A Case for Dynamic Sets in Operating Systems

David Steere M. Satyanarayanan

November 30, 1994

CMU-CS-94-216

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Recent trends have exposed three key problems in today's operating systems. The first is the emergence of I/O latency as the dominant factor in the performance of many applications. The second is the need to cope with mobile communication environments where bandwidth and latency may be highly variable. The third is the importance of search activity to locating files of interest in a distributed system. In this paper we describe a single unifying abstraction called *dynamic sets* which can offer substantial benefits in the solution of these problems. 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, presents the design of a system that embodies this abstraction, and evaluates a prototype implementation of the system via measurements and an analytical model.

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

Keywords: Search, Browsing, Mobile Computing, Distributed Systems, File Systems, Prefetching, Performance, Latency, Modeling, Dynamic Sets

1. Introduction

In this paper we consider the problem of high latencies during search or browsing in a large scale distributed data repository such as AFS[13] 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 common scenarios.

Dynamic sets can be used to reduce the aggregate I/O latency of a group of related requests. 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.

To obtain early validation of the potential benefits of dynamic sets, we have built a simple user-level prototype. The prototype exhibits significant performance improvements, although it is limited in scope. For example, on a suite of test programs modelling overlap of data processing with network transmission, dynamic sets reduce total elapsed time by almost a factor of four at 9600 baud. The benefits are, of course, highly dependent on the specific applications and system parameters, but the use of dynamic sets did not hurt performance in any of the cases we considered.

We have also developed an analytical performance model, and confirmed its accuracy by comparing model predictions with measurements of the prototype. The model is valuable in helping us understand the interplay of factors such as network latencies and the amount of client data processing in determining performance. It also enables us to better predict the behavior a full implementation currently under development will exhibit.

We begin the paper with a more detailed rationale for dynamic sets. Next we describe the design for our realization of dynamic sets, then, in Section 4, we describe our prototype implementation. Section 5 derives a performance model for dynamic sets, establishes its validity, and discusses the efficacy of sets in several different arenas. We conclude with a description of work in progress, and a comparison of our work with other related research.

2. Motivation

To understand the value of dynamic sets, consider searching for data on a Unix-like file system. Although the file system interface was not explicitly designed for efficient search, it is frequently called upon to support many types of search. For instance, searching a collection of source files for a variable declaration is an everyday occurrence. More recently, the advent of browsing systems such as NCSA Mosaic enabled the construction of hypertext documents embodying numerous file system objects. In these and other similar examples, the caching embodied in most file system implementations is ineffective at masking I/O latencies — there is very little temporal locality to exploit.

Consider the execution of the simple 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.

Of course, `grep` involves so little processing that there is little to be gained by overlapping processing with I/O. But the reduction in total latency due to reordering of file opens can still be significant if there is parallelism within the I/O subsystem. More importantly, the above example is only a representative of a common Unix programming idiom. One could envision a similar scenario using a query-by-image-content search program [8] or an interactive search program accessing a collection of images stored as Unix files. In those cases, the processing or user think times will be large enough that overlapping them with file access will substantially reduce overall latency.

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. The set abstraction allows applications to identify a related group of file accesses and allows this grouping to be exposed to lower levels of the system without imposing an unnecessary ordering.

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 reports returned by a query run on the databases. In syntax presented below, such a query would look like "`grep "distributed systems" '/afs/{mit,cmu}/tr-db/\select name from reports where author like "david"\'`".

The databases used in the above example effectively serve as "navigational databases" to better focus search in a large subtree of files. A forerunner of this idea was embodied in the Semantic File System (SFS)[4]. WAIS, gopher, and various web *spiders* serve a similar purpose for the WWW. We will discuss the relationship between dynamic sets and these systems in Section 7.

3. Design of SETS

A proper realization of dynamic sets must posses the following characteristics. 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. This section describes the design of an instantiation of dynamic sets on a Unix platform. For clarity, the realization will be called "SETS", while the abstraction will continue to be "sets".

A set is a dynamically created collection of objects. "Dynamic" means a set is not persistent, as it need only exist as long as the application that created it. Preserving the results longer can not only consume resources (sets can be big), but adds no useful function to the system. We posit that searchers are interested in timely information; e.g. a user wishing to examine today's weather in a number of cities would not be satisfied if presented with the data that was obtained when he ran the search the previous week. If a user wishes to create a permanent copy of a set he could create a copy using standard file system tools.

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 feel that these solutions are not appropriate to the large scale distributed systems such as WWW or AFS[14]. For one, many types of data do not need such strong guarantees[2, 5], and two, the target systems do not provide the mechanisms necessary for these strong guarantees.

In addition, long running queries (due to large, physically dispersed data repositories) are likely in a system that spans the Internet. Maintaining tight consistency is either too restrictive (e.g. limiting partitioned access) or imposes too high a performance penalty (e.g. distributed locking). We believe that although some applications require strong consistency, users of many other applications are willing to trade these guarantees for better performance. For this reason, SETS provides a much weaker promise, 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 may seem like weak promises, they strike a balance between the needs of scalability and availability, and the need to offer useful semantics. An earlier paper sketches the space of weak consistency in this context, and contains a more detailed discussion of these issues[15].

