March 1997

Exploiting non-determinism in set iterators to reduce I

David C. Steere

Exploiting Non-Determinism in Set Iterators to Reduce I/O Latency

David C. Steere

March 26, 1997

Abstract

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. We propose a new file system abstraction called dynamic sets that allows the system to transparently reduce I/O latency without relying on reference locality, without modifying DFS servers and protocols, and without unduly complicating the programming model. We present this abstraction, and describe an implementation of it that runs on local and distributed file systems, as well as the World Wide Web. Substantial performance gains are demonstrated – up to 50% savings in runtime for search on NFS, and up to 90% reduction in I/O latency for Web searches.

1 Introduction

A central problem facing distributed systems is the high latency to access 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 reduce the usability of a system, especially if the variance in the delay is high. In this paper we show that a small, carefully designed extension to the system-call interface of an operating system can result in a substantial reduction in the aggregate I/O latency seen by applications that use iterators, without requiring locality of reference or modifications to protocols or servers.

The essence of our argument is that *extending the system interface to support iterators will allow the system to reduce I/O latency transparently for those applications that use them*. Our solution is based on three observations. First, current file system interfaces restrict the system's opportunity to reduce latency by forcing applications to process groups of objects in a serial, and often imposed order. As a result, systems manage I/O for applications without accurate knowledge of their future data needs, yet pushing I/O management to the application significantly increases the complexity of the programming model. Second, iterators are a convenient mechanism for processing groups of objects, as attested by the widespread use of iterator-like constructs such as cursors in SQL; foreach loops in shells like perl, tcl, and sh; and iterators in higher level languages like Alphard[26] and CLU[17]. Third, the use of iterators on sets of objects could allow a system to transparently reduce the aggregate I/O latency of accessing the set members if the iterator was visible to the system.

To explore the utility of iterators, we have added a new abstraction called *dynamic sets* to the application programmer interface (API) of a distributed file system (DFS). 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 that it wishes to process. An object's membership in a dynamic set indicates the likelihood of near-term access, allowing the system to safely prefetch the objects' data to reduce latency.

An application creates a dynamic set by supplying a membership specification that is evaluated by the system to ascertain 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 that has already been fetched. As a result, the application sees either little or no latency to access the object's data. Applications can also manipulate set membership using standard set operations. 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.

A crucial aspect of this work is that the application's use of iterators on unordered sets frees the system to determine the order in which it yields objects to the application. There is currently no way for an application to express such non-determinism to the kernel, which forces determinism on applications and restricts the system's opportunity to reduce latency. Dynamic sets provides a means for applications to disclose non-determinism to the system, allowing the system to schedule I/O and manage its

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.

caches more efficiently. Dynamic sets do allow applications to request an order, but applications that do so limit the system's ability to optimize access through reordering and may pay a performance penalty for their ordering requirements.

One application domain that can benefit from dynamic sets is search and retrieval of file data, hereafter referred to as search. Search applications identify a group of candidate objects, and then fetch and examine them in turn to find an object or objects that satisfy the search criteria. Search has several important characteristics that make it an ideal application for dynamic sets. First, search is an important application as is dramatically apparent to any user of a large distributed system. Second, search exhibits poor locality of reference and thus gets little benefit from caches. For example, one study of a World Wide Web (Web) caching proxy saw only a 33% hit rate, even though the cache had unlimited size and served all external references from employees of a large computer company[8]. Third, searches often run until some satisfactory object is found, and thus either have no preference of the order in which objects are fetched or have insufficient knowledge to specify the order at the time of search.

2 Related Work

I/O latency has long plagued computer systems, and system builders have developed two basic techniques to overcome it: caching and prefetching. Caching is widely used, and is nearly ubiquitous in distributed file systems[10, 24, 20] in which accessing remote data incurs high latency. However, caching is effective only if applications exhibit locality of reference. Prefetching does not rely on locality and so is more suited to applications with poor locality like search. The drawbacks of prefetching are that one must somehow predict future data accesses in order to prefetch them, and inaccurate predictions increase the load on the I/O subsystem, and can lead to thrashing. Systems that infer future accesses based on past history[16, 6, 31, 22, 9] are most susceptible to this problem. One study found a 20x slow down in one case when prefetching data from disk on a parallel computer[15]. However, prefetching can produce substantial improvement if the access pattern is sufficiently regular and easily detected, such as Unix's one-block read-ahead mechanism[1, 27].

One way to avoid the problem of inaccurate predictions is to expose asynchronous I/O directly to applications, and let applications manage their I/O explicitly. However, this approach increases the burden on the application programming, and violates software engineering principles which call for hiding low-level details beneath strong interface boundaries. 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. An example of explicit prefetching is the Queued RPC mechanism of the Rover toolkit[11], which exposes asynchrony to application programmers and users. Although this can result in more efficient I/O, it requires the application programmer to poll to determine when an operation has completed and to maintain the operation's context until the operation terminates.

In another approach, called *Informed Prefetching*, the application informs the system of its future data needs but leaves the management of asynchrony to the system. The system can safely prefetch based on these hints, and the application is not complicated by the need to control prefetching or manage system resources. Recent studies by Patterson et al[23], Cao et al[4, 5], and Kimbrel et al[13] have found significant speedups from informed prefetching in local file systems, particularly when reading data from multiple disks in parallel. These systems require application programmers to manually augment their code to pass hints of future block accesses to the file system. Mowry et al[19] describe a similar approach which uses compiler generated hints to pre-page in a virtual memory system[19]. Their compiler generates prefetch requests by analyzing program loops to determine near-future data accesses in virtual memory. Similar analysis allows the compiler to insert hints to release pages as well.

The work described here also uses informed prefetching to reduce latency, but differs in several respects. First, the hints of future access are derived from the membership of a dynamic set as opposed to being supplied by the compiler or application programmer. Second, dynamic sets offer the opportunity to schedule file accesses more efficiently through reordering. Third, the implementation of dynamic sets is tuned for search on distributed file systems and prefetches whole files, as opposed to prefetching blocks within a file.

3 Dynamic Sets

To better understand how applications could use dynamic sets, consider a search using the Unix command `grep`, such as "`grep pattern *.c`". Currently, the shell expands the wildcard "`*.c`" into an alphabetical list of filenames, and `grep` opens each of these files in that order. For each file, `grep` reads the file's data and prints lines matching the pattern. Although `grep` knows the identity of the files it will read when it starts, it has no way of disclosing this information to the system, and thus the system has no opportunity to prefetch the files. In addition, the order in which the files are opened is imposed by the shell, independent of `grep`'s, the system's, or the user's needs.

Now consider how `grep` might use dynamic sets. First, `grep` would create a dynamic set to hold the files named

Main loop of `grep`	Main loop using dynamic sets

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

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`. The code on the right shows the main loop of `grep` using dynamic sets. This example illustrates two points. 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 a file, does not need to be modified to use dynamic sets. In this example, the command line argument must be quoted to prevent shell expansion of "`*.c`".

