physical memory, and must generate many more I/O requests than is necessary. They overcome this problem by exposing the state of the page tables to the user-level, and having a library filter out unnecessary page-faults at run-time. One drawback of their approach is that it violates software engineering principles like information hiding by exposing page tables to the application. Further, the approach is limited to applications with regular loop structures and available source code. One interesting extension of their idea would be to couple compiler-generated hints with an informed prefetching mechanism such as TIP or ACFS.

11.1.4 Informed Prefetching

Informed prefetching avoids the problems with explicit and inferred prefetching by having the application inform the system of future accesses. Prediction is not needed since the application tells the system what to prefetch. In addition, the application does not have to control prefetching itself, and can leave decisions of what and when to prefetch to the system. Dynamic sets is a form of informed prefetching, where the knowledge of remote access is derived from a set's membership.

Informed prefetching was pioneered by Patterson, Gibson, and Satyanarayanan in the TIP system[67, 66, 68]. TIP provides a hint-based interface which allows applications to disclose future accesses or access patterns to the file system. The system uses a dynamic estimate of resource value to control its prefetching decisions. Results of TIP implemented as part of the OSF/1 operating system show speedups on a local disk array similar to those seen by SETS on NFS, but for a different mix of applications. For reasons stated in Chapter 6, the approach taken by TIP is not suited to the target domain of SETS, since it relies on the ability to accurately estimate the latency of a remote operation. SETS could be layered on TIP, for instance as a new estimator, allowing SETS to prefetch remote data while TIP manages the local buffer cache.

A similar approach is taken by Cao et al. in their studies of application controlled file caching and prefetching[12, 13, 14]. In addition to use of hints for disclosure, their system

[1] Their prefetch requests appear to be explicit I/O requests, but the release operations seem similar to informed prefetching hints. Because the composition of a system is open to interpretation, compiler-generated hints to a run-time system can either be viewed as explicit or informed.

(ACFS) allowed applications to request specific cache policies, such as a *most recently used* eviction policy. A comparison between the behavior of ACFS and TIP based on trace-driven simulation shows the two systems offer comparable performance across a range of applications[40], although studies of the effects of either approach in a DFS or GDIS have not been published.

11.1.5 Reducing Network Overhead

Another approach to reducing latency is to reduce the cost of fetching remote data. Studies of HTTP performance have detected opportunities to improve performance by modifying the protocols[54, 88]. Padmanabhan and Mogul examine two such strategies, long-lived connections and pipelining of requests[54]. The first technique is based on the observation that TCP connections are expensive to setup. Reusing the TCP connection for multiple fetches to the same server can amortize the cost of setting up the connection. The trade off is that maintaining connections consumes state on the server, and may limit their scalability. The second technique batches the requests for a document and some or all of its inlined-images into a single request. Next generation HTTP protocols (HTTP-NG) propose similar extensions[89], although it is unclear when these protocols will be in widespread use.

Although these techniques seem promising, they require modifying protocols and servers. In a system like the WWW which has a large embedded base, upgrading clients and servers may be infeasible. In addition, there may be only limited opportunity to benefit from these techniques, since they require a form of spatial locality – multiple requests to the same server over a short period of time. As discussed in Section 3.1.1.4, dynamic sets only require modifications to the client and thus do not have this drawback. However, SETS could be extended to leverage the benefits of these approaches if they became widely available. For instance, SETS could schedule requests to increase spatial locality in a manner similar to the way it reorders to exploit cache state to reduce latency.

11.1.6 Deriving Hints from Operation Semantics

The SETS prefetching engine achieves good performance by leveraging off the semantics of search applications, in particular the fact that they read whole-files sequentially. Several other systems use a similar philosophy, such as a number of studies of buffer pool management in database systems[16, 74, 18]. In particular, Chou and DeWitt analyze a number of relational database access methods to determine the buffer needs of each method. When a query enters the system, it first determines which access methods will

be used and then allocates a sufficient number of buffers to that query. By using operation semantics to carefully allocate buffers, the system can support potentially more concurrent queries and thus increase the throughput of the database.

Similarly, Grimshaw and Loyot propose extending the file system interface to include objects (types with methods) in a system called ELFS (Extensible File System)[30]. A file's methods can be written to exploit domain-specific knowledge, such as reading a matrix by rows or scanning a database's index. Where appropriate, this knowledge can be used to prefetch data or to change the file's layout on disk. The chief drawbacks of this approach are that application programmers must learn to use a new interface, and that file access is directly tied to file type. Applications which access a file contrary to the file type's implementation may see poor performance as a result.

11.2 Improving Support for Search

Another way to improve the performance of search applications is to reduce the amount of information accessed during a search by increasing the precision of the set of candidate objects. The information retrieval community (see the text by Salton and McGill[75]) has made great strides in automatic indexing technology and improving query languages. Dynamic sets benefit from their advances: with higher quality sets, the set's members are more likely to be accessed and fewer prefetches will be wasted.

Recently, a number of researchers have explored ways to add better support for search to file systems. Essentially, they propose either extending or replacing the standard file system interface and name space structure to support associative (attribute-based) naming. These proposals must provide ways to automatically extract a file's attributes and index them, and must also provide a query language for finding files. This language can either be integrated with an existing system, be part of a new system interface, or only be accessible to the user through new applications.

Dynamic sets do not provide a mechanism for extracting attributes, but do extend the standard file system interface with a flexible query language, as described in Chapters 3 and 4. Extending SETS to use a new type of index, such as WAIS[39] or Essence[33], requires extending an existing warden or supplying a new one. Adding a warden is relatively straightforward and should take no more than a man-month. Fortunately, most of these services provide a WWW interface, and are thus accessible today via SETS's HTTP warden.

11.2.1 Extracting Attributes from Files

One approach provides associative naming through a user interface. Schwartz et al. survey early examples of this approach, which they call "Internet Resource Discovery"[84]. These systems are layered on top of an existing system and do not require modifications to the underlying DFS or GDIS or its interface. Examples of this approach are GLIMPSE[50], Archie[21], WAIS[39], Essence[33], and WWW search engines such as Altavista[2] or the WebCrawler[3][69]. These systems focus on a different problem than dynamic sets, that of identifying sets of candidate objects. As a result, this work benefits from their successes because searchers can create more accurate sets through queries to these search engines.

Essence, a resource discovery system from Hardy and Schwartz[33], allows users to register *summarizers* which can parse files of a particular type. A file's type is inferred from its name (or possibly its contents) using user-supplied rules. Consistency between the index and the file contents is bounded; the index is regenerated by rerunning the Essence program.

The WebCompass search engine from Quarterdeck Corporation[95] performs two tasks: it runs queries on a user-supplied list of WWW search engines and collates the results, and builds a local database of interesting objects. This local database can be automatically updated by an *agent* which re-executes a user's queries periodically, for instance to get the latest news articles on a particular subject.

WebCompass is in many ways similar to dynamic sets, but it differs in two key ways. First, WebCompass creates persistent copies of query result sets, but does not ensure that the copies remain consistent. The drawback of this approach is that the local copies consume valuable disk resources and may not be useful to future searches (as discussed in Section 3.1.3.1). For example, almost all newspaper articles have little value after the day on which they are published. The effort to collect this data and the cost of storing it may be completely wasted. Further, WebCompass only provides rudimentary mechanisms for pruning unneeded objects from its database. A review of WebCompass in Byte Magazine cites this problem of excessive consumption of disk space as one of WebCompass's drawbacks[4].