3.1. SETS Interface

The operations in the SETS interface are presented in Figure 1. This subsection discusses the four most important operations: `setOpen, setIterate, setDigest`, and `setClose`. The remainder are standard set operations and will not be discussed further.

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)());

Table 1: SETS system call interface

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` assumes that a summary of the objects is sufficient to guide selection, and so can avoid the cost of obtaining the members' data. `setIterate` assumes that the data of all the objects is of interest, and fetches each in turn. The distinction between these operations is similar to the difference between running "`ls`" and using the names or attributes to select a file vs running "`grep foo *.c`" and choosing a file based on the results.

`setDigest` is very lightweight, presenting only summary information about the members. The nature of the summary is dependent on the type of the members. For Unix files, the name of the file is returned by `setDigest`, while an image object summary might be a lower resolution/thumbnail presentation of the image. A special value of `errorCode` is returned to indicate that the set has not yet been fully expanded. Upon seeing this condition, applications know that future calls to `setDigest` on this set may return addition values. This is valuable to interactive applications that cannot afford to wait for the set to be fully expanded.

`setIterate` is used when accessing the contents of individual members. Each call to the iterator returns a standard Unix file descriptor for a previously unprocessed member, so a member must be at least partially fetched before it can be returned by `setIterate`. But its use allows the system to more confidently allocate resources for prefetching the member, since it is a stronger hint of future access to that member than the use of `setDigest`.