Figure 1: Code Example Showing the Use of Dynamic Sets

by "`*.c`". `Grep` would then loop, calling the iterator to retrieve the next file, and processing it using the same code as standard `grep`. Each call to the iterator returns a previously unseen file, and the loop terminates when all set members have been seen and processed. Figure 1 contains the main loop of `grep` with and without dynamic sets.

Modifying `grep` to use dynamic sets yields three benefits. First, the system can prefetch the files named by "`*.c`" with a reasonable assurance that `grep` will shortly 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, 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. Third, prefetching and reordering together give the system greater flexibility to adapt its behavior to changing resources. For instance, the system might prefetch all of a set when communicating with a lightly loaded server, but may only prefetch one or two members on a low bandwidth or loaded connection. In addition, the system can chose to prefetch only some of the members to avoid wasting I/O bandwidth should the search terminates prematurely.

3.1 Properties of Dynamic Sets

We designed the dynamic set abstraction to be general, since it offers benefit to any application that can iterate, that suffers substantial I/O latency, and that can inexpensively name the objects in its short-term working set. In particular, the design includes only that functionality necessary to support search applications in order to avoid overrestricting the implementation's ability to reduce latency. The following paragraphs describe the dynamic sets abstraction.

- **Created on-demand**

  Applications create dynamic sets on-demand by supplying a specification that the system evaluates to determine the names of the set members. The membership specification language is orthogonal to the design of sets, and is discussed below. Because this specification is evaluated at runtime, a set's membership depends on the state of the system at the time of the set's creation, hence the moniker "dynamic". One advantage of determining membership at runtime is that applications see current information by default, but can relax currency by opening a set before it is needed. Fortunately, evaluating membership consists of name resolution, e.g. Unix filename *globbing*, which is typically a small percentage of the time to fetch the set members' data.

- **Short-lived**

  Because the membership of each newly created set is dynamically determined, the system need only preserve a set while its creator is running. This in turn allows the system to maintain sets in volatile memory, which results in sets being lightweight. Since the time spent creating and maintaining a set directly offsets any potential benefit of prefetching and reordering, lightweight sets can reduce latency in a wider variety of settings.

- **Unordered**

  By using a set as the abstraction underlying dynamic sets, we allow applications to disclose their short-term working set without imposing a deterministic order on it. This non-determinism frees the system to schedule file access for greater efficiency. For instance, the system could yield a cached member immediately, and overlap the I/O to fetch other members with the time to process the cached member. Further, the system could prevent cached members from being evicted from the cache before the application gets a chance to read them. In addition, the ability to reorder allows the prefetcher to prefetch speculatively, rather than based on estimates of server latency. For instance, the prefetcher can initiate three fetches and use the first to return, rather than waiting to determine the size of every object and calculating the expected latency to fetch them. This is particularly important in the presence of unpredictable failures which result in lengthy timeouts.

Currently, many applications have no particular ordering needs and so could use dynamic sets, but are forced to serialize their accesses by current system APIs. Often the order is provided by some third party, such as the `csh` in the case of filename globbing. For search, the proper order is unknown until the search terminates with a satisfactory object, since that object would be first in the optimal order. Search engine rankings approximate this order, but are not sufficiently accurate to warrant strict adherence to them. For those applications that do have an ordering preference, dynamic sets allow applications to submit ordering hints, for instance based on search engine rankings. The system attempts to satisfy this order, but may violate it rather than blocking the application if a member with lower rank is available.

One can think of dynamic sets as a specific instance or component of a general $N-$tuple data structure. The order within the tuple represent a partial order on all its members. Elements within an equivalence class can be stored in a dynamic set; all members have the same rank in the partial order. A dynamic set would be a 1-tuple in which all members had equal rank. An ordered list would be an $n-$tuple where n is the number of objects in the tuple. However, we leave the design of this more general abstraction to future work.

- **Loosely consistent**

  Ideally, membership would be evaluated atomically and have perfect precision and recall (no false positives or negatives). However, it can be expensive in system complexity and performance to provide these properties[33]. Further, dynamic sets are layered on top of existing systems for simplicity, and as such cannot provide a stronger consistency model than the underlying system. Fortunately, many searches on DFS are satisfied without strong consistency guarantees, as the widespread use of these systems can attest. For example, a programmer can usually find the right version of a source file without having to lock all the candidate files and directories for the duration of the search.

  Rather than promise the illusion of atomicity, dynamic sets instead guarantee that:

  - Every member must satisfy the membership 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.

  Together, these guarantees ensure that the membership of a set is current but not necessarily complete. In addition, the state of each member captured in the set is the state that satisfied the specification, and not necessarily the most current version of that object. However, specifications that involve queries to search engines can only be as correct as the search engine's index, since the engines are external to dynamic sets.

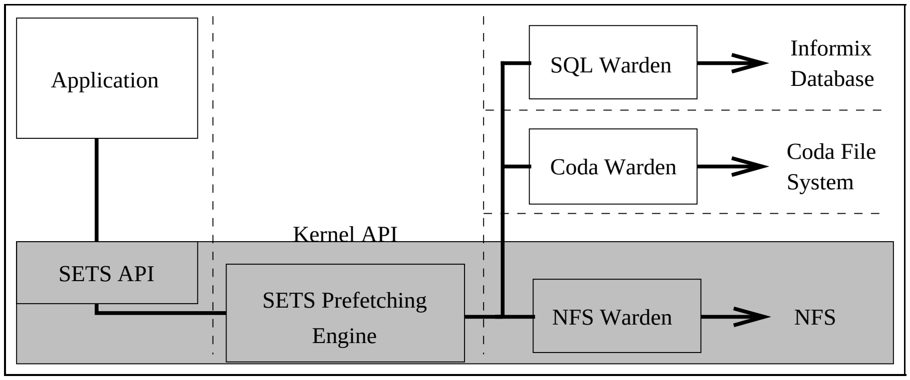
- **No duplicates, immutable**

  Dynamic sets are similar to mathematical sets in that they do not contain duplicate members and are immutable. Duplicates can be eliminated automatically by testing for name or value equivalence with other members. Using name equivalence has the added benefit of eliminating duplicates before fetching their data, while still providing reasonable semantics. To ensure immutability, operations which would otherwise modify a set's membership instead create a new set. Since sets are lightweight, the cost of immutability is small.

4 Implementation

To evaluate our design, we have added dynamic sets to the Unix file system interface. Since this implementation is just one possible use of the dynamic sets abstraction, we refer to the implementation as SETS to distinguish its features from those of the dynamic sets abstraction. Figure 2 depicts the architecture of SETS, which contains three basic components. The API component manages the dynamic sets data structures and exports the SETS API to applications. The prefetching engine evaluates membership specifications to determine the names of members and manages prefetching. Wardens are DFS clients extended to support queries and prefetching. SETS defines asynchronous interfaces between these components to avoid unnecessarily stalling the processing of a set.

