Using Dynamic Sets to Speed Search in World Wide Information Systems

David C. Steere

March 27, 1995

CMU-CS-95-174

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Search on wide area distributed systems is plagued by the high latencies inherent in remote access. A solution is to prefetch information before it is requested by the searcher to hide latency. But this raises the problem of knowing what to prefetch, since fetching data that will not be used can actually hurt performance. This paper proposes extending the Unix file model to support *dynamic sets*, short-lived and unordered collections of objects created by searchers to hold the results of queries. An object's membership in a set is a *hint* of future access, informing the system that prefetching that object can improve performance. An additional benefit of using set membership as the hint is that it allows the system to determine the order in which objects are returned to the searcher, further increasing the opportunity for performance improvement. This paper presents the design of SETS, a system extension to Unix to provide dynamic sets. A performance evaluation of SETS shows dynamic sets offer substantial opportunity to reduce the aggregate latency to fetch a group of objects. Experiments on existing world wide information systems show as much as a factor of 8 performance improvement from using sets.

This research was supported by the Air Force Materiel Command (AFMC) and ARPA under contract number F196828-93-C-0193. Additional support was provided by the IBM Corporation, Digital Equipment Corporation, and Intel Corporation.

Keywords: Search, Browsing, Information Retrieval, Distributed Systems, File Systems, Prefetching, Latency, Dynamic Sets

1. Introduction

The last few years have seen the emergence of a new class of data repository, world wide information systems (WWIS). WWIS, such as FTP[19], WWW[1], Gopher[15], and world-wide AFS[25], consist of loose collections of data repositories spanning the globe, and store massive amounts of data. The vast amount of accessible data makes *search* a critical application for these systems. Unfortunately, accessing data on remote machines has an inherently high latency, and limits the performance of search.

This paper proposes extending the current file model to support a new type of object, the dynamic set. A dynamic set is a short lived and unordered collection of objects. The creation of a dynamic set is a very good hint that the members of the collection will soon be requested by the application. This hint can be exploited to obtain the benefits of prefetching: use of available parallelism between servers or disks, overlap of I/O and processing, and improved network utilization. In addition, a set has no inherent order, and so the system is free to impose any ordering on the members and thus improve throughput. For example, it could first yield members that happen to be local, overlapping the time to fetch remote data with the cost to process the local objects. A third benefit is that direct support for sets of objects provides a richer application interface to search tools. For instance, the use of sets leads naturally to the use of iterators. Iteration is a useful operation when searching, but one with only rudimentary support in today's systems.

The idea of dynamic sets is based on three observations. First, caching, which uses local copies of recently used objects to satisfy fetch requests, will not work well for search. Search exhibits poor temporal locality, and it has been shown that caching achieves only limited benefits in world wide information systems[5]. Second, prefetching data based on past observations of file requests is ineffective for search. By nature, each search tends to access different objects, and using observations of previous searches to predict a future search's file activity is likely to produce inaccurate predictions. Basing prefetch decisions on inaccurate predictions can actually harm performance, since prefetching consumes potentially scarce resources[11, 24]. Third, search is in essence a process of identifying a set of candidate objects and examining the objects to determine which if any satisfy some desired properties. For example, when searching for the definition of a procedure in a large piece of code, one would first determine which source files should be included in the search, and then examine each in turn to determine which contain references to the procedure in question.

These three observations suggest that the Unix file model is insufficient for supporting search. Fortunately, the nature of search provides an opportunity. Since one is likely to examine some or all of the candidate objects identified in the first part of the search, disclosing the names of the objects to the system can allow it to prefetch with the knowledge that doing so will reduce the aggregate latency of the search. A drawback to dynamic sets is that existing applications must be modified in order to use them. However, sets are a natural addition to the Unix file model, and it is relatively easy to modify existing search tools to use them. In addition, applications can achieve substantial performance improvements using dynamic sets. In the experiments discussed in Section 4, dynamic sets provided a speedup of between 1.40 and a 8.7 in elapsed time for searches on existing WWIS.

1.1. Using Dynamic Sets to Aid Search

Search in wide area systems is complicated by several factors. Global coverage implies that the latency due to propagation delay is high. Searches that involve a number of objects repeatedly suffer this latency which produces intolerable delays. In addition, the current global network is actually an interconnection of many local area networks. Transmitting data across long physical distances can involve many network crossings, each of which add to the latency. Another factor that can increase latency is the load from the growing number of users of the Internet. Response times on the order of tens of seconds are not at all uncommon in the WWW. More importantly, the likelihood of communication failure grows with the number of components in the system. Timeouts can effectively block forward progress for the duration of the timeout period. The frequency of failures, together with the lack of synergy between components of typical wide area systems make it impractical to provide the tight consistency guarantees databases provide for queries. Given all these factors, how can dynamic sets help performance? This section answers the question by comparing how one would perform several example searches with and without dynamic sets.

Suppose you want to find the references to a global variable in the source files of a large piece of code stored in a distributed system. This example is a common search activity in a distributed file system. Typically, one would use an operation similar to the Unix command `grep`, as in "`grep varname *.c`" in each of the program's source directories. In Unix, the wildcard "`*.c`" is expanded to a list of file names ending in "`.c`". Each file is opened in turn, read in its entirety while looking for occurrences of "`varname`", and then closed. Although the files to be searched are identified once the wildcard has been expanded, this information cannot be exploited by the system. In addition, the order in which the files are opened is fixed, even though the ordering is artificial.

Modifying `grep` to use dynamic sets would yield three benefits. First, the files named by "`*.c`" could be safely prefetched, since `grep` is almost certainly going to access them. Through prefetching, files on separate servers may be fetched in parallel, and the fetching of some files may overlap the processing of others. Second, prefetching may result in better network and server utilization, as many requests may be queued at a time. Third, the system can reorder the fetching of the files, since `grep` does not require that the files be processed in any order. Thus if some of the files are local and others remote, the system could return the local files first to reduce the time to begin processing the data, and could overlap processing these files with the fetching of remote files.

For a second example, suppose that while travelling, you would like to use your portable computer and cellular phone to look at the menus of local restaurants before choosing where to stop for dinner[1]. This search consists of locating, downloading, and displaying images of restaurant menus. These menus could be located by querying one of the many WWW search engines, perhaps one maintained by a local travel bureau[2]. However, once a list of menus is obtained the user is forced to fetch and examine them *serially*. If the user is accessing the data over a telephone, as supposed, this search would be an extremely costly operation. (Fetching a typical image of 20KB would take at least 10 seconds over a 14.4 Kbps modem).

With dynamic sets, the high latency of fetching data over a slow link can at least partially be overlapped with the processing of the data. In this example, "processing" is done by the user and so the processing time is on the order of seconds. If the user spends at least 10 seconds on average reading a menu, the next menu will be available as soon as the user requests it. User processing is in fact common for searches on the WWW: popular WWW clients such as NCSA Mosaic[3] or Netscape[4] have no standard facility for allowing programs to direct a search.