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) ) {
       process(fd);
       close(fd);
}
setClose(handle);
```

The command is invoked with the names of sets to process. Each 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 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 pathname specifying the names of the objects in the set. SETS defines three types of set specifications, an example of each being given in Figure 1. First, in an *explicit specification*, a user enumerates the members of the set, either using full names or pattern matching. Second, *type-specific specifications* are special strings that can be evaluated by servers that export a particular type of data, such as a database or the WWW. 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. It is important to note that SETS only provides naming support, and does not supply executable binaries nor the type-specific services (e.g. databases). Because many Unix users are already familiar with the csh wildcard notation, we have extended this notation to include all three forms of set specification.

As seen in Figure 1, a pathname in the extended syntax can have a set specification in any component. The portion of the pathname after a 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 query 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 name that refers to only one object.

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 1: Examples of the three types of names supported by SETS.

4. Prototype Implementation

To gain an understanding of the performance implications of SETS, we have built a prototype based on a simplistic distributed file system. The prototype is a library that exports the set abstraction to applications. The file system consists of a client library and a singly threaded user-level server which exports remote procedure calls (RPCs) to open, read, write, and close files local to the machine on which the server is running. 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 by the client library, although a file may be preread after it has been opened depending on the current activity of the client and set library. Caching at the server is limited to Unix's disk block buffer cache.

The prototype supports the $\{,\}$ `csh` notation, and SQL queries as shown in example 2 in Figure 1. Any component can contain a query, although meaningless queries (such as an SQL query run on the file system root) may return the empty-set. The experiments discussed in Section 5.2 use paths like `/{serv1,serv2,serv3}/tmp/file{1,2,3,4}`.

The application used in the experiments is a prototypical search program. It processes all the members of a set, sequentially reading data from a file and spending a parameterized amount of time processing each byte read. The application has one thread, but the number of threads in the SETS library is specified at run-time. As in a time-sharing environment, if the application uses the CPU for more than a predetermined limit (roughly 10 msec), it is interrupted and other threads are allowed to run. To prevent SETS from blocking the application's forward progress, the SETS threads run at a lower priority than the client thread.

4.1. Limitations of the Prototype

Several implementation decisions were orthogonal to the design of SETS, yet had negative impact on the prototype's performance. These solutions were chosen to ease the implementation, but they also added unnecessary inefficiencies which detracted from the benefit of using dynamic sets. The kernel implementation of SETS currently in progress should avoid these costs.

First is the use off-the-shelf user-level thread and RPC packages([11]), which introduce inefficiencies that restrict the achievable parallelism. The threads package provides non preemptive coroutines, and so can only schedule a thread when the currently active thread explicitly yields. This means the entire application (and client library) are suspended when any thread makes a system call (such as a read from a socket). Additionally, the RPC package has higher client CPU overheads than a streamlined kernel implementation. This loss of parallelism has significant impact when network latencies are are low. Larger network response times tend to hide this effect.

Second, we chose to transfer data via arguments to RPCs, which limits our maximum transfer size to 3K. This means the prototype pays a substantial amount of overhead per byte. Although this effect is lower for a small file, it has a clear impact as the size of the file grows. Predictions of the performance of $SETS$ under other conditions, such as whole file caching (as in Coda[12]) or large block transfer (as in AFS[6]), are presented in Figure 5.

Third, the prototype uses simplistic resource scheduling. For instance, contention for the CPU is managed by the thread package, and does not take contention for other resources (data, sockets) into account. Additionally, the prototype aggressively prefetches without regard to client activity. So a client demand read can be blocked by a preread for some other file on the same server. These prefetch policies will be one of the interesting issues explored in the kernel implementation of SETS.

5. Performance Model