One controversial aspect of this architecture is that the API and prefetcher reside in the operating system kernel, as opposed to residing in a user-level library. A kernel implementation allows SETS to interact closely with the file system at low cost. For instance, the prefetcher needs low



This figure depicts the main components of SETS: the API layer, the prefetching engine, and the wardens. The bold dashed line indicates the kernel boundary, other dashed lines separate different threads of control. The API layer extends the kernel interface with the SETS operations, the prefetching engine sits within the kernel. In the picture two wardens are outside and one is inside the kernel; the location is chosen by the implementor.

Figure 2: The Architecture of SETS

Explicit:	`/projects/*src*/*.c`
Interpreted:	`/staff/\select home from users 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 that is interpreted by search engines as queries, returning the names of the objects that satisfy the query. *Executable* specifications name executable programs whose execution results in a list of filenames. With these types of specifications, SETS can easily be extended to support a variety of query languages and modes of search.

Figure 3: Examples of SETS Membership Specification Language

latency access to the file system's buffer cache to locate cached set members and to avoid overrunning the cache with prefetch data. Note that this decision is specific to SETS: in some other domain such as a Web browser it may be more appropriate to implement dynamic sets as a plug-in or library.

4.1 Application Programming Interface

The dynamic sets API provides operations to create and destroy sets, merge sets through union or intersection, create a subset, query a set's membership, determine a set's size, list the names or properties of members, and iterate on the set. For brevity, we discuss only the representation of an open set and the membership specification language.

In SETS, an open set is similar in nature to an open file descriptor. The open set handle is an index into a per-process table of open sets. This open set handle can then be passed to set operations, in much the same way that a file descriptor is passed to the `read()` system call. When the process exits, open sets are automatically destroyed and their resources freed.

When a set is created, the creator supplies a *specification* which SETS evaluates to produce a list of the names of the set members. The specification language used by SETS extends the `csh` wildcard set notation[12] to supports three types of specifications. Figure 3 gives examples of each. *Explicit* specifications use standard `csh` wildcard notation to indicate the names of the members of the set. *Interpreted* specifications contain strings in some query language, such as SQL, delimited by "\". The query is passed to the warden responsible for the object named by the prefix of the specification, resulting in a list of names which are then used to further expand the specification. The warden that interprets the query is not necessarily responsible for the objects named by the query, for instance a GLIMPSE[18] warden could reference NFS objects. The second example in Figure 3 would cause SETS to send the SQL query to a database mounted at "`/staff`". If this object's warden did not support SQL queries or the selected fields did not contain valid filenames, the specification would result in the empty set. *Executable* specifications name programs that act as filters over a portion of the system's name space, returning the names of satisfactory files to SETS. Note that interpreted specifications use existing search engines such as SQL databases to provide functionality similar to that provided by search-enhanced file systems[7, 18, 3].

4.2 SETS Prefetching Engine

The prefetching engine consists of a number of worker threads, and is responsible for evaluating specifications and prefetching set members. The API layer generates requests on behalf of applications and queues them for workers. The workers handle the requests, possibly updating the set's data structures to reflect new members or to indicate that a member has been cached.

Workers evaluate specifications to determine the names of members, checking to see if the object is already a member, and if not adding it to the set. Workers evaluate explicit specifications directly using Unix's name resolution mechanisms. For interpreted specifications, a worker establishes a *cursor*, or query handle, with a warden and queues a request to have the cursor expanded at some later point. When this request is handled, the worker reads member names from the cursor and adds them to the set, requeueing the request for further expansion if necessary. Executable specifications are handled in a similar manner, but start a new process in which to run the command instead of contacting a warden.

4.2.1 Prefetching Policy

We designed the SETS's prefetching policy to work in an environment where remote access incurs a high latency, such as a wide-area DFS like AFS[28] or a mobile client connected over a low-bandwidth link. The policy has to balance conflicting goals: aggressive prefetching results in lower latencies, but may overwhelm disks, networks, or servers, resulting in thrashing and loss of performance. Prefetching in a DFS is complicated by variance in latency, e.g. due to load or non-uniform access times between servers. Load due to other clients of the system is difficult for a prefetcher on one client to measure or predict. Access times may vary widely between servers, due to load, geographic or network location relative to the client, or the performance of the server itself. Various caches throughout the system can also affect latency in ways that a given client cannot predict.

In order to prefetch in the face of inaccurate or incomplete knowledge of system state, we make three simplifying assumptions. First, we assume that accessing data off the local file system is faster than fetching it from a server. This is true when the data is in the local buffer cache, when the data is very remote (propagation delays and connection setup are often larger than local disk I/O in wide-area DFS), or the remote accesses miss in the server's memory cache[1]. Second, SETS assumes that local disks have a cache large enough to hold reasonably sized sets. Disks capacities have been growing exponentially, and even low-end PCs typically come with several gigabytes of free disk space. Third, SETS assumes that set members will be accessed sequentially and as whole files.

These assumptions result in several simplifications. SETS stores prefetched data in the local file system, SETS prefetches whole files opportunistically, and SETS can tune its policy to adapt to different kinds of systems. When a set is opened for iteration, SETS concurrently fetches a small number of files, spreading the requests across servers or disks if possible. The number of files initially fetched depends on the client's guess of the available bandwidth, but is currently limited to five files (based on typical file size, application processing rate, and latency). When the application calls the iterator, SETS returns the largest fully cached member which has the highest rank if the application has specified an order. On each call to the iterator, SETS starts a new prefetch, and thus automatically tunes the rate at which it prefetches files to the rate at which the application consumes them.

In addition, SETS needs to manage its consumption of the file system buffer cache to maximize the application's hite rate and to avoid overruning the cache. For instance, if several of the concurrently fetched files are larger than the buffer cache, prefetching them entirely would evict the beginning of the files from the cache along with everything else in it. Since the application will read these files sequentially, it will miss on the evicted data, evicting the next blocks to read and so on, thus missing on every block in the file. In this situation, other applications will see lower hit rates in the buffer cache as a result of prefetching.

SETS extends Unix's buffer cache management to handle buffers with prefetched data in three ways. First, SETS limits the number of buffers that can be used to hold prefetched data. This also limits SETS consumption of network bandwidth, since the prefetcher will stall when it runs out of buffers. Second, SETS pins data in the buffer cache to prevent other data from evicting it. SETS then uses the knowledge of what is pinned when decideding what objects to yield to the application. Third, SETS can proactively warm the cache with evicted data when all pinned data has been consumed by the application.