A second difference is that WebCompass does not prefetch members of a result set, therefore it offers no performance advantages to interactive search. The key observation that WebCompass's designers failed to make is that a query's result set is a hint of future access. Dynamic sets shows that this hint can be used to fetch objects just-in-time, to get the performance improvements without consuming a significant amount of local resources.

[2] http://altavista.digital.com
[3] http://webcrawler.com

11.2.2 Extending File Systems to Support Search

A second class of solutions adds some form of attribute-based naming to an existing file system. These systems extend the file system's interface to support a query language in addition to providing automatic attribute-extraction and indexing. They are similar to dynamic sets in that they add a query language to the file system name space, and may add operations to the file system interface. However, none of these systems attempt to use search information to drive a prefetching engine. Further, they require modifications to servers.

The most notable example of this approach is the Semantic File System (SFS) from Gifford et al.[26]. SFS is a modified NFS server which automatically extracts attributes from files as they are stored, using file-type specific *transducers* or filters. SFS queries are pathnames consisting of successive attribute-value pairs. Each pair adds a constraint (the specified attribute should have the specified value) which the identified files should satisfy. Pathname lookup results in a *virtual directory* which contains objects that satisfy all of the constraints. This mechanism is similar to that employed by SETS, but is unwieldy, restricted (it only supports conjunctive queries), and not extensible. However, standard Unix applications can access SFS files with no modifications. In addition, the SFS implementation does not modify the NFS client, and so cannot benefit from the approach advocated in this dissertation. Dynamic sets, however, could easily be extended to access files in a semantic file system with the addition of an SFS warden, and may even be able to access SFS files now via the NFS warden.

11.2.3 Replacing the File System

Another alternative is to abandon the file system interface in favor of a more powerful paradigm. One example of this approach is the WWW, which provides a hypertext name space and defines its own operations through the HTTP protocol[6]. A survey of Internet resource discovery systems[84] describes this and other Internet-based systems such as Gopher[52] or Prospero[60]. Dynamic sets eschews this approach, and instead makes a small extension to the file system interface without totally abandoning it. As a result, one can modify existing applications to use SETS with little effort.

One interesting idea is to replace the traditional file system with an object-oriented repository or database. The benefits of such a move are that all objects have a type, and the system need not conform to file system (e.g. POSIX) semantics. Since an object's type is well known to the system, the system can leverage an object's semantics to prefetch, much in the way SETS does for dynamic sets. The penalty is that applications would have to be rewritten for this system, and may not be portable to new platforms.

An example of this approach is the ELFS file system discussed earlier[30]. Another example is the Rufus system by Shoens et al.[86]. Rufus "provides searching, organizing, and browsing for the semi-structured information commonly stored in computer systems"[86](p 97). Rufus consists of a class hierarchy, an object-oriented database which stores file contents and attributes, and an automatic indexing mechanism based on type-specific *classifiers*. Users *import* objects into the Rufus system to add them to the database, supplying the object's class which identifies which classifier to use to extract the object's attributes. New classes (and classifiers) can be defined by the user to support new types of files. The main drawback of Rufus is that applications must be rewritten to invoke Rufus methods. Another drawback is that Rufus does not maintain consistency between the original and imported copies of an object, although one can re-import objects to update the database with little overhead (such as periodically running a Rufus daemon).

11.3 Mechanisms

In addition to the preceding systems, there is also work which is related to specific aspects of dynamic sets. The following subsections discuss work that is relevant to three aspects of dynamic sets: iterators, digests, and user-level file systems.

11.3.1 Sets and Iterators

Use of sets and iterators are not new to dynamic sets. One example of their use in programming languages is cursors in SQL. The unique contribution of this work is the exploitation of the semantics of dynamic sets to reduce the aggregate latency to access a group of objects.

One could use an approach similar to dynamic sets by using membership in an SQL cursor to prefetch tuples in a database. However the payoff in a database is much smaller than the payoff in a DFS or GDIS, as the I/O costs of executing a query (select statement) far outweigh the costs of fetching tuples. In fact, in some cases the database must fetch the tuple's data in order to perform the join. With DFS or GDIS, the time to identify the candidate objects is often equivalent to the cost of fetching just one of the objects, providing an opportunity for an approach like dynamic sets.

11.3.2 Digests

Chapter 3 introduces a *summary* type which provides a way of obtaining the attributes of a set's members to allow searches to control their search without having to fetch the

members. Searchers can obtain these summaries via the `setDigest()` iterator, and then use these summaries to restrict the focus of their search to include only promising objects. Since summaries may be obtained much more cheaply (either in terms of money, time, or bandwidth) than the objects' data, use of digests may result in faster and more efficient searches. One should note that this dissertation does not explore the use of digests in detail, and SETS only offers one type of summary: the object's name. However, it would be easy to extend SETS to provide attributes which many WWW search engines include with the results.

Fox et al. describe an approach that creates *distillations* (digests) dynamically[25]. The basic idea is similar to that of digests in SETS: reduce the bandwidth requirements by type-specific lossy compression. Because distillation is successful at reducing the amount of transmitted data, the time to perform it on demand is more than covered by savings in transmission time. To allow a user to see the full object (or some portion of the original representation) after viewing the distillation, their system also provides type-specific *refinement*. Refining a distillation consists of fetching and displaying some or all of the original object. For instance, one might download the distillation of a dissertation into an abstract and keywords, and then refine the distillation to include the introduction chapter. One interesting question they do not answer is whether distillation and refinement should be type-specific or search-specific. For instance, a searcher may prefer a particular form of lossy video compression for one search and another form for some other search.

Another approach is taken by the Synopsis File System (SynFS), which associates a summary (synopsis) of a file's contents with every object in a file system[36, 11]. Like the Semantic File System[26], synopses are created when a file is stored, and maintained by a separate server. The chief use of a synopsis is in file location, although the contents of a synopsis could be used interactively along the lines of `setDigest()`. Synopses differ from distillation because they are pre-computed and persistent. The chief difference between the SFS and SynFS is that the latter stores attributes independently from the index of these attributes, and thus allows searches to make use of them outside of queries.

11.3.3 User-level File Systems

Although not central to the dissertation, SETS does make use of user-level file system client subsystems through Coda's Minicache. The chief benefits of this approach are the ease of prototyping wardens and the ability to dynamically extend SETS to access objects in other systems. Recently, a number of other techniques have allowed file systems to be extended in a similar manner. For instance, Linux provides dynamically loadable modules with which one can add a new file system on the fly. Although this increases

dynamic extensibility, it does not simplify the job of prototyping since new modules run as part of the kernel.

Watchdogs[8] and the Intensional File System[20] both provide a mechanism to allow users to modify the semantics of file system operations. Watchdogs allow users to attach a program to files or directories. When the object is accessed, the operation is passed by the kernel to the watchdog process. The watchdog can then either satisfy the operation itself to provide modified semantics, or let the system handle the request to provide the default semantics. Although not directly relevant to SETS, similar techniques could be used in the execution of queries.

The Intensional File System modifies the standard Unix name syntax to allow programs to be executed as part of name resolution. The result of executing the program can be used as a name, to be further resolved by the system in order to satisfy the original call. Thus IFS is also similar to SETS' executable membership specifications. The Intensional File System is implemented in libraries at the user level, and provides neither a secure environment in which to execute the commands nor a tight integration with the system.

11.4 Conclusion

In summary, the problem of I/O latency and the lack of support for search applications are critical to the future of distributed systems. Dynamic sets provides a unique solution to the problem of I/O latency for search applications, and increases system support for search by supporting iteration over sets and query execution. Dynamic sets can substantially reduce latency for search without requiring modifications to protocols or servers, without violating software engineering principles, and without relying on caching.