This section presents a linear model of the aggregate latency to fetch a group of objects. The model focuses on the chief advantage of dynamic sets: the exploitation of available parallelism in common data storage systems. This parallelism takes two forms: overlapping CPU and I/O activity, and overlapping I/O through independent channels. Comparisons between the model predictions and experiments on the prototype are presented below and confirm the model to be accurate to within 10%.

To simplify the model, we assume that there is no contention between the acquiring of data and processing of other data, and that communication to independent servers will not interfere with each other. The advantage to this assumption is that it greatly simplifies our model;without it we would need to use a queueing network. The disadvantage is that such contention may reduce the benefit of using SETS, and by ignoring it the model may overestimate the benefit of sets. However, the graphs below show that the model underestimates the benefit in almost every case, and given the relative accuracy of our predictions, the assumption seems reasonable.

In this section, we distinguish between $SERIAL$, the time to fetch and process a set of elements serially (as it is normally done on today's systems), and $SETS$, the time taken using dynamic sets. The predictions and experimental results are presented as ratios between $SERIAL$ and $SETS$, both to bring focus to the key point of the study, as well as to reduce the impact of the prototype's inefficiencies on the results.

5.1. Derivation of Model

The model consists of a number of equations describing the various costs involved in fetching and processing the members of a set. The main parameters, listed below, allow the model to describe a wide variety of scenarios, some of which are discussed in Section 5.3. Other parameters used in the model (e.g. $OpenCost$) are implementation dependent, and are presented in the next section.

s	The number of independent servers storing members of the set.
$band$	The network throughput (bps). All routes to servers have the same bandwidth.
n	The number of objects in the set. Members are evenly distributed across the servers.
$size$	The size of the files. All the files in a set are the same size.
$think$	The amount of processing/byte, either from the application or the user.

In both the $SERIAL$ and $SETS$ cases, three kinds of work get done: opening a file ($OpenCost$), reading from a file ($ReadCost$), and processing data ($think$). The first two are RPCs; the basic cost of an RPC is described in equation (1). $ServCost$ is the amount of processing on the server (different times for $OpenCost$ and $ReadCost$). $Null$ is the round trip time for an RPC with no arguments and for which $ServCost = 0$. These two terms comprise the latency of an RPC. The last term is the bandwidth, where $Bytes$ is the amount of data transferred, $band$ is a model parameter, and $Slope$ is a constant factor containing the percentage of achievable bandwidth and various conversions (such as 8 bits/byte).

$$RPCCost = Null + ServCost + \frac{Slope}{band} \times Bytes \tag{1}$$

Equation (2) shows the amount of work done to process an open file. In the prototype, an open returns one buffer of data, and a read triggers a preread of the next buffer. Thus processing and reading of a file's data can be overlapped, saving the application the lesser of the two costs. $nReads$ represents the number of RPCs required to read the file, and is a function of the size of the read buffer, how much data is read at open time, and the manner in which end-of-file is detected.

$$Work = max(size \times think, nReads \times ReadCost) \tag{2}$$

The equation for $SERIAL$ is fairly simple, the only overlap occurs from prereading the data in an open file. The cost for $SETS$ is constrained as follows: before data can be processed it must be read; before a file can be processed, it must be opened. The read pipeline is started by the open fetching a buffer; the open pipeline is started by opening the first member. However, every time an RPC is made, $s - 1$ other RPCs can happen in parallel.[1] In particular, this means each RPC delay can be overlapped with the processing of the results of s previous RPCs. The last term in (5) represents the cost to drain the pipeline.

$$\begin{aligned} Serial &= n \times (OpenCost + Work) & (3) \\ end &= ((n-1) \bmod s) + 1 & (4) \\ Sets &= OpenCost + \frac{n-s}{s} \times max(s \times Work, OpenCost) + end \times Work & (5) \end{aligned}$$

The chief benefit of using dynamic sets is summarized in the second term of equation (5); effectively, s opens can be done at once. In addition, if the client is sufficiently CPU intensive (large $think$), network costs can be completely hidden behind client processing. We conjecture that in the future the amount of processing between demand fetches will continue to be high, due to both sophisticated applications (e.g. pattern matching, signal processing), and the reliance of flow control on human actions (such as "clicking" for the next element) by some applications. Both the model and the prototype show that these effects can be additive.

5.2. Validation of Model

