Using Dynamic Sets to Overcome High I/O Latencies During Search

David Steere M. Satyanarayanan

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{dcs,satya}@cs.cmu.edu

Abstract

In this paper we describe a single unifying abstraction called dynamic sets *which can offer substantial benefits to search applications. These benefits include greater opportunity in the I/O subsystem to aggressively exploit prefetching and parallelism, as well as support for associative naming to complement the hierarchical naming in typical file systems. This paper motivates dynamic sets and presents the design of a system that embodies this abstraction.*

1 Introduction

In this paper we consider the problem of high I/O latencies during search or browsing in a large scale distributed data repository such as AFS[10] or the World Wide Web (WWW)[1]. To solve the problem, we identify a new operating system abstraction called *dynamic sets*. The essence of the abstraction is *the explicit grouping of sets of file accesses and the communication of this grouping by applications to the operating system.* We demonstrate that this simple abstraction can have powerful performance implications across a spectrum of operating characteristics.

The advantage of dynamic sets lies in *informed prefetching*[9]. The disclosure of the membership of a group of related files is a hint of future access that can be exploited by the system to gain the benefits of prefetching: exploiting parallelism in the I/O subsystem, overlapping processing of data with its access, and higher resource utilization. In addition, by using a set to represent the grouping, the system is free to optimize the order in which the set members are fetched to further reduce the aggregate latency to process the set. The advantages of dynamic sets can be realized in a number of areas. For example, they can be used to cope with the increasing mismatch between CPU and disk speeds in high-performance computing systems. As another example, dynamic sets allow mobile clients to overlap processing of data with its access over low-bandwidth wireless links. As a third example, the explicit identification of associativity provided by dynamic sets can partially compensate for the lack of temporal locality in contexts such as search.

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.

Preliminary experience with dynamic sets[11] suggests that these potential benefits are indeed realizable. For example, our prototype implementation reduced the total elapsed time to fetch a group of objects over a high latency network by close to a factor of four. The benefits are, of course, highly dependent on the specific applications and system parameters, but the use of dynamic sets often helped and never hurt performance in all of the cases we considered.

2 Motivation

As users of the WWW or a variety of distributed file systems can attest, fetching remote data can involve large delays due to high network latencies. Some applications can exploit caching to reduce the impact of these delays, but others which exhibit little temporal locality tend to have poor cache hit rates. Search (and retrieval) is an example of this latter class. Although Unix-like file system interfaces were not explicitly designed for it, they are frequently called upon to efficiently support many types of search. For instance, searching a collection of source files for a variable declaration is an everyday occurrence. More recently, the widespread use of browsing systems such as NCSA Mosaic has enabled the construction of hypertext documents referencing numerous file system objects.

To understand how dynamic sets can aid search, consider the execution of a simple search command, `grep foo *.c`, in a typical Unix system. The wildcard "`*`" is expanded by the shell, and the application program "`grep`" is given a sequence of filenames matching the pattern "`*.c`". Each file is successively opened, read in its entirety while searching for occurrences of "`foo`", and then closed. Although the precise identity of files needed is determined once the wildcard expansion has been performed, this information cannot be exploited by the operating system to prefetch the files from a disk or over the network. Further, the order of file opens is fixed at wildcard expansion time although searching those files in a different order would still preserve the semantics of the command. This means that the operating system cannot reduce the overall elapsed time for the command by reordering requests to exploit differences in the I/O latencies for different files.

Although the simplicity of `grep` makes it a good choice to illustrate the power of sets, it is by no means the only application that can use dynamic sets. In fact `grep` is representative of a common Unix programming idiom. Search applications with significant processing time (e.g. query-by-image-content (QBIC[8]) or that involve human actions (e.g. clicking for the next element) are other examples for which dynamic sets offer significant promise.