Chapter 12

Conclusion

A key goal of distributed systems is to provide prompt access to shared information repositories. The high latency of remote access is a serious impediment to this goal. Latency is especially problematic for search, a critical application, because it tends to access many objects and may experience long latencies for each object accessed. Existing techniques like caching, inferential prefetching, or explicit prefetching are either ineffective at reducing latency for search, or greatly increase the complexity of the programming model.

This dissertation shows that dynamic sets can substantially reduce the aggregate latency of search applications. Dynamic sets offer a powerful interface for search applications without sacrificing the simplicity of the file system interface. An application's use of dynamic sets discloses hints of future access to the system. The system can use these hints to drive an informed prefetching engine, thereby reducing the aggregate I/O latency to process set members. Dynamic sets offer the benefits of prefetching: overlapping concurrent requests to exploit I/O parallelism, overlapping computation and I/O, and more efficient resource utilization. In addition, dynamic sets allow the system to reorder access to set members to fetch them more efficiently.

Dynamic sets are unique in that they offer the performance improvements of asynchrony without significantly increasing the complexity of the system interface. Instead, dynamic sets offer clean and well-defined semantics, and increase the power of existing interfaces by supporting iterators, associative naming, and direct management of sets of objects. In addition, dynamic sets adhere to well-established software engineering principles by preserving the strong boundary between applications and the system.

12.1 Contributions of the Dissertation

The original insight of this dissertation is that current interfaces restrict a system's ability to reduce latency, and that a small and carefully designed set of interface extensions can overcome this limitation without sacrificing the integrity of the system interface. Dynamic sets can substantially reduce or eliminate I/O latency for a variety of search applications on a broad range of systems. Further, they achieve this without pushing the onus of reducing latency to the application programmer, which would both result in inefficiency and unnecessarily complicate the programming model. The demonstration of the dynamic sets abstraction and its benefits are thus the primary benefit of this dissertation.

Specific contributions of this dissertation are listed below, categorized in four major areas:

- Design

  The design of the dynamic sets abstraction provides a powerful yet natural and easy-to-use extension to the file system interface. Dynamic sets expose asynchrony to the application programmer in a well-defined and controlled fashion. In addition, dynamic sets are designed to minimize restrictions on the implementation. In particular, dynamic sets can be implemented without modifying underlying system protocols or servers, and still achieve substantial reductions in latency.

- Implementation

  This dissertation described SETS, an implementation of dynamic sets as an extension to the file system interface of Mach 2.6. SETS experimentally validates the design by showing that dynamic sets can be implemented in a clean and efficient manner, achieving all of the benefits of the design. Further, SETS demonstrates that a single tunable prefetching engine can provide performance improvements on a range of platforms, from search on local disk files to search on the WWW.

  The implementation serves as a platform for further research in areas such as dynamic system adaptation to changing resource availability, mobile search on global distributed systems, and function shipping to reduce network bandwidth consumption. In addition, SETS may prove to be useful as a platform on which to explore the benefits of dynamic sets in domains such as data mining.

- Qualitative Evaluation

  Another contribution of the dissertation is a qualitative evaluation of the application and system interfaces of dynamic sets through the implementation of several applications and wardens. The applications include a number of Unix search utilities such as `grep`, `ls` and `more`, and the Mosaic WWW browser, and represent

both interactive and non-interactive search tools. The wardens allow SETS to access data in a variety of distributed systems, including the WWW, SQL databases, NFS, and the Coda file system.

This experience, although subjective, does indicate that the dynamic sets programming model is indeed suitable to support search applications and is simple to program. Changes to the Unix applications involve a handful of lines of source code; changes to Mosaic are more extensive because they also involved extensions to the GUI. In addition, the design of the warden interface allows support for new distributed systems to be easily added to SETS. The NFS and Coda wardens which were based on existing client subsystems were easy to implement, requiring less than one percent additional code and a few days of effort. The HTTP and SQL wardens required more work, but were implemented from scratch. Based on the experience of adding these four wardens, I conjecture that support for other GDIS or DFS could be added to SETS with a minimal investment of time.

- Quantitative Evaluation

A fourth contribution of this dissertation is the quantitative evaluation of the performance benefits of dynamic sets. This evaluation consists of three experiments, each of which examines the benefits of SETS to search in a different domain: GDIS, DFS, and local file systems. Together, the three experiments show that the performance benefits of using SETS are robust across a range of very different systems, including one for which SETS was not intended. An additional contribution of the evaluation is a performance model which characterizes the performance benefit of dynamic sets.

The GDIS experiment examined the effect of SETS on interactive search on the WWW through trace replay, and found that SETS provided an order of magnitude reduction in I/O latency seen by the application. The DFS experiment used a synthetic benchmark to examine the effect of SETS on search on NFS files. This experiment shows that SETS can decrease runtime by up to half over a range of factors, as predicted by the model; in some cases SETS can eliminate I/O stalls altogether. The third experiment used the same synthetic benchmark to determine the performance benefit of SETS on search on the Unix Fast File System. Although not designed for this domain, SETS was still able to provide significant reductions in latency for small and medium-sized files under a wide range of conditions, and for large files under a smaller range.

12.2 Future Work

This dissertation points out a number of minor modifications which would improve the implementation of dynamic sets. For instance, the NFS and FFS tests found that SETS increased the cost of I/O in certain circumstances by interleaving accesses to different files through prefetching. In addition, the FFS test performed poorly on large files because of a limitation on the prefetching engine due to an aspect of Mach's buffer cache implementation. Although SETS performs well enough as is, fixing these minor problems would improve its usefulness in these situations.

Beyond these minor enhancements, the contributions of this dissertation could be strengthened in the areas of evaluation, performance, and dynamic adaptability. Each of these areas is discussed in the following subsections. Some of these points will be addressed by other members of the Odyssey project; others are left as suggestions.

12.2.1 Further Qualitative Analysis

The experiments and implementation which comprise the evaluation described by this dissertation conclusively demonstrate the tremendous performance improvements that can result from using sets. These results could be strengthened by a broader validation which examines the utility of dynamic sets, for instance through deployment to an external community of users. Such an exercise would answer a number of open questions. Do opportunities exist to express applications' file needs as dynamic sets? Can users employ dynamic sets to perform search? Can users precisely specify membership, or does set membership necessarily include false positives? If so, does the presence of these false set members influence the performance benefits of sets? Most importantly, does the use of sets improve the overall runtime of search in practice?

The exploration of two interesting avenues might answer these questions. One would be to reimplement the SETS functionality as a Netscape plug-in or Java *applet*. Doing so would allow any user of the WWW to download dynamic sets and gain most of the benefits of SETS. In addition, it would become possible to study some of these users to determine how they use dynamic sets. Such an implementation is necessarily less general and efficient than SETS, since it will only run within Netscape or other Java-enhanced browsers at the user level. However, it may still possible to achieve significant performance improvements with such an approach.

A second avenue to explore would be to re-design the user interface to SETS applications, train a user population on the use of dynamic sets, and then perform a study of how dynamic sets influence the users' search behavior. Possible extensions to the current interface include specifying set membership by highlighting portions of a WWW page

with the mouse, color coding the names in a digest to indicate which have been fetched, and allowing users to indicate use of SETS for forms and inlined-images on a page-by-page basis.

A different extension of the evaluation would be to explore the benefits of sets in domains other than search. Examples of such domains are listed in Section 3.3. SETS could be used as a platform for such exploration, perhaps after porting it to a more suitable operating system such as Linux[5] or NetBSD[59].