4.3 Wardens

A SETS warden is the client of a distributed system extended to support prefetching and interpreted specifications. Wardens can run in the kernel, such as the NFS warden which is based on an in-kernel NFS client, or in user-level processes. User-level wardens communicate with SETS using an existing upcall mechanism[29] which passes VFS file system operations[14] to user-level DFS clients, caching data in the kernel to avoid upcalls where

[1] When this assumption is false, SETS can adjust its buffer cache eviction policy to refetch data from the servers on demand instead of evicting it to the local disk.

possible. We extended this mechanism with operations to prefetch an object, open a cursor for an interpreted specification, expand the cursor to retrieve the resulting filenames, and close the cursor. This mechanism also allows wardens to mount themselves as virtual file systems in the local file system namespace. Wardens can implement all or part of this extended VFS interface. For instance the warden to an SQL database may chose to support only queries, while an NFS warden may support prefetching and the standard VFS operations, but not queries.

The open cursor operation passes the specification to the warden, which responds with a cursor – a handle to an as-yet empty set of names. The warden then asynchronously interprets the specification to produce a list of filenames, potentially contacting servers in the process. Expand cursor operations return any names that are currently available, or block until the warden produces some names or finishes the interpretation. The close cursor operation is necessary to allow SETS to inform wardens to prematurely terminate the cursor if the application closed the set.

The prefetch operation causes the warden to fetch an object, and blocks until the entire object is cached. Simple wardens fetch the data on demand, more complicated wardens can use asynchronous I/O or lower priority operations if their system allows. Once an object is cached, SETS holds it open to prevent the warden from evicting its data. However, the warden can chose to evict a prefetched object's data without violating the semantics of SETS as long as it prevents updates to the object to ensure consistency.

4.4 Current Status

We have implemented SETS as an extension to the file system of the Mach 2.6 operating system, a variant of 4.3BSD Unix. Although we used Mach for historical reasons, our implementation avoids Mach-specific functionality and we are in the process of porting SETS to NetBSD and Linux. The NFS warden took 3 days to implement (starting from the NFS client source code), and adds or modifies 379 (out of 6887) lines of code.

We have modified a number of Unix utilities to use dynamic sets. Although one must recompile an application to use sets, the changes are relatively simple as shown in Figure 1 and are easy to make.

5 Evaluation

Our evaluation is based on a number of synthetic benchmarks that examine the potential benefits of dynamic sets with respect to the cardinality of the set of objects being examined, the size of these objects, the degree of parallelism, and the amount of application computation. In addition, we ran two experiments to examine the effect of reordering and the benefits of dynamic sets for search on a local file system. A more complete set of experiments, including low bandwidth and interactive search tests, is described elsewhere[30].

5.0.1 Test Methodology

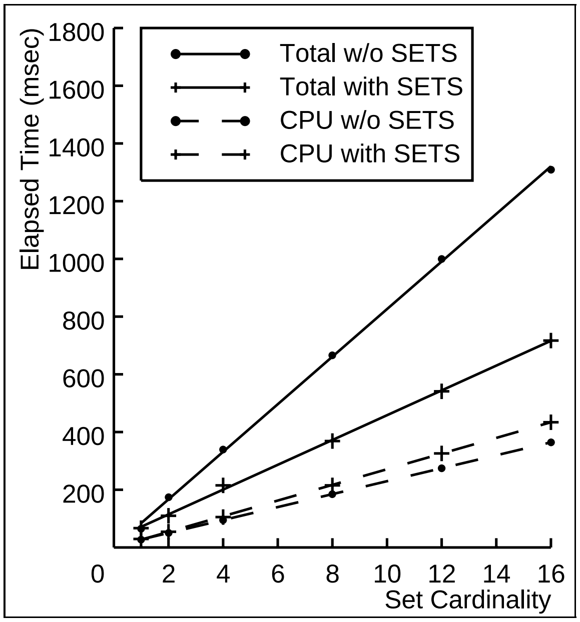
Our experiments use a benchmark program called *synthGrep* to generate a workload for the system. SynthGrep is derived from the Unix `grep` utility, preserving the I/O pattern of `grep` (whole file sequential, process a block before reading the next), but providing two parameters to control the amount of computation. 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 allows synthGrep to emulate user think time, but is not utilized by the experiments described here.

Each experiment consists of running synthGrep on a set of uncached NFS files, once using the standard file system operations and again using dynamic sets. The experiments flush both the client's and the servers' buffer caches before running synthGrep to eliminate dependencies between runs. The experiments record the total elapsed time to run the test as well as the amount of time spent in the idle loop. In the absence of competition for the client's CPU, idle time is equivalent to the amount of time the application was blocked waiting for data. The results presented below are the average of 10 trials.

The experiments ran on DECStation 5000/200s (25Mhz Mips R3000A) 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 10Mbps Ethernet, and 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.

5.0.2 Cardinality

Figure 4 shows the results of running the benchmark on sets of size N of uncached 16KB files on one server with $Comp = 1$. 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:



This graph shows the cost and benefit of SETS vs. set cardinality. The points are experimental results 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.

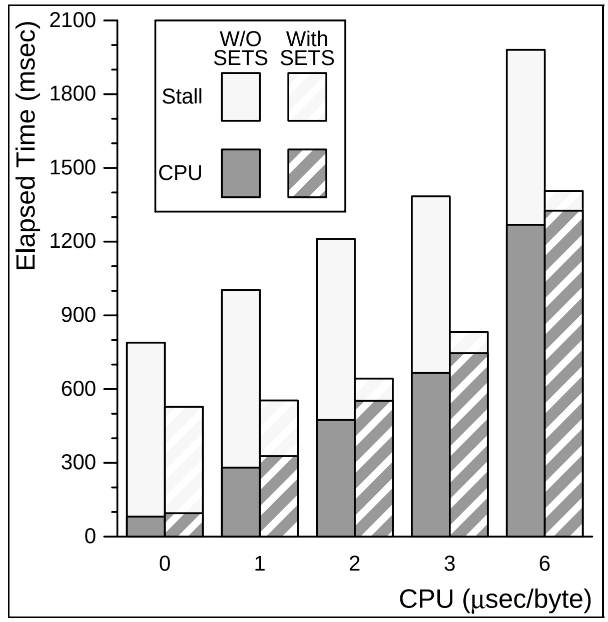
Figure 4: Benefit of SETS vs. Cardinality

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. This increase in computation is shown by the higher line for CPU when using dynamic sets. Fortunately, this increase is small and in particular much smaller than the decrease in latency from prefetching.