The model was validated by comparing its predictions with the results of experiments on the prototype. Three experiments were run to show the accuracy of the model along different dimensions: varying the size of the set (n), the size of the files ($size$) in the set, and the bandwidth ($band$). The graphs in Figure 2 and Figure 3(a) present both the predicted (curves) and observed (points) results for the three sets of experiments, while the tables in the appendix present the means (and variance) of the observed results. A note of caution: in order to make details of the curves visible, the scale of the y-axis changes from graph to graph, although the scale of the x-axis remains the same.

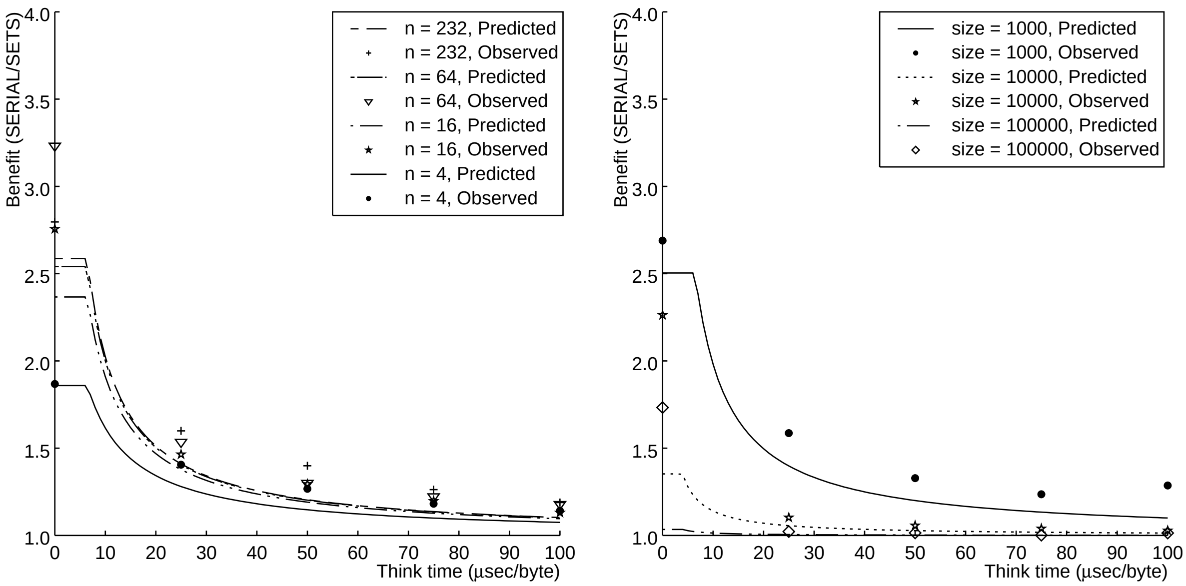
The experiments were run on 5 Decstation 5000/200s (1 client/4 servers) connected by 10base2 Ethernet, running the Mach 2.6 operating system. The data files were uniformly distributed over the servers, and resided on the servers' local disks. The data sets were small enough to fit into the servers' buffer caches (with the exception of $size = 100000$), and the experiments were run on warm server buffer caches. Each data point is the mean of 10 trials; no effort was made to restrict use of the network nor the machines during the experiments. Times were obtained via a hardware cycle counter which our measurements show to be accurate to 50 microseconds.

To obtain the RPC parameters used by the model in the previous section, we timed round trip delays to open a file and read one buffer; successive calls increased the size of the buffer from 0 bytes to 3000 bytes. The parameters obtained are: $Null = 5.1$ msec, $Slope = 8221$ (unitless), $ServCost = 8.3$ msec for opens, and $ServCost = 6.4$ msec for reads. With these numbers, our network delivers roughly 4 Mbps end-to-end.

Low bandwidth connections were emulated by a delay mechanism in the low levels of the RPC package. This mechanism multiplies the size of the packet and the desired bits per second to determine the time the packet should be delayed before being sent. Successive packets to the same machine will be additionally delayed behind earlier packets, although packets to different machines will not affect one another. Although this mechanism is an adequate predictor of actual networks[7], we suspect it added some error to our results. In particular, the delay mechanism uses the client CPU, increasing the contention for it, and reducing the amount of attainable parallelism. An effect of this contention can be seen in the discrepancy between the model and experimental results in Figure 3(a). For smaller values of $band$, the amount of CPU required to delay packets is higher, and the observed discrepancy is larger. Unfortunately, it is too difficult to experimentally prove that the error for these curves is entirely due to the use of this mechanism.

Examining the graphs in Figures 2, and 3(a), one can see that the model closely predicts the benefit of sets over several dimensions. In fact, the average magnitude of the error is 9.5% ($\sigma = 11.2$). The high variance is due to three data

[1] This stems from our assumption that sufficient network bandwidth is available; see Section 5.3.1



(a) Predicted and observed benefit of $SETS$ vs n (b) Predicted and observed benefit of $SETS$ vs $size$

These graphs presents the results of two experiments on the prototype. The lines show the model predictions, and the points the measurements of the prototype over Ethernet ($band \approx 4Mbps$), and in both cases $s = 4$. The high error at ($size = 10000, think = 0$) and ($size = 100000, think = 0$) in (a) and at ($n = 64, think = 0$) in (b) is mostly due to the model's restricting $SETS$ aggressive prereading of data.

Figure 2: Two dimensions along which the model was validated.

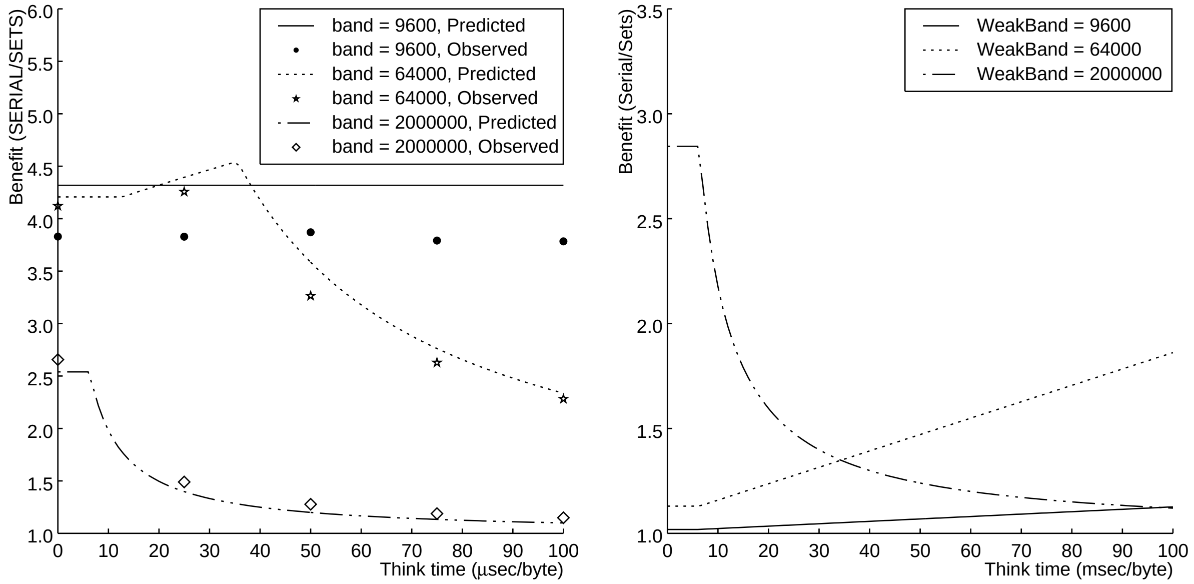
points, without which the average and variance would be 7.3% and 3.9 respectively. The three data points occur at $(size = 100000, think = 0)$, $(size = 10000, think = 0)$, and $(n = 64, think = 0)$. The error in all three is largely due to the fact that our model uses a simplified prefetching strategy which is more restrictive than the prototype's. A more exact (but complicated) model presented in Appendix 1 significantly reduces the error of these three points, yielding average error magnitude of 8.7% ($\sigma = 5.6$).

Each curve can show potentially three distinct phases (as exemplified by the $conn = 64$ Kbps curve in Figure 3(a)). Initially the curve is flat, since processing time is completely subsumed by I/O costs. Next, a small benefit is gained by further overlapping processing with open costs. Finally, the effective benefit of sets drops off, asymptotically approaching 1.0 as processing costs become the dominant factor. For $conn = 9600$ bps, only the first phase is seen; the cost of processing never exceeds the cost of reading a buffer. For $conn = 2$ Mbps and higher, processing costs overwhelm open costs before they overwhelm read costs, thus skipping the second phase.

It should be noted that even in the worst cases, $SETS$ does provide a benefit (although the percentage benefit is small). The reduced benefit is due to the decreased contribution of I/O to overall performance; $SETS$ reduces I/O costs significantly in all the cases we examined. Thus even in the bad cases, if $think$ is human processing time, the savings from $SETS$ will be very noticeable because the latency is directly visible to the user.