As a third example, suppose that you would like to combine the results of queries to some number of search engines (indexes of objects in a system) in order to find research papers on some subject. Unlike the performance benefits highlighted by the first two examples, this example shows the benefit of extending the Unix programming model to support dynamic sets. Currently, there is no easy way to merge the results of queries to multiple indexes. But with direct support for dynamic sets, one can create a set to hold the results of each query, and then merge the results into a single set using standard set functions like union, intersect, or subset.

A related benefit of dynamic sets is that one can use the set to act as a placeholder in the processing of the query results. With sets the user can explore the current member's local web (subtree), and then jump on to the next element without having to backtrack. In this regard, sets are similar to cursors in SQL[6] or iterators in programming languages (for instance in CLU[13]). Thus a user of dynamic sets can easily perform a hybrid of depth and breadth first searches in wide area systems.

The rest of the paper describes SETS, an extension to the Unix file model to provide support for dynamic sets. Section 2 presents an overview of the design of SETS. Section 3 discusses some implementation details of SETS relevant to the

---

[1] For example, look at `http://www2.ari.net:85/edc/dcdining/dcdining.html` on the WWW.

[2] For example examine `http://www.fleethouse.com/fhcanada/western/bc/van/van_home.htm` on the WWW.

[3] http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/NCSAMosaicHome.html

[4] http://home.mcom.com/home/welcome.html

evaluation, which is presented in Section 4. Section 5 discusses related work, and the paper closes with a discussion of potential extensions to SETS.

2. Design of SETS

Basic SETS Operations	`setHandle`	**`setOpen`**(char *setPathname);
	`errorCode`	**`setClose`**(setHandle set);
	`fileDesc`	**`setIterate`**(setHandle set, int flags);
	`errorCode`	**`setDigest`**(setHandle set, char *buf, int count);
Auxiliary SETS Operations	`setHandle`	**`setUnion`**(setHandle set1, setHandle set2, int flags);
	`setHandle`	**`setIntersect`**(setHandle set1, setHandle set2, int flags);
	`setHandle`	**`setSubset`**(setHandle set, char *buf, int count);
	`errorCode`	**`setRewindIterator`**(setHandle set);
	`errorCode`	**`setRewindDigest`**(setHandle set);
	`int`	**`setSize`**(setHandle set);
	`bool`	**`setMember`**(setHandle set, char *elem);

This figure lists the operations in the SETS API. The first four operations are the core of the interface, and allow an application to create and open a set, close and delete it, invoke an iterator to yield a member, and produce a summary of the set. The other operations allow one to create a new set to hold the union or intersection of existing sets, create a subset containing the members listed in `buf`, reinitialize the iterator or digest, determine the size of the set, and discover if an object is a member of the set.

Figure 1: SETS application programmer interface.

The primary goal of SETS is to enhance the performance of search without requiring modifications to existing protocols or servers. Thus each user can benefit from dynamic sets by choosing to use them without affecting other users of the system. In addition, by using existing systems as is, it is easier to extend SETS to access data from more systems.

A second goal is to show that dynamic sets could support different kinds of search and search in different systems. For instance, the SETS interface should be usable both when browsing and searching. In addition to a flexible interface, it should be possible to extend SETS to support new types of search engines. The manner in which a collection of objects is identified should not affect the performance benefits that dynamic sets offer in fetching the collection.

A third goal is to minimize the overhead of SETS. Since the time to create and open a set cannot be overlapped with I/O, it adds to the aggregate latency to process the set. In addition, the overhead of sets may increase the application's time to process the data if its memory or CPU consumption is excessive.

To achieve these goals and to support search in wide area systems, dynamic sets need to have two properties. First, they should be short-lived. Dynamic sets are intended to hold the results of a query while the searcher processes them. Thus a set can be deleted when the search is finished. Even if a query is rerun, it is unlikely that the results of the previous run will be useful. Set membership may change over time, and the collection of objects for a past search may not be relevant once that search is completed. In addition, the space of collections is exponentially larger than the number of objects, and it would be infeasible to store any but a miniscule fraction of them. However, one can manually create a persistent copy of a set by copying its members to the local file system. Second, a dynamic set itself should not be distributed, although its members may reside on other machines. A dynamic set may change over time through addition of members, or as a result of lazy evaluation of membership. Maintaining a consistent copy of the set across machines can have serious performance implications.

In addition to these properties, two philosophical principles influenced the design of SETS. First, the primary performance benefit of dynamic sets comes from parallelism, so SETS should be decomposed to allow maximal parallelism between subsystems. For instance, the subsystem that interacts with the application should be decoupled from the subsystem that performs the prefetching to avoid needlessly blocking the application before it requests data.

Second, interfaces should be designed to provide maximal disclosure of future access, but with as little constraint as possible on the lower layer. An example of this is the way in which dynamic set membership informs SETS of future access without imposing an artificial order on access.

The remainder of this section presents the salient aspects of the design of SETS. Section 2.1 discusses the operations in the SETS interface, listed in Figure 1, and gives an example of how an application could be coded to use sets. Then Section 2.2 discusses the semantics of dynamic sets in terms of set membership and the currency of the set members. Finally Section 2.3 describes the architecture of SETS.

2.1. SETS interface

Explicit:	`/afs/projects/*src*/*.c`
Type-specific:	`/afs/staff/\select home where name like "%david%"\`
Executable:	`/afs/sources/%myMakeDepend foo.c%`

Because many Unix users are already familiar with the `csh` wildcard notation, SETS extends this syntax to support set specification. *Explicit* specifications use standard `csh notation`. *Type specific* specifications have two portions: the prefix identifies the type of the query, and the query (delimited by "\") is interpreted by the type manager (e.g. client to an SQL database) to obtain a list of object names. In this case the query is an SQL statement interpretable by the SQL manager mounted at `/afs/staff`. *Executable* specifications cause the command line delimited by the "%" to be run. The program in the command line acts as a predicate over files, returning the names of those files in the argument list that satisfied the predicate. In any of the cases, failures cause the resulting set to be the empty set.

Figure 2: Examples of the three types of names supported by SETS.

The operations in the SETS application programmer interface (API) are listed in Figure 1. The first four operations are the core of the interface. An application creates a set with `setOpen` and deletes it with `setClose`. `setIterate` is an iterator over sets, producing a previously unyielded member on each call, and `setDigest` produces a summary of the set. These operations are discussed in more detail below. `setUnion`, `setIntersect`, and `setSubset` create new sets that hold the union, the intersection, and a subset of the arguments. `setRewindIterator` and `setRewindDigest` reinitialize the iterator or summary of a set. `setSize` returns the number of members in a set, and `setMember` is a predicate indicating whether an object is a member of the set.

Sets are created by calling `setOpen` with a string specifying the set's members. SETS determines the names of the set members by evaluating this specification. SETS is free to control the aggressiveness with which it determines the membership; in fact `setOpen` may return before any evaluation has been done. For instance, on a high performance workstation, it may be appropriate to resolve membership fully when the set is open. But on a hand-held portable it may be better to lazily evaluate the membership specification to avoid resource contention. An advantage of this approach is that the aggressiveness is determined by the level of the system responsible for resource allocation, such as the file system client manager.