From whence comes the reduction in latency? One source is clearly the ability to overlap computation and I/O. Rather than blocking, the application can process data and the system can send and receive other messages, reducing the amount of idle time with legitimate work. Another source is a higher utilization of the I/O system, which results in higher I/O efficiency. For instance, while the server is waiting for a disk read to complete it can process other read requests or send data over the network. It should be noted that this higher utilization from prefetching can have a negative impact if the server or network is fully utilized by demand traffic. SETS also derives a small benefit by pre-reading a file's data immediately, while Unix read-ahead must wait for a sequential access pattern to be established.

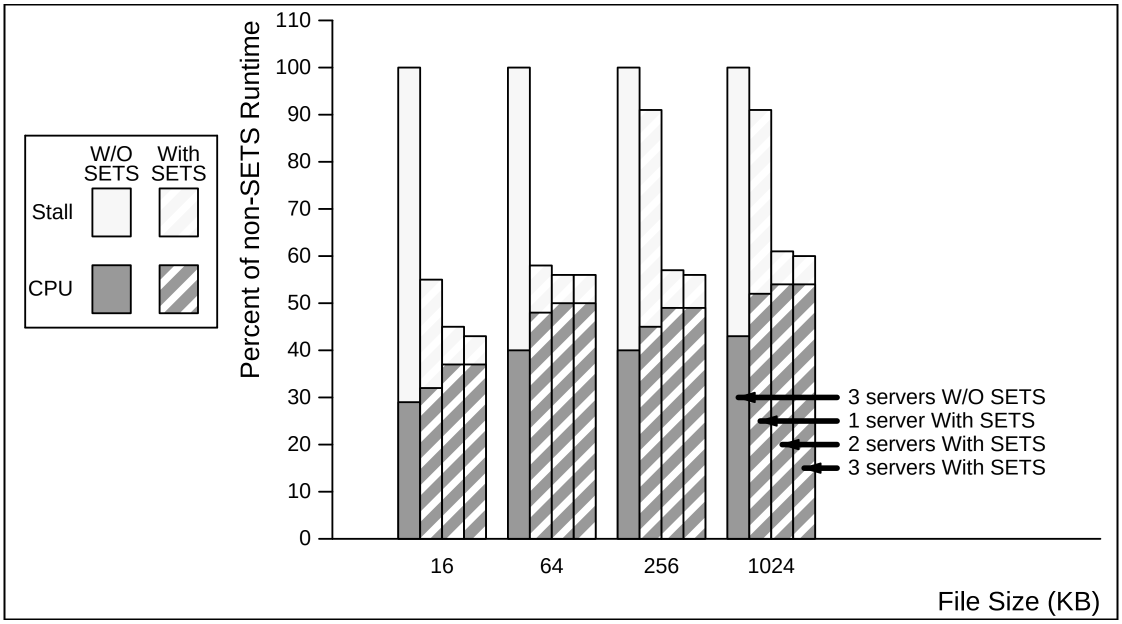
5.0.3 Overlapping Computation and I/O



This graph shows the time to run synthGrep with different amounts of computation (μsec/byte). Solid bars show the results for runs without and stripped bar for runs with SETS. The light portion of each bar is the amount of time spent in the idle loop, which indicates the amount of time the application blocked on I/O. From the graph, one can see that SETS can reduce latency by overlapping I/O and computation. For higher amounts of computation the application becomes compute bound, which reduces the affect of prefetching.

Figure 5: Benefit of SETS vs. Computation

One reason that prefetching can lower latency is that I/O can be performed in parallel with computation, hiding the delays and increasing client CPU utilization. The second experiment examines this effect by varying the amount of computation ($Comp$) performed by synthGrep on sets of 12 16KB files stored on one server. Figure 5 show the results of this experiment. As shown in the graph, there is almost no difference in synthGrep's runtime between $Comp = 0$ and $Comp = 1$ when using SETS, even though the application spends more time computing. The additional computation hides I/O latency from the application, reducing the amount of idle time. For $Comp > 2$, the application is compute bound because SETS has eliminated as much latency as it can. For higher values of $Comp$, the potential benefit from prefetching to runtime becomes insignificant.



This graph shows the time to run synthGrep on different file sizes and files stored on multiple servers. Each cluster represents a file size, and bars within the cluster are normalized to the total time for runs without SETS. The dark portion of each bar is the time spent computing, the light portion is the time spent in the idle loop stalled on I/O. The graph shows that SETS can exploit parallelism through concurrent prefetching.

Figure 6: Benefit from SETS vs. Concurrent Prefetching

5.0.4 The Effect of Parallel I/O

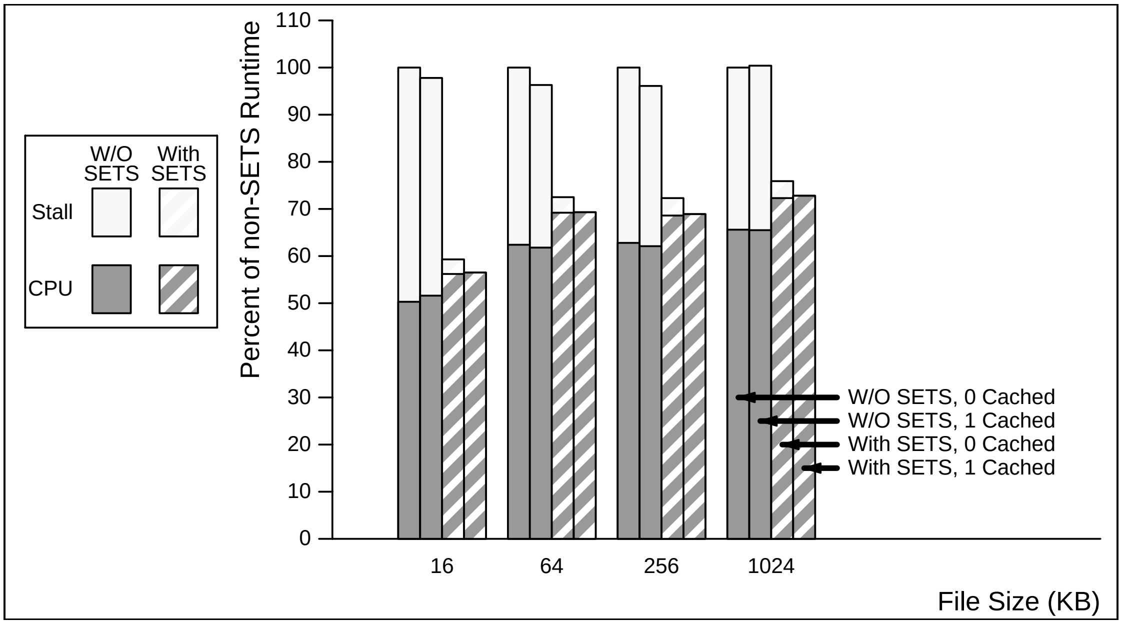
A second benefit of prefetching is the ability to exploit parallelism by fetching data from independent disks or servers concurrently. Such parallelism would exist, for instance, if a search's candidate objects were stored on multiple servers. The third experiment examined the effect of concurrent fetches by running synthGrep on sets of files stored on one, two, or three servers. Because SETS is able to eliminate most of the latency to access 16KB files by overlapping I/O and computation, this experiment also ran synthGrep on larger files.