12.2.2 Enhancing the Performance of SETS

An explicit goal in the design of dynamic sets was to avoid modifying protocols or servers to support sets. If, however, one were to remove this restriction there are several opportunities to improve the performance of SETS, or to reduce bandwidth consumption. The set could be exposed to lower levels of the system, such as to proxies or servers, to allow them to obtain the benefits of SETS to reduce latency.

One such opportunity improves search performance on a low bandwidth link and results from exposing sets to a proxy. As discussed in Section 8.2.4, a low bandwidth link between a client and the Internet introduces a problem: overcoming Internet latencies demands concurrent access, but doing so may overwhelm the weak link. If one added a proxy on the remote side of the weak link, the proxy could expand the set and fetch the members to its local disk utilizing its high bandwidth connection to the servers to fetch objects concurrently. The client could then optimize use of the low bandwidth link when fetching from the proxy.

A second extension mentioned in Section 3.2.3.4 involves partially exposing a set's membership to the server to enable batching. Batching allows the client to fetch multiple objects with one request, and reduces the number of packet exchanges. In addition, this technique could make use of compression to reduce the amount of bandwidth transmitted, or reuse the connection in protocols which currently establish a new connection for every request (such as the WWW's HTTP protocol)[54].

A third extension would completely expose sets to servers to enable function shipping. Clients ship a filter to servers; this small piece of code can examine a server's data locally. Objects that satisfy the filter are shipped back to the client for closer examination. If the filter is successful, fewer objects need to be shipped and the ones that are shipped are more likely to satisfy the search. In addition to lowering bandwidth consumption, these filters may increase the productivity of users by showing them higher quality matches to their searches.

12.2.3 Dynamic Adaptation to Changing Resources

In performing the experiments and in use of the system, it became clear to me that a prefetching engine must be tunable in order to perform well across the range of systems pertinent to this dissertation. As described in Chapter 6 (in Figure 6.5), the SETS' prefetching engine has five parameters, plus one which results from use of the Mach buffer cache. Although one may manually set these parameters, SETS does not provide a mechanism that automatically tunes these parameters to adapt the behavior of the prefetching engine to suit current resource availability.

The basic idea of dynamically tuning these parameters is that optimal performance comes from pushing the system as aggressively as possible without driving the system into overload. The appropriate level of aggressiveness depends on the current situation. For instance, in the NFS experiments on the Ethernet, SETS benefited when `limitOpens` $> S$, since it could overlap use of the network with work at the server. However, this same setting produced poor performance on Wavelan or SLIP by overloading the network. Reducing the value of `limitOpens` improved the performance for these lower bandwidth networks. In general, the ability to dynamically adapt to changing resource availability would result in more efficient resource utilization. Dynamic adaptability would be especially valuable to clients which see a large range of performance, such as mobile clients which are sometimes directly connected to an Ethernet, sometimes to a wireless network, and sometimes connected over SLIP.

The key unanswered question is whether or not there exist reasonable algorithms which can infer the right settings for SETS parameters using information available to the client. One such algorithm measures the response time from NFS servers and the average I/O to the local disk, and prefetches NFS files to the local disk if the server is too slow. The dangers of these algorithms is that they may lead to thrashing and device overutilization by either reacting too quickly to transients or through positive feedback loops.

The idea of dynamically adapting to changing resources is not new. Dynamic window sizing network protocols such as the one used in TCP[35] or Coda[56] tune the behavior of a network connection to increase the average bandwidth over a long transfer of data. A dynamically adapting prefetching mechanism is slightly more complicated, however, as it has more degrees of freedom.

12.3 Closing Remarks

Reducing I/O latency is critical to the continuing success of distributed systems. Long latencies lower the productivity of users. Variance in the length of the delay increases their annoyance. Unfortunately, technology and usage trends indicate that latency will

continue to be a problem, and may worsen as latencies grow longer relative to CPU speeds. With the large growth in use of global distributed systems like the WWW, load from usage is likely to overwhelm the increase in performance from technology for the foreseeable future. Existing techniques to overcome latency either depend on temporal locality or shift the burden of providing efficient data access to the programmer.

Dynamic sets address the problem of I/O latency by exposing the benefits of asynchrony to applications through a controlled and well-defined interface. By using sets, applications disclose hints of future data needs to the system, and the system can use these hints to drive a prefetching engine to reduce the aggregate latency of accessing a set of objects. Use of a set is also an indication that reordering access to the members is acceptable to the application, which frees the system to fetch the members in optimal order.

This dissertation demonstrates that dynamic sets can offer substantial reductions in latency to applications without significantly increasing the complexity of the programming interface. The experiments presented herein have shown that dynamic sets can reduce latency by up to an order of magnitude, can eliminate I/O stalls in some cases, and that these benefits are robust across a diverse set of domains. Further, adding the dynamic set abstraction to a system enhances its functionality by supporting iterators, associative naming, and direct management of sets of objects. With these advantages, I believe dynamic sets will become a standard feature of distributed systems on which search is a common operation.

Appendix A

Manual Pages for SETS API

NAME

setClose – deallocate a set descriptor

SYNOPSIS

#include <sys/sets.h>

setClose(int d)

DESCRIPTION

The *setClose* call deletes a descriptor from the per-process set reference table. If this is the last reference to the set, then it will be deactivated. For example, on the last close of a set the current *iterator* associated with the file is lost; and the set's *descriptor* is no longer valid. If this descriptor is the last reference to a set, all holds on objects in the set will be released, and the resources (memory) consumed by the set will be freed.

A close of all of a process's descriptors is automatic on *exit*, but since there is a limit on the number of active descriptors per process, *close* is necessary for programs that deal with many descriptors.

When a process forks (see *fork*(2)), all descriptors for the new child process reference the same objects as they did in the parent before the fork. If a new process is then to be run using *execve*(2), the process would normally inherit these descriptors. Most of the descriptors can be rearranged with *dup2*(2) or deleted with *setClose* before the *execve* is attempted, but if some of these descriptors will still be needed if the execve fails, it is necessary to arrange for them to be closed if the execve succeeds. For this reason, the call ''fcntl(d, F_SETFD, 1)'' is provided, which arranges that a descriptor will be closed after a successful execve; the call ''fcntl(d, F_SETFD, 0)'' restores the default, which is to not close the descriptor. Similarly, a set that is opened with the ''SETS_NODUP_ON_FORK'' will not be inherited by the child process (see *setOpen(2)).*

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and the global integer variable *errno* is set to indicate the error.

ERRORS

SetClose will fail if:

[EBADF] *d* is not an active set descriptor.

[EINVAL] The process has no open sets.

[EOPNOTSUPP] Sets are not supported on this machine.

SEE ALSO

accept(2), flock(2), setOpen(2), execve(2), fcntl(2), close(2)

NAME

setDigest – get a list of member names

SYNOPSIS

#include <sys/set.h>

int setDigest(int set, struct digest *buf, int count);

DESCRIPTION

setDigest returns a list of the names of set members in the buffer *buf*. *count* should hold the size of the buffer in bytes. *setDigest* will return as many names as it can fit into the buffer. If it cannot return the names of all unyielded members, the last entry will hold the string ''...''. If this is the case, the next call to *setDigest* will return these unyielded names; successive calls are guaranteed to avoid returning the same name twice (with the obvious exception of the special name ''...'').

The structure *struct digest* contains two fields, *next* and *name*. *next* is a pointer to the next digest in the buffer, *name* is a null-terminated string containing the next name.