These examples reveal two distinct limitations of current Unix systems. First, knowledge of related file accesses is lost to lower levels of the system even when it is evident to higher levels of the system. Second, an ordering of such file accesses is imposed too early in their handling. Dynamic sets address both these limitations by providing a way in which an application can expose a group of related accesses to lower levels of the system without imposing an unnecessary ordering. Once so informed, the lower levels can then prefetch the objects in a more efficient order, exploit available parallelism in the I/O subsystem (such as independent disks), and overlap the high latency of accessing data with processing activity.

A secondary benefit of dynamic sets is that they allow the superior scaling characteristics of hierarchical naming to be seamlessly combined with the convenient search capability of associative naming. For instance, suppose MIT and CMU maintain databases indexing their computer science technical reports stored in world-wide `AFS`. A user of dynamic sets might browse for a report on some topic by using `grep` to search through files returned by a query run on the databases. Using dynamic sets, such a query would look like "`grep "distributed systems" '/afs/{mit,cmu}/tr-db/\select name from reports where author like "david"\'`".

3 Design of SETS

This section presents the highlights of the design of a realization of dynamic sets, which we call SETS. SETS has three goals. First, the set mechanism must be lightweight to minimize unnecessary overhead. Second, the semantics should be strong enough to satisfy application requirements, but not be overly restrictive on system design. Third, the set interface should be easy to use, while allowing applications to cleanly inform lower levels of future accesses.

A set is a dynamically created collection of objects that only exists as long as the application that created it. If one wants a more permanent collection, one could create it by storing the set and its members in a persistent store.

The mutability of sets and objects raises two important issues: "What is the proper definition of set membership?" and "How current do the members of the set need to be?". These questions have been answered in the context of distributed databases (e.g. read-only transaction[3]), but we believe that these solutions are not appropriate to the large scale distributed systems such as WWW or AFS. First, many types of data do not need such strong guarantees[2, 5]. Second, many current distributed storage systems do not provide the mechanisms necessary for SETS to guarantee data consistency. Third, and most importantly, higher degrees of consistency impose large performance penalties. For instance, one technique to ensure serializability is to block mutations while a query is running. On a system that spans the Internet, however, this could adversely affect availability of data during partitions (a frequent occurrence on the Internet[7]) and the cost of distributed locking would be prohibitive.

For these reasons, SETS provides a much weaker consistency guarantee, captured in these two assertions:

1. Every object in the set satisfied the query at some point during its run.
2. Once an object is in the set, it will remain in the set.

Although these are weak promises, they strike a balance between the needs of scalability and availability, and the need to offer useful semantics. We believe that these promises are sufficient to allow many forms of browsing or searching. An earlier paper sketches the space of weak consistency in this context, and contains a more detailed discussion of these issues[12].

3.1 SETS interface

The operations in the SETS interface are presented in Figure 1. This subsection discusses the four basic operations: `setOpen, setIterate, setDigest`, and `setClose`. For brevity, we omit discussion of the other operations.

Sets are created by calling `setOpen` with a *set pathname* (using syntax explained in Section 3.2), and receiving a handle for the open set in return. The system expands the set pathname to obtain a list of names of individual objects in the set. SETS is free to determine the aggressiveness with which this expansion should be performed. In particular, `setOpen` does not require that any expansion be completed before the call returns. The system may also fetch individual objects in the set at this time, but is not required to do so. `setClose` terminates use of a set handle, allowing the system to free any resources used by the corresponding set.

SETS provides two ways of examining the contents of an open set to allow both browsing (`setDigest`) and iteration (`setIterate`). `setDigest` is very lightweight, presenting only summary information about the members. Each call to `setIterate` returns a standard Unix file descriptor for a previously unprocessed member. The former does not require fetching the member returned, while the latter requires that it at least partially be fetched. Use of `setIterate` provides the system with a stronger hint of future access, allowing the system to more safely allocate resources for prefetching.

The following pseudo code is typical of the way standard Unix applications such as `grep` would use SETS.

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

The command 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, in the case of `grep` reading the file sequentially and searching for a specific string. When processing is complete, the member is closed, and upon termination of the iterator, the set is closed. As one

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

Figure 1: SETS system call interface

can see, it is quite possible to modify existing applications to use sets with little or no knowledge of the details of the application.