Figure 6 shows the results of running synthGrep on sets of 12 files of equal size, with $Comp = 1$. The graph shows four clusters, each corresponding to a different file size (16, 64, 256, and 1024KB). The bars within each cluster correspond to (from left to right) a set of 12 files stored on three servers (4 files per server) using standard file system operations[2], 12 files on one server, 12 files on 2 servers (6 on each) and 12 files on 3 servers (4 each). The leftmost bar in each cluster presents times without using SETS, the other bars are for runs with SETS. All values are normalized to the average total execution time without SETS. By comparing the results across clusters one can see the effect of file size on the relative benefit from dynamic sets, by comparing within the cluster one can see the effect of parallel fetches.

This experiment has two chief results. First, by comparing the bars corresponding to runs on one server, one can see that the benefit from SETS drops off for larger files. The reason is that the relative benefit SETS gets by prefetching decreases as the performance improvement from read-ahead increases. 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[2, 21, 25].

The second result is that SETS is able to exploit parallelism between servers to virtually eliminate latency, even for large files. In fact, the remaining latency is close to the minimum achievable by the implementation's use of whole file transfer, since the best SETS can do is eliminate all latency but the time to fetch the first file. Without prefetching, NFS can only read from one file, and thus one server at a time, and so cannot exploit parallelism between servers as can SETS. The drawback of concurrently fetching data is that it consumes more network and server bandwidth by fetching the same data in a shorter amount of time.

[2] There is no significant difference between times for non-SETS tests with one, two, and three servers, so the graph only shows the results for three servers.



This graph shows the time to run synthGrep when one set member is cached. Each cluster represents a file size, and bars within the cluster are normalized to the total time for runs without SETS. The dark portion of each bar is the time spent computing, the light portion is the time spent in the idle loop stalled on I/O. These results show that SETS can eliminate I/O when a file is cached by reordering access to use the cached file before it is evicted.

Figure 7: Benefit of Reordering When One File Is Cached

5.0.5 Reordering

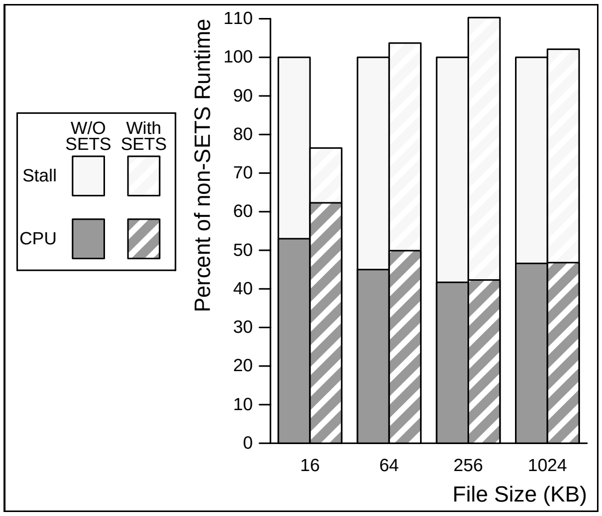
In addition to the benefits of prefetching, dynamic sets allow the system to reorder fetches. Reordering is advantageous when I/O latency differs between members, such as when some members are in the cache when the set is created. Figure 7 shows the results of an experiment which cached one member of the set before running synthGrep, and used sets of 12 files of equal size stored on 3 servers. In order to best demonstrate the benefits of reordering, the experiment used $Comp = 3$ to achieve the maximal benefit from prefetching. The graph in Figure 7 shows four clusters of bars corresponding to files of 16, 64, 256, and 1024KB in size. The two bars on the left of each cluster correspond to runs without SETS, the ones on the right to runs with SETS. The first and third bars in each cluster show the results when no files were in the cache, the second and forth bars show the results when one set member was cached.

The chief result of this experiment is that reordering allows SETS to eliminate all I/O latency. In the previous experiments, SETS could not eliminate the latency to fetch the first file since the application had no data on which to perform computation. A secondary effect is shown in the 1MB file tests. Because the client's buffer cache is too small to hold the entire set, the cached member is evicted before the application can read it. By reordering, SETS is able to determine the member is cached and yield the object before its data is evicted. The benefits of reordering are more clearly seen when the disparity in latency is very high, such as when some fetches timeout.

5.0.6 Accessing Data from the Local File System

The previous experiments show that SETS offers substantial benefits in the domain for which it was designed – search on a distributed file system. Figure 8 shows the results of running synthGrep on files on the local disk. The experiment ran synthGrep on sets of 12 files stored on one disk, using $Comp = 1$. The graph shows clusters corresponding to file sizes of 16, 64, 256, and 1024KB, normalized to run times without SETS. For small files (16KB and smaller), SETS reduces latency and overall runtime. For larger files, however, SETS' prefetching results in an increase in latency!

Three factors contribute to this negative result. First, we designed SETS to prefetch into the local file system, and so it does not prefetch aggressively from the local disk. Second, the Unix read-ahead mechanism is very effective at reducing latency, leaving little additional opportunity for SETS. Third, SETS prefetching strategy, which was designed for network reads, attempts to prefetch from more than one file at a time. As a result, the accesses seen by the disk are not sequential, and force the disk to seek more often. The performance penalty incurred from these seeks does depend on data layout, and by carefully placing the



This graph shows the time to run synthGrep on sets of local files. Each cluster represents a file size, and bars within the cluster are normalized to the total time for runs without SETS. The dark portion of each bar is the time spent computing, the light portion is the time spent in the idle loop stalled on I/O. These results show that prefetching local files off one disk has limited benefit over read-ahead, but SETS still provides a sizeable benefit (25%) for the majority of files which are small.

Figure 8: Benefit of SETS for Local Disk Files

data on disk we were able to eliminate the increase in latency. However, controlling data layout in this manner is not practical in a real-world setting. An alternate strategy that avoided concurrently reading more than one file from the same disk should not suffer this problem. In addition, we could easily extend SETS to use a system like TIP2[23] to manage local disk prefetching, if it were available on the same platform.

6 Dynamic Sets and the Web

We now turn to the question of whether search on the World Wide Web could benefit from dynamic sets. The Web is an interesting domain because latencies are very high, there is substantial variance in latency between different servers and over time, and because Web search tends to be interactive. Unfortunately, Web browsers currently only support "point-and-click" interaction, which leaves little opportunity to use set iterators. However, one could easily extend a browser's interface to support user-controlled creation and iteration over sets of objects, and then use dynamic sets to reduce I/O latency. It is critical that users control the creation and membership of sets in order to ensure the accuracy of the hints inherent in a set.