`setOpen` returns an open set handle for the newly created set. The handle can then be used as an argument to other set operations. The open set handle is similar to an open file descriptor in Unix. It is owned by the process that created it, but can be shared with the process's children. `setClose` terminates use of a set handle, allowing the system to free any resources used by the corresponding set.

Since it may take time for a set's membership to be fully defined, due to lazy evaluation for instance, several of the operations have been designed to allow partial evaluation. For instance, `setSize` returns the number of elements currently known to be in the set, and an error code indicating whether or not the set has been fully expanded. As another example, `setMember` returns *true* if the object is a member, but if the object is not yet a member, it may return *false* and an indication that the set's membership is only partially resolved.

2.1.1. Set membership specifications

There are three ways to specify membership in a set: *explicit*, *type-specific*, or *executable* specifications. Examples of each are given in Figure 2. *Explicit* specifications use standard `csh` wildcard notation to indicate the names of the members of the set. For instance, `{src1,src2}/*.[ch]` would be the set of all files ending in "`.c`" or "`.h`" in the subdirectories `src1` or `src2`.

Type-specific specifications are strings that can be interpreted as queries by a search engine. SETS allows *wardens*, programs that can interpret such strings, to be *mounted* in the name space (see Section 2.3). These programs act as clients to a type of search engine such as an SQL database or a WAIS index. When `setOpen` is called with a type-specific specification, SETS parses it to obtain the mount point of a warden and a query, and passes the query to the warden. The warden sends the query to a search engine, parses the query results to obtain the names of the set members, and passes these names back to SETS.

Executable specifications are programs that act as predicates over a portion of the system's name space. The program returns the names of the objects that satisfied its predicate to SETS. The prefix of the specification is the directory in which to execute the program, the string delimited by "`%`" is the program name and its arguments.

Because many Unix users are already familiar with the `csh` wildcard notation, SETS overloads the Unix pathname semantics by allowing any component of a name to contain a set specification. Explicit specifications are syntactically identical to pathnames in the `csh`, the other types of specifications are obvious extensions. Specifications are parsed from left to right. Given the specification `/afs/project/*/src/%myMakeDepend *.c%`, SETS would run the program `myMakeDepend *.c` for each directory that matches `/afs/project/*/src/`.

2.1.2. Processing the members of a set

```
handle = setOpen(argv[1]);
while ( (fd = setIterate(handle)) != -1 ) {
    process(fd);
    close(fd);
}
setClose(handle);
```

This figure presents pseudo-code that is typical of the way Unix applications like `grep` would use sets. Many of these applications can process multiple input files, and effectively have this structure already. Thus modifying them to use dynamic sets is very straightforward.

Figure 3: A pseudo-code example using SETS.

SETS provides two ways to process the members of a set, `setIterate` and `setDigest`. The former is an iterator, yielding an open file descriptor of a previously unyielded member. The latter provides attribute information about members which can be used to guide the search. The difference between the two has subtle implications.

Each call to `setIterate` returns an open file descriptor for a previously unyielded member. If one calls `setIterate` a sufficient number of times, every member of the set will be yielded. The advantage of `setIterate` is that it provides the best opportunity for prefetching. Neither sets nor iteration have an implicit ordering, so SETS can choose which element to yield next. Since SETS gets to determine the order, any iteration can be thought of as "sequential", allowing SETS to safely prefetch the "next" object. The disadvantage of `setIterate` is that it constrains the user's choice: the user cannot select which set member to access next. However, there are many situations in which iteration is reasonable, such as the `grep` example in Section 1.1. And by choosing to use iteration, users are rewarded by better performance.

Since `setIterate` returns a file descriptor for the member, SETS must actually *open* the member before yielding it. For some systems, this means that the member must have been completely fetched. For example, systems that cache whole files, such as the Coda file system[22], must have a local copy of an object before it can be opened. For others, only some portion of the file needs to be fetched in order for that file to be yielded by the iterator. In these systems, SETS will yield a partially cached member if no fully cached member is unyielded, but tries to finish fetching one file from a server before starting to fetch the next. This reduces the risk of forcing the application to block when reading the file. Although not addressed in the design, it would be useful to allow applications to suggest whether they would prefer SETS to prefetch whole files or only the first portion. For instance, applications like Netscape that may not read the entire file may prefer that SETS fetch only the first part of a file before going onto the next one.

Figure 3 presents pseudo-code that is typical of the way many Unix applications would use `setIterate`. The application is invoked with the specification of the set to process. The set is opened and the members are produced using `setIterate`. The routine `process()` performs the application specific function, for example, in `grep` it would sequentially read the file and print out lines that contain a specific string. When the iterator has yielded all the elements, the application calls `setClose` to free the resources held by the set. Since applications like `grep` are often written to process several arguments, they already have a similar structure. It is thus quite possible to modify existing applications to use SETS with little or no knowledge of the details of the application. Applications with multiple threads or processes within a process group can use `setIterate` to distribute members of the set for processing.

Unlike `setIterate`, `setDigest` allows the searcher to retain control over the order in which the set members are processed. When called, `setDigest` returns attribute information (a *digest*) about the currently known members of the set. As mentioned previously, set membership can be determined lazily, so that complete membership may not be known when `setDigest` is called. Given the information in a digest, the user or application can then request which member to process next.

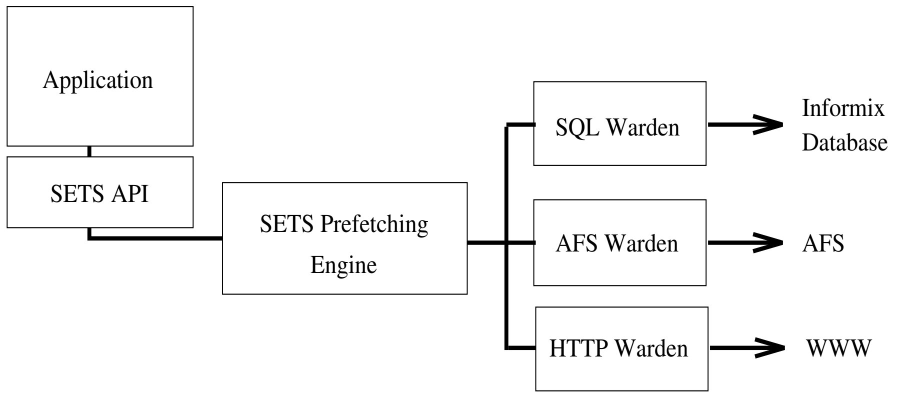
A digest may contain different kinds of information. Currently, a digest only consists of the names of set members. Providing this information has no cost since SETS needs to know the member names anyway. Providing other summary information such as the size of the object, its type (e.g. HTML, GIF, JPEG), the first portion of an HTML document, the abstract of a research paper, or a thumbnail of an image would be useful. However it would also incur additional cost since some of the work to fetch the object may need to be done in order to determine the attribute's value. Future enhancements to SETS may extend `setDigest` to allow the application to specify what information a digest should contain.