RETURN VALUE

The value returned by *setDigest* is either a whole number indicating the amount of data that was returned or -1 to indicate an error. If -1 is returned, the global integer variable *errno* is set to indicate the error.

ERRORS

The named file is opened unless one or more of the following are true:

[EBADF] *d* is not an active set descriptor.

[EINVAL] The process has no open sets.

[EOPNOTSUPP] Sets are not supported on this machine.

SEE ALSO

setOpen(2), setIterate(2)

NAME

setIntersect – create a new set to be the intersection of 2 existing sets.

SYNOPSIS

#include <sys/set.h>

int setIntersect(int set1, int set2, int flags);

DESCRIPTION

setIntersect creates a new set which is the intersection of two existing sets. *set1* and *set2* are handles for sets created earlier by *setOpen(2), setUnion(2), setIntersect(2),* or *setRestrict(2).* The *flags* argument selects the desired behavior for this set, see *setOpen(2)* for more details.

setIntersect returns a set descriptor which can be used in future operations on this set. Since sets are dynamic and transitory in nature, the set will only exist until *setClose(2)* is called on this set.

RETURN VALUE

The value returned by *setIntersect* can be used as an argument to other set operations, such as *setIterate(2).* However, other Unix operations that expect a file descriptor will fail if given a set descriptor. The new descriptor will remain open across *execve* system calls; see *close*(2). If the operation fails, -1 will returned and *errno* will hold a code indicating the error that occurred.

The system imposes a limit on the number of set descriptors open simultaneously by one process. *Getdtablesize*(2) returns the current system limit.

ERRORS

The named file is opened unless one or more of the following are true:

[EBADF] The set handles *set1* or *set2* do not refer to valid sets.

[EMFILE] The system limit for open set descriptors per process has already been reached.

[ENFILE] The system set table is full.

[EINVAL] The process has no open sets.

[EOPNOTSUPP] Sets are not supported on this machine.

SEE ALSO

close(2), dup(2), getdtablesize(2), setClose(2), setOpen(2), setUnion(2), setRestrict(2)

NAME

setIterate – open the next object in a sequence of set members

SYNOPSIS

#include <sys/set.h>

setIterate(char *set, int flags, char *buf, int bufsize, int *error);

DESCRIPTION

setIterate returns an open file descriptor for the next unyielded member of a set. The order of objects returned is not defined. It is guaranteed that *setIterate* will never return the same object twice, and that every member will be returned by *setIterate* if it is called a sufficient number of times (equal to the size of the set). SETS also guarantees that all members of a set satisfy the set's membership specification.

Since there is no way to determine the name of an object given a file descriptor for it, *setIterate* allows an application to ask for the member's name along with the open descriptor. An application requests a name by supplying a buffer and setting SETS_GETNAME in the *flags* argument. *buf* should point to a valid buffer, and *bufsize* should contain the length of the buffer in bytes.

It is sometimes the case that SETS was unable to fetch a member due to an fetch error. If the application wishes to see these error codes, it should supply a pointer to an integer in the *error* field. If the *error* field is set and the SETS_GETERR flag is set in *flags, setIterate* will return error codes for failed member fetches (possibly returning the name as will if SETS_GETNAME is also set), returning a 0 to indicate the error. If not set, SETS will skip over that member but mark it as seen.

RETURN VALUE

Upon successful completion, *setIterate* returns a (non-zero,non-negative) open file descriptor which will give the application read-only access to a previously unyielded set member. This descriptor can be used for any non-mutating operation that accepts file descriptors, such as *close(2).* Although this descriptor will be automatically closed when the set is closed (such as via *setClose(2)),* it is usually good programming style to close a file descriptor when it is no longer needed.

If there are no more members to yield and the set is fully expanded, *setIterate* returns 0. A 0 may also be returned if SETS_GETERR was set, but one can distinguish between these two cases by testing *error,* which will be 0 if the iterator has terminated. A return code of -1 indicates an error, the global integer variable *errno* is set to indicate the error.

ERRORS

The named file is opened unless one or more of the following are true:

[EBADF] *d* is not an active set descriptor.

[EINVAL] The process has no open sets; SETS_GETNAME is set and *buf* is NULL or bufsize is negative; or SETS_GETERR is set but *error* is NULL.

[EOPNOTSUPP] Sets are not supported on this machine.

SEE ALSO

open(2), setOpen(2), setUnion(2), setRestrict(2), setDigest(2)

BUGS

Unfortunately the way Unix returns errors is really ugly so *setIterate* cannot raise an exception or in some other way distinguish between set errors, fetch errors, and termination of the iterator.

NAME

setMember – Membership predicate over sets

SYNOPSIS

#include <sys/set.h>

int setMember(int set, char *path);

DESCRIPTION

setMember can be used to determine if a particular file specified by *path* is a member of the set specified by the set handle *set* (see *setOpen(2)).* The name of *path* should be the name of the object relative to the set, i.e. the name that resulted from SETS evaluation the specification when the set was created (see *setOpen(2)).* Alternatively, the name of objects can also be obtained from *setDigest(2).* *setMember* returns a nonzero value if the object is a member, otherwise it returns zero.

If SETS has not yet fully determined the set's membership, it can only answer correctly if the file is a member of this set. If the member is not known to be a member at the time *setMember* is called, *setMember* will return 0 and the test *IS_PARTIAL(return),* will succeed, where *return* is the value returned by *setMember.*

RETURN VALUE

Upon successful completion, *setMember* will return a negative integer if the object is not a member, and a positive integer if it is.

ERRORS

If either argument is invalid, *setMember* will return zero and set *errno* to indicate the error. The following values are possible:

[ENOTDIR] A component of the path prefix is not a directory.

[EAMBIG] Due to the nature of sets it cannot be told if the object is a member.

[EINVAL] The pathname contains a character with the high-order bit set.

[EINVAL] The set handle does not refer to an open set.

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] A component of the path name that must exist does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EFAULT] *Path* points outside the process's allocated address space.

[EOPNOTSUPP] Sets are not supported on this machine.

SEE ALSO

setOpen(2), setIterate(2), setDigest(2)

BUGS

Unfortunately the way Unix returns errors is really ugly so that there is no easy or clean way to indicate partial membership expansion in a clean way.

NAME

setOpen – open a set for perusal.

SYNOPSIS

#include <sys/set.h>

int setOpen(char *specification, int flags);

DESCRIPTION

setOpen opens the set specified by *specification* for perusal, either by iteration or by digestion. The *flags* argument is used to select specific behavior for the set or to indicate intended use. *setOpen* returns a set descriptor which can be used in future operations on this set. Since sets are dynamic and transitory in nature, the set will only exist until *setClose(2)* is called.

Specification is the address of a string of ASCII characters representing a specification, terminated by a null character. The syntax and semantics of the specification language are beyond the scope of this man page, please refer to the Chapter 4 of the dissertation for more details.

There are a number of flags which can be passed in, which allow the application to request specific behaviors for this set.

SETS_NODUP_ON_FORK	Prevent children from inheriting this set.
SETS_DUP_ON_FORK	Duplicate set in children, but do not share iterator.
SETS_SHARESET	Share set and iterators with children.
SETS_ANTICIPATE_DIGEST	Tell system to aggressively expand membership.
SETS_ANTICIPATE_ITERATE	Tell system to aggressively begin to prefetch members.

RETURN VALUE

The value returned by *setOpen* can be used as an argument to other set operations, such as *setIterate(2).* However, other Unix operations that expect a file descriptor will fail if given a set descriptor. The new descriptor will remain open across *execve* system calls; see *close*(2). If the operation fails, -1 will returned and *errno* will hold a code indicating the error that occurred.