There are a number of cases where iteration over sets is possible. Any hypertext page can be thought of as a set whose members are the objects to which the page has a link. For instance, Web search engines represent the query results as an HTML page, many Web servers have a top-level page that serves as an index of their site, and many pages contain links to sites with related information. If a user decides that she might wish to visit some number of the links on a page, she could create a set by selecting these links and then iterate on the set to view the members. There exist tools to prefetch sets of objects (such as WebCompass[32]), but these tools use predefined sets and prefetch set members well in advance of a search. Dynamic sets, if successful, would allow searchers to specify sets at runtime and still substantially reduce the latency of processing the sets.

6.1 Adding Web Support to SETS

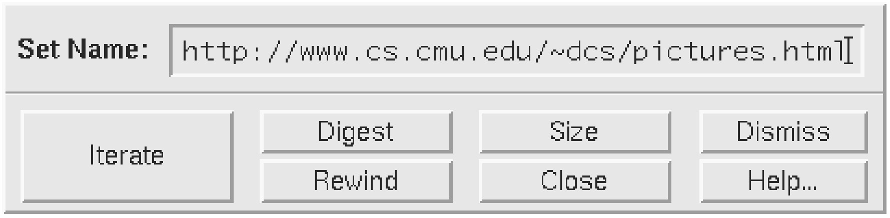
In order to evaluate the use of dynamic sets on the Web, we implemented a warden to allow SETS to prefetch Web documents and to query search engines, and extended the NCSA Mosaic 2.6 browser to use dynamic sets. The browser redirects queries to search engines through the warden when requested to do so by the user, and the warden parses the response to extract links. Currently all links on a page are added to the set, but this is just a limitation of our prototype. In fact, our Warden can use any hypertext page to define a set's membership since search engine queries are URLs (Web document names) and return HTML.

Once a set is created, the browser displays a pop-up dialog such as the one in Figure 9. Users request the next set member by clicking the "iterate" button; mosaic gets the member by calling the set's iterator and displaying the object it receives. The warden prefetches whole objects to the local disk using the standard `HTTP` protocol and stores the `HTTP` headers with the objects to allow Mosaic to properly parse their data.

6.2 Experimental Methodology

Because the Web is so large and amorphous, capturing its performance characteristics accurately in a model, simulation, or clean test environment is difficult. Our solution overcomes this problem by replaying traces of real searches to achieve both repeatability and realism. The traces[3] were captured by recording the activity of 5 expert Web users, each performing 3 searches and spending 10 minutes per search. 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 user think time –

[3] Bill Camargo at Transarc, Inc. designed and implemented the trace capturing mechanism.



This window appears when a user opens a set. Clicking on the "Iterate" button causes Mosaic to get the next set element by calling the set's iterator. The other buttons allow users to see the member names ("Digest"), print the set cardinality ("Size"), open a new set to begin iteration again ("Rewind"), and to close the set ("Close").

Figure 9: Mosaic Window for Managing Open Sets

the amount of time the user spent examining the object before moving on. The five traces can be viewed as independent samples from the population of directed search activity performed by expert Web users. Figure 10 summarizes the traces to give an idea of the workload they represent.

To create equivalent traces that use dynamic sets, we manually copied these traces, replacing demand load operations with iteration over sets. We defined these sets by creating 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.

It is important to realize that since the creation of the 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 she defines. The benefits shown by the experiments below are realizable in practice only to the extent that the user adopts this mode of operation.

6.3 Experimental Results

We replayed the traces on a DECStation 5000/200 with 64MB of RAM; all client caches were flushed prior to each run. 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 replay mechanism loads the objects in the trace from live Web servers, Mosaic displays the objects, and then pauses to approximate the user think time captured in the traces. The trace output records the latency seen by the trace mechanism, the amount of time Mosaic spent processing the object, and the amount of simulated user think time. The client shared a 45Mbps T3 connection to the Internet with several thousand other computers[4]. The traces were replayed during peak hours (afternoon EST) for greatest realism; other experiments that replay the traces on weekends, without loading inlined images, and over a phone line see vastly different latencies but similar benefits to those shown here[30].

Figure 11 shows the results of replaying these traces, broken out by search task and averaged over 5 runs. Each bar consists of three parts: the user think time captured in the trace, CPU to fetch and display the images, and the latency seen by Mosaic. The labels on each cluster denote the search task and user that cluster represents; solid bars show the times for runs that did not use dynamic sets and striped bars show the times for runs with sets.

Figure 11 shows three chief results. First, dynamic sets can dramatically reduce aggregate I/O latency on the Web by overlapping egregious Web latencies with even larger user think times, and by fetching data in parallel. The results show between a 70% and 98% reduction in latency, which means that users would wait much less time for their data if they were using dynamic sets. Second, reducing the latency reduces the magnitude of variance in latency which results in a more predictable, and therefore more usable system. Third, the savings from dynamic sets largely depend on the composition of the set, the amount of user think time, and the speed of the network. In the extreme, dynamic sets offer no performance benefits for sets of 1 object, and induce a small overhead to create the set.

This experiment also demonstrates the advantages of reordering. Several of the fetches in each trace take tens of seconds to complete. Prefetching alone would force the user to block on these fetches, even though other objects are waiting to be processed. Because of the nature of iterators, SETS can yield any member that is ready, and thus overlap these long fetches with user think time to substan-

[4] In the final paper we will identify the organizations that share the link, omitted here for interests of blind reviewing.

	Task A					Task B					Task C				
Trace #	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
Think(sec)	378	174	211	284	110	322	385	244	307	266	351	263	281	228	286
# of Objects	21	13	12	17	3	15	20	12	9	10	39	16	16	20	12
# of Images	12	30	26	15	12	56	9	22	19	18	48	30	6	46	21
Bytes(KB)	226	80	134	154	136	254	263	152	459	176	132	63	131	273	256

Figure 10: Summary of 5 WWW Search Traces

tially reduce the amount of time the user is blocked waiting for data.

7 Future Work

We are in the process of extending the work reported here in several ways. First, clients with low bandwidth connections have little opportunity to benefit from concurrent prefetching, and we are exploring other alternatives. Second, prefetching in a distributed system is inherently difficult since it involves finding an ideally optimal schedule with incomplete knowledge in the face of dynamic changes in resource availability. We are exploring strategies that will allow the prefetcher to dynamically adapt its behavior to meet the needs of applications while avoiding overrunning shared network and server resources. Third, our work has focused on the systems issues of dynamic sets. We are now exploring issues of user interface design related to use of dynamic sets on the Web. In particular, we plan on exploring more powerful membership specification techniques that would allow users to create sets without *a priori* knowledge of the members. Finally, we are in the process of porting SETS to NetBSD and Linux, and implementing dynamic sets as a Netscape plug-in for release to the wider Internet community.