5.3. Model Extensions

We now use the model to predict the benefit from using dynamic sets in several different situations. The first subsection extends the model slightly to predict the performance of dynamic sets when run on a mobile client connected to the network by a single low bandwidth link, such as a cellular modem. The following subsection then explores the impact



(a) Predicted and observed benefit of $SETS$ vs $band$ (b) Predicted benefit of $SETS$ over weak connections

Graph (a) presents both the predicted (lines) and observed (points) benefit from using $SETS$ for various values of $band$, with n = 40, s = 4, and $size$ = 1000 bytes. We conjecture that a large portion of the difference between the predicted and observed benefit for 9600 and 64000 bps is due to the use of a slow-network emulator. Graph (b) presents predictions for $SETS$ over a weak link having bandwidths of 9600, 64000, or 2000000 bps respectively, and with $band$ = 4 Mbps, s = 4, n = 40, and $size$ = 1000 bytes. The curve for $weakBand$ = 2000000 bps has the same shape the one for $band$ = 2000000 bps in (a) because the influence of the weak link is overcome by other costs.

Figure 3: Performance benefit of $SETS$ for high latency or weakly connected clients

that a whole-file or large-block caching strategy would have on the performance of dynamic sets.

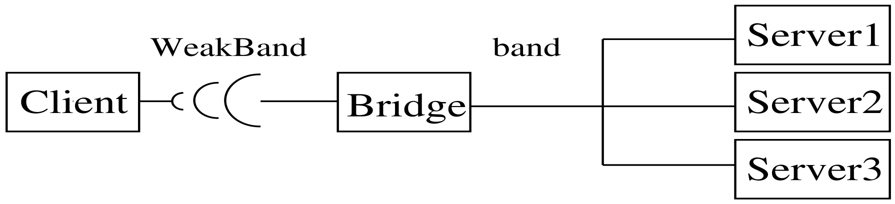
5.3.1. Weakly connected mobile hosts

A common scenario in the future will be a client connected over a weak (low bandwidth or lossy) link to the Internet, and then over various Internet routes to servers (see Figure 4). For these clients, a significant portion of communication costs will be transmission over the weak link, and since all communication must use the link serially this may have drastic effects on the benefit of dynamic sets.

Our original model is extended to include this scenario by adding a parameter, $WeakBand$, which is the bandwidth of the weak link. Since the weak link is an independent component, the cost of using it can be considered separately from the other costs. Equation (6) defines $WeakLink$ to be the total time that data is being transferred over the weak link.

$$WeakLink = n \times size\frac{Slope}{WeakBand} \qquad (6)$$

Equations (7) and (8) show the costs for $SERIAL$ and $SETS$ over the weak link. In the former case, the cost of the weak link is added to the other costs since the files are processed serially. In the latter case, the use of the client CPU can be overlapped with the use of the weak link, which in turn can be overlapped with use of the s servers. Unlike the



This picture shows a weakly connected mobile client connected over a wireless link to a set of server. The portion of the network to the left of the *bridge* is characterized by $WeakBand$, while the portion to the right is characterized by $band$.

Figure 4: Example of a weakly connected mobile host.

model in Section 5.1, the client starts the pipe by sending off the first message ($Startup$ is the client protocol overhead to send a message), drives the weak link at full utilization until the set is exhausted, and finishes up by processing the last buffer ($Finish$ is $think$ multiplied by the amount of data in the last buffer).

$$Serial = n \times (OpenCost + Work) + WeakLink \tag{7}$$

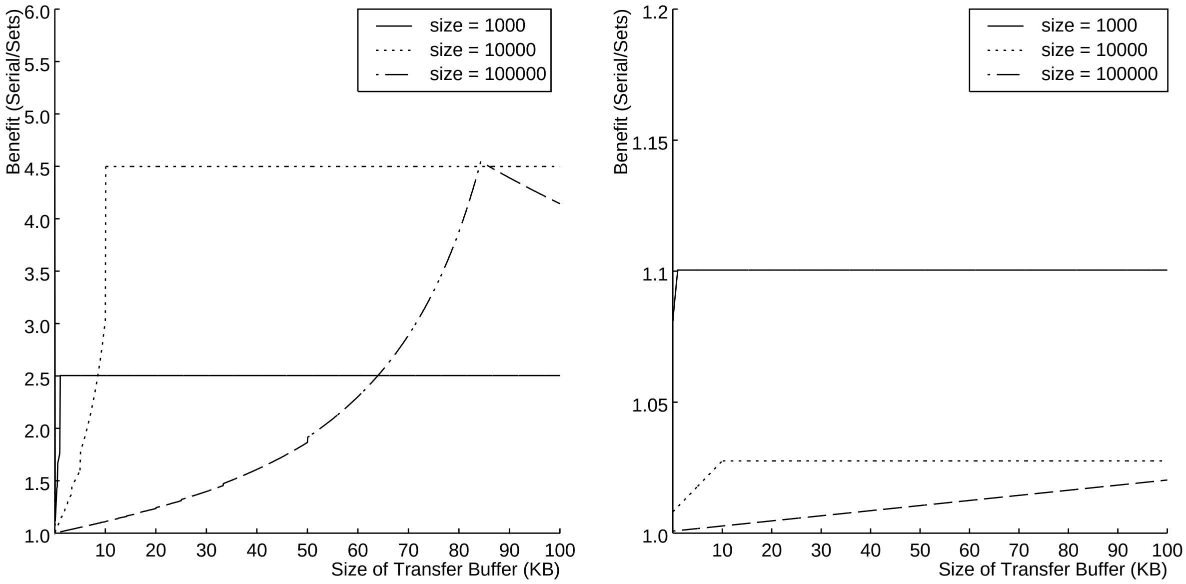
$$Sets = Startup + max(WeakLink, max(n \times Work, \frac{n}{s} \times OpenCost)) + Finish \tag{8}$$

The graph in Figure 3(b) shows the predictions of this model. The graph uses $n = 40$, $size = 1000$, $s = 4$, and $band = 2$ Mbps. The curves present the behavior for three different weak link bandwidths. When $weakBand =$ 2000000, the behavior is similar to that presented in Section 5.2 because communication is a small part of the overall cost. In the other two cases, different behavior is predicted. Both have a rising cost, since the weak link contributes the largest part of the latency. Since $SETS$ can effectively overlap both client and server processing with weak link transmission, it does not pay a cost for client processing, and thus the positive slope. This is an important result, as we conjecture resource poor mobile clients connected over weak radio or telephone links will be common in the future.

5.3.2. Impact of cache block size

The benefit from $SETS$ is maximized when a balance is achieved between the work done to open set members and work done to process them. To understand this effect, we used the model to predict performance as the buffer size is increased from 0 to 100KB. The low end corresponds to demand fetch of data, while the high end corresponds to whole-file prefetch. The graphs in Figure 5 present the benefit vs the transfer buffer size, each graph using the same model parameters as the data points ($think = 0, s = 4, n = 40$) and ($think = 100, s = 4, n = 40$) from Figure 2(b). For reasons of brevity, the intermediate values of $think$ are not shown. As expected, the benefit of sets is reduced as the influence of $think$ increases and diminishes the influence of I/O on overall performance. (Note that the scale of the y-axis is not the same for the two graphs.)