The system imposes a limit on the number of file and set descriptors open simultaneously by one process. *Getdtablesize*(2) returns the current system limit.

ERRORS

The named set is opened unless one or more of the following are true:

[EINVAL]	The specification contains a character with the high-order bit set, or the specification string is Null.
[ENAMETOOLONG]	A component of a specification exceeded 255 characters, or an entire specification exceeded 1023 characters.
[ENOENT]	A component of the specification that must exist does not exist.
[ELOOP]	Too many symbolic links were encountered in translating the specification.
[EMFILE]	The system limit for open set descriptors per process has already been reached.
[ENFILE]	The system set table is full.
[EFAULT]	*Specification* points outside the process's allocated address space.
[EOPNOTSUPP]	Sets are not supported on this machine.

SEE ALSO

setClose(2), fork(2), dup(2), getdtablesize(2)

NAME

setRestrict – Open a subset by applying a restriction to an open set.

SYNOPSIS

#include <sys/sets.h>

setRestrict(int set, char *specification, int flags);

DESCRIPTION

setRestrict creates a new set by applying a restriction function to an existing set. The new set is guaranteed to be a subset (or equivalent) to the set specified by *set.* The *specification* argument is either a string interpretable by SETS (see below) or is the pathname of an executable function to be run by SETS. The *flags* argument is used to specify set behavior, ala the flags used in *setOpen(2). setRestrict* returns a set descriptor which can be used in future operations on this set. Since sets are dynamic and transitory in nature, the set will only exist until *setClose(2)* is called on this set.

The specification is expanded relative to the base (input) set's membership. For instance, the specification string ''*'' would select all members of the base set for membership in the new set, ''d*'' would select only those that begin with a ''d''.

The system imposes a limit on the number of file and set descriptors open simultaneously by one process. *Getdtablesize*(2) returns the current system limit.

RETURN VALUE

The value returned by *setRestrict* can be used as an argument to other set operations, such as *setIterate(2).* However, other Unix operations that expect a file descriptor will fail if given a set descriptor. The new descriptor is set to remain open across *execve* system calls; see *setClose*(2). If the operation fails, -1 will returned and *errno* will hold a code indicating the error that occurred.

ERRORS

The restricted set is opened unless one or more of the following are true:

[EBADF] *set* is not an active set descriptor.

[EINVAL] The process has no open sets.

[ENAMETOOLONG] A component of a specification exceeded 255 characters, or an entire specification exceeded 1023 characters.

[EMFILE] The system limit for open file descriptors per process has already been reached.

[ENFILE] The system-wide set table is full.

[EFAULT] *Specification* points outside the process's allocated address space.

[EOPNOTSUPP] Sets are not supported on this machine.

SEE ALSO

close(2), dup(2), getdtablesize(2), setUnion(2), setOpen(2)

NAME

setSize – Determine the cardinality of (number of elements in) a set.

SYNOPSIS

#include <sys/sets.h>

int setSize(int set);

DESCRIPTION

setSize returns the number of elements in a set. *set* is the handle (descriptor) of a set returned by *setOpen(2)*. *setSize* either returns the actual number of elements (if the set's membership is fully defined) or an approximation which is guaranteed to be less than or equal to the actual numbers of the set (if the membership is only partly defined). In this case, the test *IS_PARTIAL(return)*, will succeed, where *return* is the value returned by *setSize*.

RETURN VALUE

The value returned is the size of the set, or -1 if an error occurs. In case of an error, errno will be set to indicate the nature of the error that occurred.

ERROR

The following errors are possible:

[EBADF] *set* is not an active set descriptor.

[EINVAL] The process has no open sets.

[EOPNOTSUPP] Sets are not supported on this machine.

SEE ALSO

setOpen(2), setUnion(2), setRestrict(2), setIterate(2)

BUGS

Unfortunately the way Unix returns errors is really ugly so that there is no easy or clean way to indicate partial membership expansion in a clean way.

NAME

setUnion – create a new set to be the union of 2 existing sets.

SYNOPSIS

#include <sys/set.h>

int setUnion(int set1, int set2, int flags);

DESCRIPTION

setUnion creates a new set which is the union of two existing sets. *set1* and *set2* are handles for sets created earlier by *setOpen(2)*, *setUnion(2)*, *setIntersect(2)*, or *setRestrict(2)*. The *flags* argument selects the desired behavior for this set, see *setOpen(2)* for more details.

setUnion returns a set descriptor which can be used in future operations on this set. Since sets are dynamic and transitory in nature, the set will only exist until *setClose(2)* is called on this set.

RETURN VALUE

The value returned by *setUnion* can be used as an argument to other set operations, such as *iterate(2)*. However, other Unix operations that expect a file descriptor will fail if given a set descriptor. The new descriptor is set to remain open across *execve* system calls; see *close*(2). If the operation fails, -1 will returned and *errno* will hold a code indicating the error that occurred.

The system imposes a limit on the number of set descriptors open simultaneously by one process. *Getdtablesize*(2) returns the current system limit.

ERRORS

The named file is opened unless one or more of the following are true:

[EBADF] The set handles *set1* or *set2* do not refer to valid sets.

[EMFILE] The system limit for open set descriptors per process has already been reached.

[ENFILE] The system set table is full.

[EINVAL] The process has no open sets.

[EOPNOTSUPP] Sets are not supported on this machine.

SEE ALSO

close(2), dup(2), getdtablesize(2), setIntersect(2), setRestrict(2)

NAME

setWeight – Assign weights (or priorities) to members

SYNOPSIS

#include <sys/set.h>

int setWeight(int set, int *weights, int count);

DESCRIPTION

setWeight can be used to inform SETS of the relative importance of the set members. *weights* is an integer array with *count* elements, where *count* should be less than or equal to the set's size. The order of the elements in the *weight* array should match the order in which *setDigest(2)* returned the names of the members. Each element of the array should contain the relative importance of that member. The absolute value of the weight is unimportant. As a result of this call, the set's member array will be sorted by member's weight, with ties broken arbitrarily.

RETURN VALUE

Upon successful completion, *setWeight* will return a negative integer will return zero and set *errno* to indicate the error.

ERROR

The following values are possible for *errno:*

[EBADF] *set* is not an active set descriptor.

[EINVAL] The process has no open sets.

[EOPNOTSUPP] Sets are not supported on this machine.

SEE ALSO

setOpen(2), setIterate(2), setDigest(2)

Bibliography

[1] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A new kernel foundation for Unix development. In *Summer USENIX Conference Proceedings* (Atlanta, 1986).

[2] Bach, M. J. *The Design of the Unix Operating System.* Prentice Hall, Inc. A division of Simon & Schuster, Englewood Cliffs, New Jersey 07632, 1986. Chapter 3: The Buffer Cache.

[3] Baker, M. G., Hartman, J. H., Kupfer, M. D., Shirriff, K., and Ousterhout, J. K. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (October 1991).

[4] Baldazo, R. Navigating with a web compass. *Byte 21*, 3 (Mar. 96).

[5] Beck, M., Böhme, H., Dziadzka, M., Kunitz, U., Magnus, R., and Verworner, D. *Linux Kernel Internals.* Addison-Wesley Publishing Company, Inc., 1996. Comes with a CD-ROM containing a distribution of Linux.

[6] Berners-Lee, T., Cailliau, R., Luotonen, A., Nielsen, H., and Secret, A. The World Wide Web. *Commun. ACM 37*, 8 (August 1994).