2.2. Defining membership in a dynamic set

As stated previously, a set is defined by its membership function. Unlike mathematical sets, however, a dynamic set's membership may change over time: as objects are added and removed from the set, as connectivity changes, or as objects are created and deleted from the system. This is especially true in the presence of long-running queries, lazy evaluation of set membership, and sets that contain objects stored on geographically distant repositories.

Defining set membership requires answering two questions: "What objects should be part of the set?" and "How current do the members of the set need to be?". The first question arises because the state of the system at the time `setOpen` was called (σ_o) may be different than the state at the time `setClose` was called (σ_c). For example, if an object f included in the membership specification of a set S existed in σ_o but was deleted in some state σ_i that occured before σ_c, should it be considered part of S? The second question arises because objects returned by a query may be modified after membership in the set has been determined. If an object f that is part of S has been modified since it was read by the query, should the user see the value f_{σ_o} which satisfied the query, or the current value f_{σ_c} which may not?

In databases queries, the traditional solution is to provide transactional guarantees to ensure the consistency of set membership and the currency of the members[3]. For instance, one could ensure isolation by preventing mutations to the set and any potential set members during the processing of the set. This solution, however, is unreasonable in typical WWIS for several reasons. First, the cost of locking objects in an environment of intermittent connectivity and disconnection like the Internet is prohibitive. Second, current systems do not provide the lower level mechanisms such as transactional updates which are necessary to guarantee strict consistency or currency properties. Third, components of wide area distributed systems tend to be very autonomous; it may be infeasible to achieve the synergy necessary to ensure transactional properties.



SETS is divided into three components. The SETS API interacts with the application, the prefetching mechanism determines set membership and prefetches members, and *wardens* are WWIS-specific clients which allow SETS to access and query data in that system.

Figure 4: The architecture of SETS.

Instead, SETS makes the assumption that users of WWIS are willing to accept weaker consistency guarantees for better performance. Anecdotal evidence bears this out: search engines like Lycos[14] or the Webcrawler[18] 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, for instance the use of asynchronous disk writes in the Unix Fast File System [16], the use of write-back caching in Sun's Network File System[20], or the relaxation of Unix semantics to session semantics in AFS[21].

Although there are a number of valid design points trading consistency for performance[28], SETS chooses to maximize performance and availability and simplify implementation effort by providing relatively weak semantics. In short, SETS guarantees the following three properties.

- Sets are immutable in that the searcher cannot change the set's membership once the set has been created. But due to partial evaluation, the final state of the set is unknown until membership has been fully determined.
- Every object in the set must have satisfied the specification 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.

Fortunately, it is reasonable to provide these properties in SETS. First, since SETS fetches members in their entirety, preserving the local copy of a member for the lifetime of the set will guarantee that the member remain in the set. By preventing mutations to this local copy, SETS also guarantees that a member's value does not change once it has been yielded by `setIterate`. Through this local copy, SETS can ensure the consistency of a set's membership without requiring heavyweight mechanisms such as locks. Second, SETS guarantees that any member of the set must have satisfied the set's specification during its evaluation. Third, these weak guarantees do not prevent SETS from releasing available data due to consistency constraints. Thus SETS can offer useful semantics without compromising performance or availability. The success of this solution of course is directly dependent on the lack of data consistency provided by typical wide area systems. Applications that require stronger guarantees (such as a Bank's account records) would be poorly supported by this approach.

2.3. Architecture of SETS

The chief goal of the architecture is to allow the separate components of SETS to optimize resource usage by providing them maximal autonomy through careful interface design. SETS has three basic components: the SETS API layer, the prefetching mechanism, and the *wardens*. Figure 4 shows how these components are structured.

```
handle      sets_lookup(fileId mountpt, char *query);
errorCode   sets_expand(handle h, char *buffer, int size);
```

These two operations are extensions to the standard low level file system interface[20] to allow SETS to process type-specific specifications. `sets_lookup` asynchronously initiates a call to the warden passing in the query. `sets_expand` retrieves the results of the query, blocking until results become available.

Figure 5: SETS warden interface.

The SETS API layer exports the operations listed in Figure 1. It is responsible for ensuring SETS maintains dynamic set properties, allocating and deallocating set and file handles, managing the SETS data structures, and initiating asynchronous operations for set membership resolution and member prefetching. If a request has been completed by the time its results are needed, the API layer can return immediately to the application. If not, this layer will block the caller until the results become available.

The prefetching mechanism performs the work of determining set membership and fetching set members. Determining set membership is the process of discovering the names of the objects that are contained in the set. Depending on the type of membership specification, SETS may manipulate names (for explicit specifications), interact with wardens (for type-specific specifications), or execute programs (for executable specifications) in order discover the names of the members. When the name of a member becomes known, SETS must decide whether or not to initiate a prefetch operation on it. Determining an optimal prefetching policy is complex: aggressive fetching may overrun available resources, yet postponing the fetch may force the application to delay on the object. Ideally, a dynamically adaptable algorithm that considers current resource utilization would be used. Currently, SETS uses one of two static policies, discussed below in Section 3.3.

If SETS decides to fetch the object, it does so by invoking an operation on the underlying system, such as opening an AFS file or fetching a WWW document. Since these operations may block, the prefetching mechanism is multithreaded to avoid blocking the application on an expensive operation.

A warden is a type-specific client to some distributed service mounted in the file system name space. If the prefix of an object's name matches the mount point of a warden, operations on that object are sent to the warden. To allow querying of wardens, SETS extends the standard low level file system interface[20] with the two calls in Figure 5. When expanding a type-specific query, SETS calls `sets_lookup`, passing in the mount point used to locate the warden and the query. The warden asynchronously runs the query and determines the names of the set members. SETS then uses `sets_expand` to retrieve the object names. `sets_expand` returns as many names as possible, but will block if none are available.

It is important to note that both the interface between the API layer and the prefetching mechanism, and the interface between the prefetching mechanism and the wardens are asynchronous. This allows the work at each layer to be overlapped. More importantly, it allows each layer to determine for itself how aggressively it should behave. Thus modularity is carefully preserved without sacrificing the dynamic nature of SETS.

3. Implementation

The goal of the implementation of SETS is to provide a general mechanism for prefetching and a single point of control for resource management. In addition to the SETS mechanism, the implementation consists of several wardens and a browser that has been modified to provide a set interface to the user. This section discusses the aspects of the implementation that affect the experiments presented later in this paper.

A key implementation decision was to place SETS in the operating system kernel. In a typical Unix environment, resource management and name resolution are performed by the operating system. Placing SETS in the kernel minimizes the cost of these activities, both of which are central to SETS. In addition, this allows SETS to easily monitor and control the resource usage of many competing applications. For instance, it is relatively easy to overutilize a network connection with only 9600 bps bandwidth. Careful scheduling is necessary to ensure reasonable performance.