Each curve shows that maximum performance is obtained when the amount of work to fetch data is split between the open and the read phases. For small files that is equivalent to whole-file transfer. For large files, however, the high cost of fetching the file at open can delay the application. This is an example of where the fetching policy of the system can greatly affect overall performance. We hope to explore these issues, and to develop policies to allow systems to correctly adapt their strategy to current operation load.



(a) benefit vs bufsize, $think = 0$ msec/byte (b) benefit vs bufsize, $think = 100$ msec/byte

These graphs show that the benefit of dynamic sets is partly dependent on the choice of prefetching strategy. The X-axis shows the maximum amount of data that can be transfered at a time, while the Y-axis shows the benefit of $SETS$. Lower X values correspond to demand fetching pages; higher values correspond to large chunk or whole file caching. As can be seen, maximum performance is obtained when the correct balance between fetching at open and pre-fetching data is obtained. In both graphs $s = 4$, $n = 40$, and $band = 4$ Mbps. Note that the scale of the y-axis is not the same for the two graphs.

Figure 5: Predicted effect of the size of the unit of transfer on the benefit of SETS

6. Work in Progress

The initial results are sufficiently promising that we are currently implementing a refined and more complete version of SETS in several versions of Unix (Mach 2.6, NETBSD, Linux) and integrating it with several types of storage systems (Coda, Informix databases, WWW). To achieve maximal performance, we have added the interface in Figure 1 to the system API, placing SETS inside the kernel in close proximity with the name resolution code. With this design choice, we should be able to achieve tight integration with multiple lower level file systems, allowing a set to span different 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[2], the WebCrawler[3], 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

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

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

competition for and consumption of resources can only be explored when a system is supporting real user workloads.

7. Related Work

The problem of locating information in a distributed systems has received alot of attention over the past decade. There are two aspects to this problem: locating the information that satisfies a query and fetching the information for the application. The former is the domain of Information Retrieval (Salton and McGill[10] is a good introductory text). One reason so many researchers have focused on this aspect of the problem is that until now, the cost to locate the information subsumed the cost to fetch it. The rise of wide area systems and mobile clients, however, changes this picture because of the high latency to access remote data. Our work focuses on the second aspect of search and assumes the existence of reasonable indexes, and so can leverage the results of others.

There are a number of recent systems that can benefit from our techniques. The most popular of these is the WWW[1], a hypertext document that spans the Internet. As mentioned in Section 6, we are developing a system that treats URLs as queries, treating the documents pointed to by links in an HTML file as members of the result set. We hope the benefit of dynamic sets can then be made apparent to users of the Web.

Another example is the Semantic File System (SFS)[4], which 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.

As a method for prefetching, dynamic sets are similar in nature to Transparent Informed Prefetching (TIP)[9]. In fact, dynamic sets and TIP were both inspired from the same basic problem, and dynamic sets can be thought of as a very strong form of TIP-style hints. 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.

8. Conclusion

The current status of our work provides preliminary confirmation of the feasibility and potential benefits of implementing dynamic sets in a Unix environment. Although our prototype is limited in many ways, it is realistic enough to give us first-hand experience in the use of dynamic sets. Our understanding of the performance benefits of dynamic sets is confirmed by the validation of our model with respect to the prototype. The more refined implementation of SETS currently under way will allow us to realize the full potential of dynamic sets.

As discussed earlier in the paper, search in various forms is an increasingly important activity in computing systems. Current file system mechanisms perform poorly in the absence of temporal locality, which is typical of search scenarios. We believe that dynamic sets will substantially improve the performance and functionality of search in distributed environments, including ones involving mobile clients.

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] M. Kazar, B. Leverett, W. Anderson, V. Apostolides, B. Bottos, S. Chutani, C. Everhart, A. Mason, S. Tu, and E. Zayas. Decorum file system architectural overview. In *Summer Usenix Conference Proceedings*, Anaheim, CA, 1990.

[7] L. Mummert and M. Satyanarayanan. Large granularity cache coherence for intermittent connectivity. In *Summer Usenix Conference Proceedings, Boston*, 1994.