[7] Berners-Lee, T., Fielding, R., and Frystyk, H. HTTP/1.0 Internet Draft, third edition. IETF Hypertext Transfer Protocol (HTTP) Working Group, Nov. 1994. Available as http://www.w3.org/hypertext/WWW/Protocols/.

[8] Bershad, B., and Pinkerton, C. B. Watchdogs – extending the UNIX file system. *Computing Systems 1*, 2 (Spring 1988).

[9] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., and Chambers, C. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (December 1995).

[10] Birrell, A., Levin, R., Needham, R., and Schroeder, M. Grapevine: An exercise in distributed computing. *Commun. ACM 25*, 4 (April 1982).

[11] Bowman, M., Spasojevic, M., and Spector, A. File system support for search. Transarc white paper, 1994.

[12] Cao, P. *Application Controlled File Caching and Prefetching.* PhD thesis, Princeton University, 1996. Available as technical report Princeton TR-CS-522-96.

[13] Cao, P., Felten, E. W., Karlin, A., and Li, K. A study of integrated prefetching and caching strategies. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May 1995).

[14] Cao, P., Felten, E. W., and Li, K. Implementation and performance of application-controlled file caching. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation* (November 1994).

[15] Cate, V. Alex - a global file system. In *Proceedings of the USENIX 1992 File Systems Workshop* (Ann Arbor, MI, 1992).

[16] Chen, C. M., and Roussopoulos, N. Adaptive database buffer allocation using query feedback. In *Proceedings of the 19th VLDB Conference* (Dublin, Ireland, 1993).

[17] Chen, J. B., and Bershad, B. N. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (December 1993).

[18] Chou, H., and DeWitt, D. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 11th Int. Conf. on Very Large Data Bases* (Stockholm, 1985).

[19] Curewitz, K. M., Krishnan, P., and Vitter, J. S. Practical prefetching via data compression. In *Proceedings of the 1993 ACM Conf. on Management of Data (SIGMOD)* (May 1993).

[20] Eggert, P., and Parker, D. File systems in user space. In *Winter USENIX Conference Proceedings* (San Diego, 1993).

[21] Emtage, A., and Deutsch, P. Archie – an electronic directory service for the internet. In *Winter USENIX Conference Proceedings* (San Francisco, 1992).

[22] Enderton, H. B. *Elements of Set Theory.* Academic Press, a division of Harcourt Brace Javanovich, Publishers, 111 Fifth Avenue, New York, New York 10003, 1977.

[23] Engler, D. R., Kaashoek, M. F., and O'Toole Jr., J. An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (December 1995).

[24] Firestone, D., Project Manager, Tables of Contents, INC. Personal Communications, Message ID: 01BB6E44.AAF476C0@dick.firestone. Jul 10, 1996.

[25] Fox, A., Gribble, S. D., Brewer, E. A., and Amir, E. Adapting to network and client variability via on-demand dynamic distillation. In *The ACM Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, October 1996).

[26] Gifford, D. K., Jouvelot, P., Sheldon, M. A., and O'Toole, Jr., J. W. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (October 1991).

[27] Glassman, S. A caching relay for the world wide web. *Computer Networks and ISDN Systems 27*, 2 (Nov. 1994). Special Issue: selected papers from the First International WWW Conference.

[28] Griffioen, J., and Appleton, R. Reducing file system latency using a predictive approach. In *Summer USENIX Conference Proceedings* (June 1994).

[29] Griffioen, J., and Appleton, R. The design, implementation, and evaluation of a predictive caching file system. Tech. Rep. CS-264-96, Department of Computer Science, University of Kentucky, June 1996.

[30] Grimshaw, A. S., and Loyot, E. C., J. ELFS: Object-oriented extensible file systems. Tech. Rep. TR-91-14, Computer Science Department, University of Virginia, July 1991.

[31] Gwertzman, J. S., and Seltzer, M. The case for geographical push-caching. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HOTOS-V)* (Orcas Island, WA, May 1995).

[32] Harbison, S. P. *Modula-3*. Prentice Hall, 1992.

[33] Hardy, D. R., and Schwartz, M. F. Essence: A resource discovery system based on semantic file indexing. In *Winter USENIX Conference Proceedings* (San Diego, 1993).

[34] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., and West, M. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6*, 1 (Feb. 1988).

[35] Jacobson, V. Congestion avoidance control. In *Proceedings of the SIGCOMM '88 Conference on Comuunications Architectures and Protocols* (1988).

[36] John, R., Bowman, M., and Spasojevic, M. An extensible type system for wide-area information management. In *Proceedings of the 1994 IWOOS Workshop* (August 1995).

[37] Joseph, A. D., deLespinasse, A. F., Tauber, J. A., Gifford, D. K., and Kaashoek, M. F. Rover: A toolkit for mobile information access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (December 1995).

[38] Joy, W. An introduction to the C shell. In *Unix User's Manual, Supplementary Documents*, M. J. Karels and S. J. Leffler, Eds. Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, 1980.

[39] Kahle, B., and Medlar, A. An information system for corporate users: Wide area information servers. *ConneXions – The Interoperability Report 5*, 11 (Nov. 1991).

[40] Kimbrel, T., Tokmins, A., Patterson, R. H., Bershad, B., Cao, P., Felten, E. W., Gibson, G. A., Karlin, A. R., and Li, K. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation* (Oct. 1996).

[41] Kistler, J. *Disconnected Operation in a Distributed File System.* PhD thesis, Carnegie Mellon University, 1993. Available as technical report CMU-CS-93-156.

[42] Kleiman, S. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Summer USENIX Conference Proceedings* (Atlanta, 1986).

[43] Korner, K. Intelligent caching for remote file service. In *Proceedings of the 10th International Conference on Distributed Computing Systems* (1990).

[44] Kotz, D., and Ellis, C. Practical prefetching techniques for parallel file systems. In *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems* (Miami Beach, Florida, Dec. 1992).

[45] Kuenning, G. H. The design of the SEER predictive caching system. In *Proceedings of the Workshop on Mobile Computing Systems and Applications* (Santa Cruz, CA, Dec. 1994).

[46] Kumar, P. *Mitigating the Effects of Optimistic Replication in a Distributed File System.* PhD thesis, Carnegie Mellon University, 1994. Available as technical report CMU-CS-94-215.

[47] Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. *The Design and Implementation of the 4.3BSD Unix Operating System.* Addison-Wesley Publishing Company, Inc, 1989.

[48] Liskov, B., and Guttag, J. *Abstraction and Specification in Program Development.* The MIT EECS Series. MIT Press, Cambridge, MA ; McGraw-Hill, New York, 1986.

[49] Long, D., Carroll, J., and Park, C. A study of the reliability of Internet sites. In *Proceedings of the 10th IEEE Symposium on Reliable Distributed Systems* (Pisa, Italy, Sept. 1991).

[50] Manber, U., and Wu, S. Glimpse: A tool to search through entire file systems. In *Winter USENIX Conference Proceedings* (1994). Also available as The University of Arizona Department of Computer Science Technical Report TR 93-34.

[51] Mauldin, M., and Leavitt, J. Web-agent related research at the CMT. In *Proceedings of the ACM Special Interest Group on Networked Information Discovery and Retrieval (SIGNIDR-94)* (Aug. 1994). Also available as http://fuzine.mt.cs.cmu.edu/mlm/signidr94.html.

[52] McCahill, M. The Internet Gopher: A distributed server information system. *ConneXions – The Interoperability Report 6*, 7 (July 1992).