The implementation makes two important assumptions. First, like AFS[21], SETS assumes that all clients have access to a local disk. Second, it assumes that one can read data off the local disk faster than one can fetch it from the server. For typical cases, this is reasonable. WWW servers are often overutilized, the latency of accessing distant data is often very high, and it is likely that the operation will require a disk read at the server. Based on these assumptions, SETS prefetches set members to the local disk. This provides a large local spooling area to hold prefetched objects, and simplifies the implementation. However, this raises the issue of overrunning the in-kernel buffer cache with prefetch traffic, and harming the performance of other applications on the same machine.

3.1. SETS Wardens

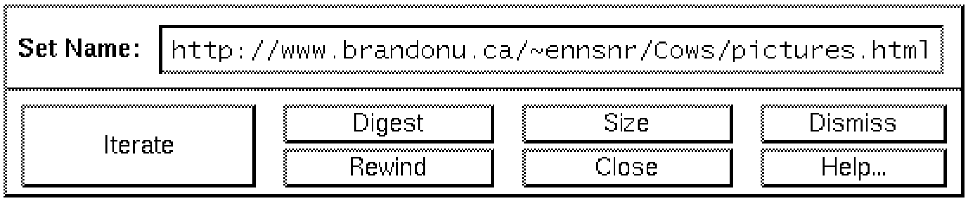
As described in Section 2.3, a warden is a client of a distributed system which allows SETS to run queries or access data in that system. I have implemented two wardens, one that allows SETS to run queries on databases, and one that allows SETS to access information in the WWW. Both are implemented as user-level processes, with SETS making upcalls to invoke operations.

The SQL warden allows SETS to run SQL queries on Informix databases[8]. The mapping between SETS and database operations is straightforward: `sets_lookup` is equivalent to opening a cursor and `sets_expand` is equivalent to moving the cursor forward[6]. The SQL warden uses information stored in the mount point to direct the query to the appropriate database. The database is assumed to have a field that holds file names. If a type-specific set specification is a valid SQL query that selects this field, the query will produce a valid set. If not, the query will result in the empty set. The warden only provides access to the database, it does not ensure consistency of the database contents with the file system.

The WWW warden provides operations to fetch WWW objects and to run "queries". A query is the name of a WWW object (URL) of an HTML document. The resulting set consists of objects for which the document contains hypertext links. The WWW warden only supports *http* and *file* URLs currently, but could be easily extended to support others.

Treating HTML documents as queries is both intuitive and useful. Most search engines on the WWW return the results of queries as HTML documents. Thus this mechanism allows one to easily query a search engine like the WebCrawler using SETS, and then peruse the results with `setIterate` or `setDigest`. In addition, dynamic sets are very useful for browsing documents that are in essence collections. For example, 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.

3.2. NCSA Mosaic as a SETS application



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. "Rewind" reinitializes the iterator, "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..." produces a window containing a brief background on dynamic sets.

Figure 6: Mosaic window for managing open sets.

I have modified several Unix utilities to use dynamic sets, but the most interesting SETS application is a modified version of NCSA Mosaic version 2.4. Mosaic is a browsing tool for the WWW, and can be used to access information in other systems as well. 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 a *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 creates and returns an HTML document that contains hypertext links to the WWW objects that satisfied the query.

The modified version of Mosaic adds the ability to open and manage sets to user interface. One opens a set either by clicking the *right* mouse button on the anchor of an HTML document, by selecting a sets dialog box and typing in the set's specification, or by requesting that Mosaic automatically create a set to hold the results of a submitted form. When a set is opened, Mosaic displays a dialog box (Figure 6) containing a number of buttons which provide a way for users to invoke set operations. For instance, clicking on the `iterate` button causes Mosaic to load and display the next element in the set.

3.3. Controlling prefetching

SETS provides an excellent opportunity to explore prefetching strategies and dynamic resource management in distributed systems, but to date I have only explored two static approaches. I was initially interested in controlling utilization of the network by SETS. However, early experiments indicated that other resources could greatly affect an application's performance. In particular, if SETS prefetched too much data, it could overrun the client's in-kernel buffer cache, forcing demand fetches to go to disk. This was acceptable for typical accesses to the WWW, since the latency of remote access is much larger than the latency of a disk read. However, in cases where prefetching was offering only a small performance advantage, SETS was actually hurting the application's performance.

SETS has two prefetching strategies. The aggressive strategy is to limit the number of outstanding fetch requests at any one time. This provides better network utilization, but may potentially overrun the buffer cache. The conservative strategy is to limit the number of files SETS can prefetch ahead of the iterator. This helps avoid overruning the buffer cache, but reduces potential performance improvements. For example, after the first n files have been fetched, only 1 prefetch per call to `setIterate` can be issued. This substantially limits SETS ability to exploit parallelism between servers.

Currently, the system does not attempt to infer which strategy is more appropriate. This is left for future research. Instead, a user may toggle between the two strategies when desired. In practice, sets and set members tend to be small, and the aggressive strategy usually performs well.

4. Evaluation

The goal of the evaluation is to answer the following two questions: "What are the performance benefits of SETS?" and "Are these benefits realizable in everyday usage of wide area systems?" Unfortunately, these questions are difficult to answer. The time to process a set of elements is directly influenced by the size of the set, the size of the members, the location of the members with respect to the client, the load on the network and servers, etc. Thus experiments that reflect real usage may not be repeatable, since many variables cannot be controlled. On the other hand, the results of experiments run in a controlled environment (low server utilization, low network latency) may not accurately reflect typical performance.

In order to address these issues, this section presents the results of two experiments. The first experiment consists of several tests run on real data in the WWW and AFS. The tests were chosen to represent typical search activity, and so the results should be reasonable indicators of how SETS would perform under normal usage. The second experiment consists of a number of microbenchmarks which characterize the impact of various factors on the performance of SETS. The microbenchmarks were run on an isolated local area network on lightly loaded servers, and so the benefit from SETS in these tests should underestimate the benefit that users would observe.

Each of the experiments below capture the time to process a set of objects with and without the use of dynamic sets. The times are presented in seconds, and are the average of three independent trials with standard deviations in parenthesis.

Weekend	What's Cool		WebCrawler		`grep` on AFS	
w/ SETS	15.30	(2.8)	126.09	(1.8)	25.47 (2.4)	
w/o SETS	133.20	(30.2)	299.87	(46.1)	35.83 (1.3)	
Speedup	8.71		2.38		1.41	
Weekday	What's Cool		WebCrawler		`grep` on AFS	
w/ SETS	40.40	(16.2)	84.48	(6.0)	65.37	(7.7)
w/o SETS	133.68	(41.6)	199.49	(50.0)	103.40	(7.13)
Speedup	3.31		2.36		1.58	

This table presents the observed latency to open and process the objects referenced during three searches on the WWW and AFS. Each point is the average of three trials, with the standard deviations in parenthesis. The first test loads and displays the set of objects referenced by the "Yahoo What's Cool" page. The second test queries the WebCrawler for documents containing the word "cows", and loads and displays the results. The third test runs the Unix application `grep` on the Mosaic sources stored at a remote AFS site. Speedup is the ratio of the times without SETS to that with SETS. The significantly reduced latencies with SETS are due to the benefits of prefetching and reordering discussed in Section 1.1. Due to the high variance in Internet load, the results of the test during relatively low (weekend) and high (weekday) network utilization are reported.