3.2 Naming

When a set is opened, the application supplies a string specifying the names of the objects in the set. SETS understands three types of set specifications, an example of each being given in Figure 2. First, in an *explicit specification*, a user enumerates the names of the members of the set, either using full names or pattern matching via csh wildcard notation. Second, *type-specific specifications* are special strings that can be evaluated as queries by search engines, such as SQL databases or WAIS indexes[6], which will return the names of objects that satisfy the query. The specification indicates both the engine to be used and the string, so it is assumed that the proper query format is known to the specifier. Third, *executable specifications* are binaries that return a list of the names of files to be put in a set. This class of set specification raises many issues, such as security and heterogeneity, which are not addressed in this paper. Note that SETS is a framework which provides access to existing services, and is not responsible for providing the search engines (many of which exist already).

In order to integrate SETS with Unix, we have extended the `csh` wildcard notation to include all three forms of set specification. A name in the extended syntax can have a set specification in any component of the pathname. The portion of the pathname after the set specification treats the set members as directories. For instance, `/coda/{cmu,mit}/staff` looks for a subdirectory of `/coda/cmu` and `/coda/mit` named `staff`. If no such directory exists, the resulting set is the empty-set. As another example, a type-specific specification applied to an object of the wrong type will also return the empty-set. A normal Unix name is just a special case of a set pathname that refers to only one object.

4 Status and experience

Our implementation of SETS has taken two forms. The first is a user-level prototype which was built to get an initial understanding of the performance of SETS. The second is a more functional kernel implementation, which is currently in progress.

4.1 User-Level prototype

The user-level prototype[11] is based on a simplistic distributed file system. To use the prototype, applications link in two libraries; one exports the set interface in Figure 1, the other is the client end of the file system. The client parses pathnames of the form `/<serv>/<local-file>`, redirecting I/O requests on the file to the server on machine `<serv>`. No caching is done, although SETS prefetches objects and the client may preread an open file.

To evaluate the prototype, we measured the performance along a number of dimensions. The experiments were based on a parameterized version of `grep`, which allows one to specify the amount of processing time as a function of the data size. In the experiments, we compared the time to process a set of files stored in the file system using SETS to the time without using SETS. Separate experiments examined the effect that the amount of processing per byte, the number of objects in the set, the size of the files in the set, and the speed of the network had on the performance of SETS.

Although the prototype's absolute performance was modest due to implementation inefficiencies, the benefit of dynamic sets turned out to be substantial. We observed up to a factor of 4.25 (out of a predicted 4.39) speedup using one client and four servers. Although this appears to be a greater than linear speedup, it results not only from the four fold parallelism across the servers, but also from the parallelism between the client's processing of the data and the fetching of it. In no case did the use of dynamic sets hurt performance.

We codified our experience with the prototype into a linear performance model[11]. The model was validated by comparing its results with measurements of the prototype. The model confirms that use of dynamic sets should yield benefits in a number of environments, from mobile clients over slow networks to standard Unix applications to applications with high processing per byte of data.

4.2 Kernel implementation

Given the encouraging results of the prototype, we are currently adding the set interface (Figure 1) to the API of several Unix-like systems (Mach 2.6, NetBSD, and Linux). We chose to place SETS inside the kernel to be in close proximity to the name resolution code, and to allow a tight integration between SETS and the lower level file systems

Explicit:	`/coda/usr/dcs/*src*/*.c`
Type-specific:	`/coda/{cmu,mit}/staff/\select home where name like "%david%"\`
Executable:	`/coda/sources/%myMakeDepend foo.c%`

Because many Unix users are already familiar with the `csh` wildcard notation, we extended this syntax to support set specification. *Explicit specifications* use standard csh notation. *Type specific specifications* have 3 portions: the prefix identifies the object on which the query will be run (such as a database), and is terminated by the first "\"; the query is delimited by "\...\"; and the suffix which will be appended to the query's results. *Executable specifications* are similar to type specific; the prefix is the name of the working directory in which the binary will be run, and the "%" delimits the command used to invoke the binary.