[53] McKusick, M., Joy, K., Leffler, W., and Fabry, R. A fast file system for Unix. *ACM Trans. Comput. Syst. 2*, 3 (August 1984).

[54] Mogul, J. C. The case for persistent-connection HTTP. In *Proceedings of the SIGCOMM '95 Conference on Comuunications Architectures and Protocols* (1995). An expanded version is available as Digital Equipment Corporation Western Research Lab Research Report 95/4.

[55] Mowry, T. C., Demke, A. K., and Krieger, O. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation* (Oct. 1996).

[56] Mummert, L., Ebling, M., and Satyanarayanan, M. Exploiting weak connectivity for mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (December 1995).

[57] NCSA Mosaic 2.6 for X Window System. National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign. Available as http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/NCSAMosaicHome.html.

[58] Nelson, M., Welch, B., and Ousterhout, J. Caching in the Sprite Network File System. *ACM Trans. Comput. Syst. 6*, 1 (Feb. 1988).

[59] NetBSD Operating System. The NetBSD Foundation, 131 Santa Marina, San Francisco, CA 94110. See http://www.netbsd.org.

[60] Neuman, B. The prospero file system: A global file system based on the virtual system model. *Computing Systems 5*, 4 (Fall 1992).

[61] Niblack, W., Barber, R., Equitz, W., Flickner, M., Glasman, E., Petkovic, D., Yanker, P., and Faloutsos, C. The QBIC project: Querying images by content using color, texture, and shape. Tech. Rep. RJ 9203 (81511), IBM Research Division, 1993.

[62] OSF/1 Operating System. Open Systems Foundation, 11 Cambridge Center, Cambridge, MA 02142-1405. See http://www.osf.org.

[63] Ousterhout, J. K. The role of distributed state. In *CMU Computer Science, a 25th Anniversary Commemorative*, R. F. Rashid, Ed. ACM Press, 1991, ch. 8.

[64] Ousterhout, J. K., Da Costa, H., Harrison, D., Kunze, J. A., Kupfer, M., and Thompson, J. G. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (December 1985).

[65] Padmanabhan, V. N., and Mogul, J. C. Using predictive prefetching to improve world wide web latency. *ACM SIGCOMM Computer Communication Review 26*, 3 (July 1996).

[66] Patterson, R. H., and Gibson, G. A. Exposing I/O concurrency with informed prefetching. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems, Austin, TX* (Sept. 1994).

[67] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating System Principles* (Dec. 1995).

[68] Patterson, R. H., Gibson, G. A., and Satyanarayanan, M. A status report on research in transparent informed prefetching. *ACM Operating Systems Review 27*, 2 (April 1993).

[69] PINKERTON, B. Finding what people want: Experiences with the WebCrawler. In *Proceedings of the Second International WWW Conference: Mosaic and the Web* (Sept. 1994). Also available as http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/WWW2_Proceedings.html.

[70] POSTEL, J., AND REYNOLDS, J. File transfer protocol (FTP). Network Working Group Request for Comments (RFC) 959, ISI, October 1985. Available as http://www.w3.org/hypertext/WWW/Protocols/rfc959/Overview.html.

[71] RELATIONAL DATABASE SYSTEMS, INC. Informix. 4 100 Bohannon Drive Menlo Park, CA 94025.

[72] ROSENBLUM, M., AND OUSTERHOUT, J. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (February 1992).

[73] RUEMMLER, C., AND WILKES, J. UNIX disk access patterns. In *Winter USENIX Conference Proceedings* (San Diego, 1993).

[74] SACCO, G., AND SCHKOLNICK, M. Buffer management in relational database systems. *ACM Trans. Database Syst. 11*, 4 (Dec. 1986).

[75] SALTON, G., AND MCGILL, M. *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

[76] SALTZER, J., REED, D., AND CLARK, D. End-to-end arguments in system design. *ACM Trans. Comput. Syst. 2*, 4 (Nov. 1984).

[77] SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. Design and implementation of the Sun Network File System. In *Summer USENIX Conference Proceedings, Portland* (1985).

[78] SATYANARAYANAN, M. A study of file sizes and functional lifetimes. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles* (December 1981).

[79] SATYANARAYANAN, M. *RPC2 User Guide and Reference Manual.* School of Computer Science, Carnegie Mellon University, Oct. 1991.

[80] SATYANARAYANAN, M. Mobile information access. *IEEE Personal Communications 3*, 1 (February 1996).

[81] SATYANARAYANAN, M., HOWARD, J., NICHOLS, D., SIDEBOTHAM, R., SPECTOR, A., AND WEST, M. The ITC Distributed File System: Principles and design. In *Proceedings of the 10th ACM Symposium on Operating System Principles, Orcas Island* (Dec. 1985).

[82] SATYANARAYANAN, M., KISTLER, J., KUMAR, P., OKASAKI, M., SIEGEL, E., AND STEERE, D. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput. 39*, 4 (Apr. 1990).

[83] SATYANARAYANAN, M., AND SPASOJEVIC, M. AFS and the Web: Competitors or collaborators? In *ACM SIGOPS European Workshop, 1996* (September 1996).

[84] SCHWARTZ, M. F., EMTAGE, A., KAHLE, B., AND NEUMAN, B. C. A comparison of internet resource discovery approaches. *Computing Systems 5*, 4 (Fall 1992).

[85] SHAW, M., WULF, W. A., AND LONDON, R. L. Abstraction and verification in Alphard: Defining and specifying iteration and generators. *Commun. ACM 20*, 8 (Mar. 1977). Reprinted in Tutorial: Programming Language Design, text for IEEE Tutorial by Anthony I. Wasserman, 1980, pp. 145-155.

[86] SHOENS, K., LUNIEWSKI, A., SCHWARZ, P., STAMOS, J., AND THOMAS, J. The Rufus system: Information organization for semi-structured data. In *Proceedings of the 19th International Conference on Very Large Data Bases* (Dublin, Ireland, Aug. 1993).

[87] SMITH, A. J. Disk cache – miss ratio analysis and design considerations. *ACM Trans. Comput. Syst. 3*, 3 (Aug. 1985).

[88] SPERO, S. E. Analysis of HTTP performance problems. http://sunsite.unc.edu/mdma-release/http-prob.html, July 1994.

[89] SPERO, S. E. HTTP-NG architectural overview. http://www.w3.org/pub/WWW/Protocols/HTTP-NG/http-ng-arch.html, 1995.

[90] STEELE, JR., G. L. *Common LISP*. Digital Press, 1984.

[91] STEERE, D., KISTLER, J., AND SATYANARAYANAN, M. Efficient user-level file cache management on the Sun vnode interface. In *Summer USENIX Conference Proceedings* (Anaheim, CA, 1990).

[92] STEERE, D., AND SATYANARAYANAN, M. A case for dynamic sets in operating systems. Tech. Rep. CMU-CS-94-216, School of Computer Science, Carnegie Mellon University, November 1994.

[93] TAIT, C. D., AND DUCHAMP, D. Detection and exploitation of file working sets. In *Proceedings of the 11th International Conference on Distributed Com puting Systems* (Arlington, TX, 1991).

[94] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (December 1993).

[95] WEBCOMPASS 1.0. Quarterdeck, Corp. Marina del Ray, CA. (800) 683-6696. Additional information is available at http://www.quarterdeck.com.

[96] WING, J., AND STEERE, D. Specifying weak sets. In *Proceedings of the International Conference on Distributed Computer Systems* (Vancouver, June 1995). Also available as Carnegie Mellon University School of Computer Science technical report CMU-CS-94-194.