Table 1: Aggregate latency for three searches on the WWW and AFS.

Except where noted, the client is the modified version of NCSA Mosaic (see Section 3.2) on an otherwise idle DECStation 5000/200 (25MHz MIPS R3000A) with 64MB of memory running the Mach 2.6 operating system.

To control what and when objects were fetched, I instrumented Mosaic to trace object references, and added a facility to replay traces. To provide a way of capturing user processing time, the trace replay facility can be instructed to pause for a specified number of milliseconds between each trace reference. This pause time is not captured in the reported results, since the goal is to measure the aggregate time a user is forced to wait on the next object. The traces of normal Mosaic operation contained a load of an HTML document and a load for each data file it referenced. Traces of Mosaic using SETS contained a `setOpen` of the HTML document, and a call to `setIterate` for each data file in the set.

4.1. Searching on world wide information systems

The first experiment consisted of three tests. Each test was chosen to be representative of common search activity. Since the utilization of the Internet changes drastically over time[2], each of the tests was repeated twice: once on a weekend day, and once on a weekday. SETS does not currently prefetch inlined images, and so Mosaic's automatic loading of inlined images was disabled during the tests for fair comparison.

Table 1 contains the aggregate number of seconds to load the data used by the searches. For Mosaic, this is the sum of the times to load and display the next set member. For grep, this is the elapsed run time. As expected, the variance between runs is substantial, but it is clear that SETS offers a substantial opportunity for increased performance.

The first test involved fetching all the objects referenced by the "Yahoo What's Cool" page[5]. Pages that are in effect collections of related objects are quite common on the WWW, and are frequently useful when searching. At the time the tests were run, the "What's Cool" page referenced 36 objects, which totaled 140KB in size.

The second test involved querying the WebCrawler for documents containing the word "cows". Search engines like the WebCrawler are commonly used to find information in the WWW. In response to a query, they return an HTML document with links to those objects that satisfied the query. The query returned the URLs of 26 objects (including four "administrative" objects: pages for the sponsors of the WebCrawler). The 26 documents totaled 60KB.

The third test involved searching for a variable name in the sources of the modified version of Mosaic used in the previous

[5] http://akebono.stanford.edu/yahoo/Entertainment/COOL_links/

tests. This test used a common Unix search technique: "`grep cows */*.c`". Two versions of `grep` were used: an unmodified binary for the test without SETS, and a modified version of `grep` similar to the example in Figure 3. The two versions did the same processing per file, the only difference between them is the use of SETS. The source files were stored at a remote site[6] and accessed via unmodified AFS. All related files were flushed from the client's cache between trials. In both cases, `grep` read 109 files totalling 2MB in size.

The results of this experiment show conclusively that dynamic sets offer a substantial opportunity for increasing the performance of search. SETS caused a speedup of 40% to 870%. Although much of this speedup is due to prefetching, SETS ability to reorder requests played a key role as well. For instance, in the second test attempts to fetch the file `http://library2.dfp.csiro.au/cows.htm` consistently timed out. Given its position in the middle of the set, prefetching alone could only overlap 22 seconds. But by allowing other members to be yielded before it, SETS was able to reduce the apparent latency by almost 70 out of 75 seconds. Note that if in real use the searcher were to take the time to look at a member before requesting the next one, SETS may be able to deliver even better speedups. For instance, when fetching the set used in the second test with a 5 second inter-request pause time, SETS took on average only 23.88 seconds (with a variance of 2.5) to fetch and display the objects.

		Number of set members				
		4	8	16	32	64
w/ SETS	$fetch$	2.120 (.107)	2.645 (.260)	3.334 (.016)	5.636 (.442)	9.377 (.887)
	$total$	4.375 (.051)	5.508 (.168)	7.232.3 (.127)	11.957 (.336)	18.279 (3.145)
w/o SETS	$fetch$	3.456 (.023)	5.351 (.005)	9.143 (.019)	16.603 (.070)	30.082 (.432)
	$total$	5.616 (.018)	7.975 (.004)	12.706 (.019)	21.955 (.093)	38.776 (.478)

This table presents the time for the client to fetch ($fetch$) and to fetch and display ($total$) a set of data files. In each trial, the sum of the sizes of the members was constant (128KB), but the number of members was doubled for successive trials. The times are reported in seconds, with standard deviations in parenthesis. Surprisingly, the cost of opening a file is a significant factor in overall performance. It is likely that this is due to inefficiencies in NCSA's httpd and Mosaic. However, this test also shows the potential benefit of prefetching, as SETS achieves a speedup in $total$ of 1.28 for 4 members to 2.1 for 64 members.

Table 2: Performance of SETS vs the number of set members

	With SETS		Without SETS	
Size of members	$fetch$	$total$	$fetch$	$total$
2KB	1.672 (.051)	3.712 (.071)	4.696 (.053)	6.819 (.078)
4KB	1.791 (.032)	4.158 (.056)	4.966 (.069)	7.345 (.072)
8KB	2.130 (.058)	5.031 (.036)	5.239 (.073)	8.159 (.066)
16KB	2.400 (.035)	6.506 (.047)	5.758 (.100)	9.719 (.125)
32KB	3.352 (.238)	9.477 (.172)	6.691 (.072)	12.650 (.062)
64KB	5.691 (.317)	16.223 (.342)	8.760 (.310)	18.784 (.322)
128KB	6.866 (.460)	26.244 (.564)	12.584 (.119)	30.709 (.131)
256KB	8.200 (.614)	45.165 (.915)	20.770 (.310)	55.042 (.309)
512KB	13.433 (2.112)	84.920 (2.508)	38.730 (.611)	105.228 (.605)

This table presents the time for the client to fetch ($fetch$) and to fetch and display ($total$) a set of 10 objects, the objects' size is doubled for successive tests. The times are reported in seconds with standard deviations in parenthesis. As expected, the numbers show that the aggregate time to fetch and process a set of objects is linear in the total number of bytes. Although fetching and displaying members with SETS is 20% faster than without for 512KB files, the speedup for smaller files approaches a factor of 2. Thus users of dynamic sets should see reasonable performance improvements, since the distribution of file sizes in the WWW seem to be skewed towards small files[5].