8 Conclusions

Dynamic sets are a new operating system abstraction that gives systems greater opportunity to transparently reduce I/O latency and better captures application behavior. We have demonstrated that by exploiting the semantic non-determinism of iterating over sets, systems can reduce latency over a wide range of systems through reordering and informed prefetching. These benefits do not depend on locality of reference and therefore apply to applications for which caches and predictive methods perform poorly. Dynamic sets can be implemented without requiring modifications to protocols or servers, and so can be easily deployed. Finally, dynamic sets adhere to established software engineering principles by preserving interface boundaries and shielding applications from low-level system details.

Dynamic sets address the problem of I/O latency by exposing the an application's non-determinism and future data needs to the system, which can exploit this knowledge to reduce latency. Applications benefit from prefetching without having to manage I/O explicitly, and the system is given greater knowledge with which to schedule I/O and manage resources. As a result, the system can prefetch without accurate predictions of latency by fetching a small number of objects concurrently and opportunistically yielding the first to return.

9 Acknowledgements

This paper describes work originally presented in my thesis[30] performed while I was a Ph.D. student in the School of Computer Science at Carnegie Mellon University. My advisor, M. Satyanarayanan, made significant contributions to the work, as did my thesis committee – Garth Gibson, Jeannette Wing, and Hector Garcia-Molina. Hugo Patterson, Andrew Black, and Dan Revel read early versions of this paper; their insightful feedback helped me improve this document tremendously.

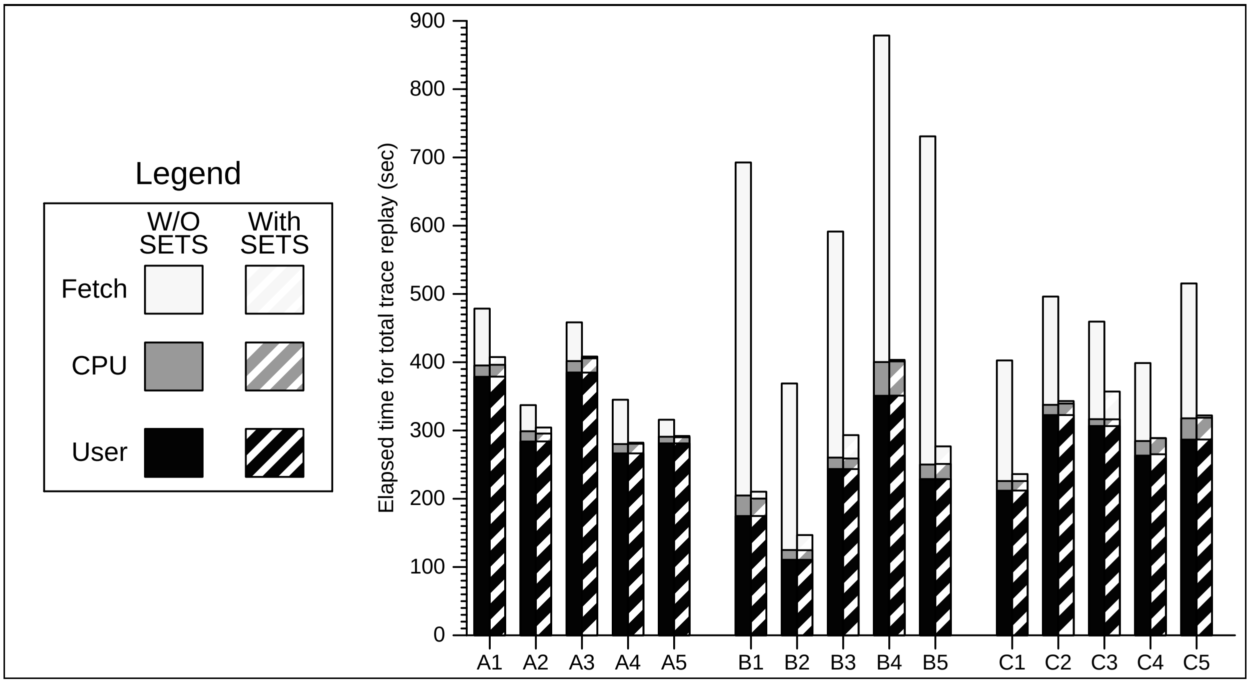
References

[1] 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.

[2] 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).

[3] BOWMAN, M., SPASOJEVIC, M., AND SPECTOR, A. File system support for search. Transarc white paper, 1994.

[4] 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*



This graph shows the results of replaying traces of user search activity on the live Web. Five users were traced, and each trace consisted of three search tasks. The graph shows the cumulative user think time, amount of computation to display data, and I/O latency to replay the trace of one search task for one user. The chief feature of this graph is the potential savings from latency that can be obtained by using dynamic sets.

Figure 11: Results of Replaying User Traces on the Web

Measurement and Modeling of Computer Systems (May 1995).

[5] 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).

[6] 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).

[7] 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).

[8] 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.

[9] 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.

[10] 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).

[11] 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).

[12] 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.

[13] 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).

[14] KLEIMAN, S. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Summer USENIX Conference Proceedings* (Atlanta, 1986).

[15] 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).

[16] KUENNING, G. H. The design of the SEER predictive caching system. In *Proceedings of the Workshop on Mo-*

bile Computing Systems and Applications (Santa Cruz, CA, Dec. 1994).

[17] LISKOV, B., AND GUTTAG, J. *Abstraction and Specification in Program Development*. The MIT EECS Series. MIT Press, Cambridge, MA ; McGraw-Hill, New York, 1986.

[18] 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.

[19] 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).

[20] NELSON, M., WELCH, B., AND OUSTERHOUT, J. Caching in the Sprite Network File System. *ACM Trans. Comput. Syst. 6*, 1 (Feb. 1988).

[21] 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).

[22] 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).

[23] 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).

[24] 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).

[25] SATYANARAYANAN, M. A study of file sizes and functional lifetimes. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles* (December 1981).

[26] 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.

[27] SMITH, A. J. Disk cache – miss ratio analysis and design considerations. *ACM Trans. Comput. Syst. 3*, 3 (Aug. 1985).

[28] SPASOJEVIC, M., AND SATYANARAYANAN, M. A usage profile and evaluation of a wide-area distributed file system. In *Winter Usenix Conference Proceedings* (San Francisco, CA, 1994).

[29] 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).

[30] STEERE, D. C. *Using Dynamic Sets to Reduce the Aggregate Latency of Data Access*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1997. Available as technical report CMU-CS-94-215.

[31] 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).

[32] WEBCOMPASS 1.0. Quarterdeck, Corp. Marina del Ray, CA. (800) 683-6696. Additional information is available at http://www.quarterdeck.com.

[33] 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.