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

(UFS, AFS, Coda, etc). We believe this design choice allows SETS to get maximal performance, while allowing a set to contain objects from different file systems.

At the time of this writing, a user of SETS can use `csh` wildcard notation, Informix SQL queries, or a subset of WWW URLs to specify queries. URLs that name HTML documents are treated as queries by parsing the HTML document, identifying all links to other documents, and creating a set to hold those documents. With this, a user of SETS can use Lycos[1], the WebCrawler[2], or other WWW indexing tools to search for WWW documents, and enjoy the benefits of dynamic sets when examining the results.

With this full implementation of SETS, we hope to explore many issues that were raised by the prototype. First, how well can dynamic sets be integrated with an existing distributed file system? How will caching and dynamic sets interact in practice? Second, how difficult will it be for an implementation to achieve substantial improvement via dynamic adaptability? Third, what techniques can be exploited to increase support for mobile, weakly connected clients? Finally, how will dynamic sets perform when exposed to a real user community? Many issues of the effects of competition for and consumption of resources can only be explored when a system is supporting real user workloads.

5 Related work

Our work is most closely related to two efforts previously described in the literature, the Semantic File System (SFS)[4] and Transparent Informed Prefetching (TIP)[9]. The Semantic File System attempts to provide automatic indexing of information in a file system by creating an index at a server, and updating it as files are created or modified. The SFS extends the traditional Unix pathname mechanism to support conjunctive queries over a space of name-value attribute pairs. Dynamic sets has a very different focus: that of reducing the latency seen by a client. As such, the SFS does not attempt to address the primary issues focused on here. However, one could easily envision adding dynamic sets to SFS, thus merging the benefits of both systems.

[1] http://lycos.cs.cmu.edu/cgi-bin/pursuit

[2] http://www.biotech.washington.edu/WebCrawler/WebQuery.html

As a method for prefetching, dynamic sets are similar in nature to Transparent Informed Prefetching (TIP)[9]. The use of sets allows the system to prefetch, lazily fetch, and/or reorder the fetching of objects in the set, whereas TIP hints are limited to prefetching. However, TIP hints can be used to specify how a file will (probably) be read (e.g. stride width), whereas dynamic sets only inform the system that an object will (probably) be accessed. TIP and SETS could easily be integrated; for instance one could envision using TIP to specify read access patterns while iterating on a set.

6 Conclusion

Search on mobile computers and wide-area information systems is likely to suffer from high I/O latencies. Conventional techniques such as caching do not help because there is little or no locality to exploit in such workloads. The work described here is an attempt to exploit the characteristics of search to reduce the aggregate latency to access a group of objects. Early results have shown dynamic sets to be a promising path to this goal. We hope that the full-fledged implementation currently in progress will offer convincing evidence that dynamic sets are indeed a valuable abstraction in operating systems.

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] A.D. Birrell, R. Levin, R.M. Needham, and M.D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4), April 1982.

[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] R. Golding. A weak-consistency architecture for distributed information services. *Computing Systems*, 5(4), Fall 1992.

[6] B. Kahle and A. Medlar. An information system for corporate users: Wide area information servers. *ConneXions – The Interoperability Report*, 5(11), Nov 1991.

[7] D.D.E Long, J.L. Carroll, and C.J. Park. A study of the reliability of internet sites. In *Proceedings of the 10th IEEE Symposium on Reliable Distributed Systems*, Pisa, Italy, Sept 1991.

[8] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, and C. Faloutsos. The QBIC project: Querying images by content using color, texture, and shape. Technical Report RJ 9203 (81511), IBM Research Division, 1993.

[9] 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.

[10] 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.

[11] D. Steere and M. Satyanarayanan. A case for dynamic sets in operating systems. Technical Report CMU-CS-94-216, School of Computer Science, Carnegie Mellon University, November 1994.

[12] J. Wing and D. Steere. Specifying weak sets. Technical Report CMU-CS-94-194, School of Computer Science, Carnegie Mellon University, September 1994.