Table 3: Performance of SETS vs the size of the set members

[6] roughly 600 miles by car, 17 hops on the network

		Inter-request pause time (seconds)					
		0		5		10	
w/ SETS	$fetch$	23.4	(0.57)	9.0	(1.03)	7.0	(0.04)
	$total$	31.1	(0.61)	16.8	(1.32)	11.9	(0.29)
w/o SETS	$fetch$	57.7	(6.25)	60.0	(1.45)	55.7	(1.58)
	$total$	62.5	(6.34)	65.2	(1.40)	60.6	(1.47)

This table presents the time for the client to fetch ($fetch$) and to fetch and display ($total$) a set of ten 2KB objects over a 9600 bps SLIP link. Successive tests increased the inter-request pause times to see if the benefits claimed in the second example in Section 1.1 were realizable. The pause times reflect the time that a user would take to read the current document before moving on to the next one. This test used a different client machine than the one used by the other tests in order to access SLIP. This client's event timer was only accurate to 15 milliseconds, and so the numbers in the table are shown with less precision.

Table 4: Performance of SETS on SLIP vs inter-request pause time.

4.2. Microbenchmarks

The experiment in the previous section demonstrates the enormous performance improvements achievable by SETS, but since so many factors can influence the results, it does not give a good understanding of how SETS achieves these benefits. This section addresses this shortcoming by presenting the performance of SETS for three microbenchmarks. Each microbenchmark examined a different factor in the performance of SETS: the number of elements in the set, the size of the objects in the set, and the performance of the network. The tests used an unmodified copy of `httpd` version 1.3 from NCSA as the server on a 66 MHz i486 DX2; the client was the modified version of NCSA Mosaic discussed in Section 3.2. Except where noted, the server and client machines were connected by an isolated 10Base2 Ethernet, and no inter-request pause times were used.

Each trial consisted of fetching and displaying the set of data files referenced by an HTML document. Each trial's results consist of the sum of the times to fetch an object ($fetch$) and the sum of the time to both fetch and display an object ($total$) over all objects in the trial. The data files did not contain any HTML tags in order to reduce the display times. Neither Mosaic nor the URL warden cached files, but prefetched data tended to remain in the client machine's in-kernel buffer cache. In fact, the conservative prefetching strategy was chosen to avoid overrunning the buffer cache with prefetched data. Since the servers were lightly loaded, requests for the data files tended to hit in the server's buffer cache.

Table 2 presents the results from the first test. In this test, the sum of the file sizes in each trial was constant (128KB), but the number of data files varied from 4 to 64. Surprisingly, the cost to open a file had significant impact on the overall performance. It is likely that this is due to inefficiencies in the implementation of Mosaic and httpd, since opening a file involves setting up a TCP connection and forking a new process on the server to handle the operation. This test shows the benefits of the SETS implementation, since the high open costs are overlapped with application processing time. In fact, SETS achieves a speedup of 1.28 to 2.1, and the speedup increases with the number of set members.

Table 3 shows the results of the second test, in which each trial processed a set of 10 members, but the size of the data files was doubled for each successive trial. As one would expect, the time to process the data grows linearly with the size of the data. However, the benefit of prefetching in this case is lower than one would expect from the previous experiment.

There are several reasons why SETS only offers a modest benefit. First, for this test the cost of remote access is minimal. A demand load fetches data from a lightly loaded server (most likely from its buffer cache) over a lightly loaded network directly into memory, and is not significantly longer than the time to read data from the client machines's buffer cache (assuming all files have been prefetched). Second, the network latencies are so low that $total$ is dominated by the display time. Since SETS can only reduce I/O latencies, its impact on $total$ is diminished. The speedups observed were 1.8 for 2KB files to 1.2 for 512KB files and are much lower than the speedups seen in the experiments on the WWW. This difference is probably due to the much higher latencies and the greater potential parallelism of the previous experiment. The reason for the reduction in speedup for larger files is that the time to display files grows faster than the benefit from SETS: the magnitude of the difference in $total$ for 512KB objects (20 seconds) is much larger than the difference for 2KB objects (3 seconds).

Table 4 shows the results of the third experiment, fetching a set of 10 2KB files over SLIP (serial line IP) on a 9600 bps phone line. Unfortunately, the client machine used in the other tests could not directly access a modem, and so this test uses a 25 Mhz i486SL portable running the Mach 2.6 operating system instead. The event timer for this machine is only accurate to 15 milliseconds, so the results of this test are less precise. The relatively high variance in the results may be due to noise in the phone line: this test saw a substantially higher number of retries than seen by the tests on Ethernet.

The results show that SETS can indeed provide benefit even in this extreme case. When no inter-request pause times are present (the user is clicking the "iterate" without bothering to look at the newly displayed document), SETS still provides some benefit. As the user spends more time reading a document, SETS can prefetch more information, thus reducing the latency apparent to the user. Note that SETS achieves two benefits in this case: the overlap of processing and I/O, and more efficient utilization of the SLIP line. In fact, it took the URL warden roughly 25% less time to fetch a 2KB file than it took Mosaic, although both fetch data in a similar manner.

5. Related Work

The idea of prefetching data is certainly not new. People have used it to improve performance in database systems[26], to increase the bandwidth of sequential reads in the Unix fast file system[16], and to increase availability in distributed file systems[9], to name just a few uses.

Two forms of prefetching exist: inferential and informed. In the former, the system bases predictions on observations of past references. For instance, Korner[10] proposes off-line analysis of file traces to establish rules for mapping file types to access patterns, and using the rules as hints to drive the preloading of data into the buffer cache. Kotz and Ellis[11] propose on-line detection of sequential access to "portions" of a file to drive prefetching in scientific workloads on a MIMD computer. They note that incorrect inferences can cause prefetching to significantly degrade the application's performance. Tait and Duchamp[27] and Kuenning[12] propose inferential prefetching to increase performance and data availability for mobile, resource poor clients of distributed file systems. Unfortunately, search tends to access large numbers of (recently) unaccessed objects, and so past references may have little bearing on the requests to be made by the current search. Thus inferential techniques are not well suited to supporting search in wide area systems.

As opposed to inferential prefetching, informed prefetching bases decisions to prefetch information on application supplied *hints*. SETS are an example of informed prefetching: by calling `setOpen`, an application is in effect giving the system a hint that each file in the set is likely to be referenced shortly. Another example of informed prefetching is Transparent Informed Prefetching (TIP)[17]. In TIP, an application supplies lists of subsequences of a file that it plans to access to the system. The system then uses these hints to prefetch blocks from a parallel disk array into memory. Although TIP works well with a variety of intra-file access patterns, most search accesses are sequential, and would not benefit from TIP's generality. Another example of informed prefetching is the ELFS system[7]. ELFS is an object-oriented framework that allows application writers to derive their own file system from a base class. Operations in a derived class could use semantic knowledge to pass hints to lower levels. For instance, a "2D-matrix" file class operation could inform the system that the file will be traversed in column-major form, allowing the system to appropriately prefetch blocks. Although both of these systems employ informed prefetching, neither provide the reordering allowed by SETS. As mentioned earlier, reordering can have substantial effect when the average time to fetch members has a large variance. However, one could combine TIP and SETS to provide two stage prefetching. For example, SETS could prefetch data to the local disk, and TIP could prefetch it from the local disk to the buffer cache just before the element is yielded to the application. This would allow SETS to be more aggressive without overruning the client buffer cache.

Along with the growth in popularity of wide area systems has come an increased interest in information retrieval systems. An early example is the Semantic File System[4], which automatically builds indexes of file system objects, and allows users to retrieve files that satisfy a conjunctive query over a space of name-value attribute pairs. Another example is the Rufus system[23], which provides both a file system and a database view of the information. In addition to supporting queries, the database view allows one to attach methods to file types, so one could have a "print" command which automatically converts a latex file to postscript before printing. The two views are kept consistent by periodic updates. In neither system, however, is the latency to fetch objects returned by a query addressed. Although both are implmented in local area distributed systems, it is likely that they could benefit from the addition of SETS. Further, search in either system would be enhanced by dynamic set's programming model.

There are many examples of search engines on the WWW. I have already mentioned two, the WebCrawler[18] and Lycos[14]. Others can be found by loading the HTML document `http://home.mcom.com/home/internet_search.html`. As mentioned in Section 1.1, search engines such as these only solve part of the problem of search in world wide systems; the problem of identifying candidate objects. SETS addresses the separate issue of reducing the latency to access the candidate objects. The popularity of querying search engines increases the usefulness of SETS, since dynamic sets are ideally suited to support this form of search. The results of the experiment discussed in Section 4.1 convincingly demonstrate the performance advantages of using SETS to fetch the result sets of search engine queries.

6. Conclusion

As put forth in the introduction, dynamic sets offer substantial opportunity to reduce the aggregate latency to fetch groups of objects. Search, an important application for world wide information systems, is essentially a process of identifying and examining such groups. When this grouping is disclosed to the system through creation of a dynamic set, the system can use prefetching and reordering of requests to speed the search. Sample searches on existing wide area systems have show substantial increases in performance from using dynamic sets, as much as a factor of 8.

SETS has been fully implemented as an extension to the Mach 2.6 operating system. Wardens that support access to a portion of the WWW and querying of SQL databases have been implemented, and support for sets has been added to the popular NCSA Mosaic browser. The only aspect of SETS not fully functional at the time of this writing is support for executable set specifications. Ports of SETS to the NetBSD 4.4, Linux, and OSF/1 operating systems are underway.

Several extensions would enhance the functionality of SETS. First, providing richer summaries as discussed in Section 2.1.2 would allow users to efficiently search through more types of data. Second, an operation which forces SETS to finish evaluating set membership may simplify some programming tasks. Currently, SETS only provides polling (via `setSize` for instance). Third, providing a means to assign weights to members would extend the usefulness of SETS to those applications which require (weak) ordering constraints.

One aspect of dynamic sets that bears further investigation is the need for automatic "filters" in search. Many search engines provide limited indexing. For instance, if one wished to locate documents that contained a particular phrase, one could use grep as a filter to documents returned by Lycos (which only matches individual words). One approach would be to add a set operator to run some filter on every member of the set, similar to the `foreach` construct in the `csh`. An advantage to this approach is that the system could ship the filter to the data, and only fetch objects that satisfied the filter. This could have significant effect when performance is bandwidth limited, such as using SLIP.

References

[1] T. Berners-Lee, R Cailliau, A. Luotonen, H.F. Nielsen, and A. Secret. The World Wide Web. *Communications of the ACM*, 37(8), August 1994.

[2] K. Claffy, H. Braun, and G. Polyzos. Tracking long-term growth of the NSFNET. *Communications of the ACM*, 37(8), August 1994.

[3] H. Garcia-Molina and G. Wiederhold. Read-only transactions in a distributed database. *ACM Transactions on Database Systems*, 7(2), June 1982.

[4] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, Jr. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[5] S. Glassman. A caching relay for the world wide web. *Computer Networks and ISDN Systems*, 27(2), November 1994. Special Issue: selected papers from the First International WWW Conference.

[6] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*, pages 36–37. The Morgan Kaufmann series in data management. Morgan Kaufmann Publishers, Inc, 1993.

[7] A. S. Grimshaw and Jr. Loyot, E. C. ELFS: Object-oriented extensible file systems. Technical Report TR-91-14, Computer Science Department, University of Virginia, July 1991.

[8] Relational Database Systems Inc. Informix. 4 100 Bohannon Drive Menlo Park, CA 94025.

[9] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[10] K. Korner. Intelligent caching for remote file service. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, 1990.

[11] D. Kotz and C. Ellis. Practical prefetching techniques for parallel file systems. In *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems*, Miami Beach, Florida, December 1992.

[12] G. H. Kuenning. The design of the SEER predictive caching system. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, December 1994.

[13] B. Liskov and J.V. Guttag. *Abstraction and Specification in Program Development*. The MIT EECS Series. MIT Press, Cambridge, MA ; McGraw-Hill, New York, 1986.

[14] M. Mauldin and J. Leavitt. Web-agent related research at the CMT. In *Proceedings of the ACM Special Interest Group on Networked Information Discovery and Retrieval (SIGNIDR-94)*, August 1994. Also available as http://fuzine.mt.cs.cmu.edu/mlm/signidr94.html.

[15] M. McCahill. The Internet Gopher: A distributed server information system. *ConneXions – The Interoperability Report*, 6(7), July 1992.

[16] M. McKusick, K. Joy, W. Leffler, and R. Fabry. A fast file system for Unix. *ACM Transactions on Computer Systems*, 2(3), August 1984.

[17] R. H. Patterson and G. A. Gibson. Exposing I/O concurrency with informed prefetching. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems, Austin, TX*, September 1994.

[18] B. Pinkerton. Finding What People Want: Experiences with the WebCrawler. In *Proceedings of the Second International WWW Conference: Mosaic and the Web*, September 1994. Also available as http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/WWW2_Proceedings.html.

[19] J. Postel and J. Reynolds. File transfer protocol (FTP). Network Working Group Request for Comments (RFC) 959, ISI, October 1985. Available as http://info.cern.ch/hypertext/WWW/Protocols/rfc959/Overview.html.

[20] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network File System. In *Summer Usenix Conference Proceedings, Portland*, 1985.

[21] M. Satyanarayanan, J. Howard, D. Nichols, R. Sidebotham, A. Spector, and M. West. The ITC Distributed File System: Principles and design. In *Proceedings of the 10th ACM Symposium on Operating System Principles, Orcas Island*, December 1985.

[22] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4), April 1990.

[23] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, and J. Thomas. The Rufus system: Information organization for semi-structured data. In *Proceedings of the 19th International Conference on Very Large Data Bases*, Dublin, Ireland, August 1993.

[24] A. J. Smith. Disk cache – miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3), August 1985.

[25] M. Spasojevic and M. Satyanarayanan. A usage profile and evaluation of a wide-area distributed file system. In *Winter Usenix Conference Proceedings*, San Francisco, CA, 1994.

[26] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7), July 1981.

[27] C. D. Tait and D. Duchamp. Detection and exploitation of file working sets. In *Proceedings of the 11th International Conference on Distributed Com puting Systems*, Arlington, TX, 1991.

[28] J. Wing and D. Steere. 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.