[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] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[11] M. Satyanarayanan. *RPC2 User Guide and Reference Manual*. School of Computer Science, Carnegie Mellon University, October 1991.

[12] 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.

[13] 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.

[14] D. Steere, M. Satyanarayanan, and J. Wing. Dynamic sets for search. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, August 1994.

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

A Extending model to allow aggressive read-caching

As mentioned in Section 5.1, the model presented earlier does not capture the full advantage of $SETS$. Since it can open a file before the file is requested by the application, $SETS$ can also start prereading the file if network bandwidth is available. Unfortunately, this adds substantial complexity to the model, and only reduces the error when $think$ is low. There are two reasons why it does not help more as $think$ grows. First, the client spends more time thinking, allowing more of the reading to be done in the simple prereading scheme of $SERIAL$, and thus reducing the benefit of $SETS$. Second, I/O is a smaller percentage of overall performance, thus reducing the potential benefit of sets.

The new model allows the set mechanism to preread an open file if a server is not being used. For any file i, $FreeTime_i$ is the amount of time the network has been unused since that file was opened. Equation (9) defines this to be the difference between the time to process the bytes in files 1 through i, and the time to open file i and to open and fully read files 1 through $i-1$ at s servers. The symbol $\dot{-}$ connotes subtraction over natural numbers, i.e. if network time exceeds processing time, $FreeTime_i$ is defined to be 0.

$$FreeTime_i \quad = \quad i \times size \times think \;\dot{-}\; \frac{OpenCost + (i-1) \times (OpenCost + nReads \times ReadCost)}{s} \tag{9}$$

The time to process a file ($Work_i$) is redefined to allow some of the read work to be performed during the free time. Equation (10) shows that some number of reads may have been done before the application first examines the file, and this reduces the number of reads that have to be done during the processing of the file.

$$Work_i \quad = \quad max(size \times think, (nReads - \left\lfloor \frac{FreeTime_i}{ReadCost} \right\rfloor) \times ReadCost) \tag{10}$$

Equation (11), like equation (5) in the simpler model, has three terms: the cost to start the pipeline, the cost of a pass through the pipe times the number of passes, and the cost to drain the pipe (process the last end members). Here we use summation instead of multiplication, since the cost of each term can change over time. Since $SERIAL$ cannot read a file before it opens it, the equation for it remains unchanged from (3).

$$Sets \quad = \quad OpenCost + \sum_{i=0}^{\lfloor \frac{n}{s} \rfloor - 1} max(\sum_{j=i \times s+1}^{(i+1) \times s} Work_j, OpenCost) + \sum_{j=1}^{end} Work_j \tag{11}$$

The overall effect of this change on the accuracy of the model is to lower the average error from 9.5% (with $\sigma = 11.2$) to 8.7% (with $\sigma = 5.6$). In particular, the error at the three most troublesome data points for the last model was substantially reduced (from 67% to at most 27%), and at only one data point was the error larger than 20%. We conjecture that the remaining error is the result of ignoring the contention; a queueing model would probably reduce the error further, but we feel the effort would not be commensurate with the greater accuracy.

B Experimental Data

Table 2, Table 3, and Table 4 present the raw data corresponding to the data points in the graphs in Figure 3(a), Figure 2(a), and Figure 2(b). Three experiments are presented: the first shows the effect of bandwidth, the second shows the effect of set size, and the third shows the effect of file sizes on the cost of processing a set of objects. Each number is the mean of 10 trials (with standard deviations in parentheses). All trials were run on warm caches. The numbers for $SERIAL$ represent the elapsed time to fetch and process a set of elements serially, as would be done today on a typical Unix system. The numbers for $SETS$ are the elapsed time to fetch the same set of objects using dynamic sets.

The experiments were run on the prototype described in Section 4 using 5 Decstation 5000/200s (1 client/4 servers) connected by 10base2 Ethernet, running the Mach 2.6 operating system. The data files were uniformly distributed over the servers, and resided on the servers' local disks. The data sets were small enough to fit into the servers' buffer caches (with the exception of $size = 100000$). To reflect the costs that real users would see, no effort was made to restrict use of the network nor the machines during the experiments. Times were obtained via a hardware cycle counter which our measurements show to be accurate to 50 microseconds.

		Think time (msec/byte)				
$band$		0	25	50	75	100
9600	$SERIAL$	44920.3 (37.3)	45773.4 (129.5)	46887.5 (85.5)	48175.0 (252.1)	49045.3 (72.8)
	$SETS$	11726.6 (279.1)	11954.7 (214.0)	12114.1 (84.7)	12706.2 (324.1)	12960.9 (240.6)
64000	$SERIAL$	6281.3 (19.8)	7490.6 (10.4)	8473.4 (85.3)	9428.1 (92.2)	10612.5 (16.8)
	$SETS$	1523.4 (22.4)	1759.4 (66.4)	2595.3 (23.7)	3585.9 (29.9)	4648.4 (97.1)
2000000	$SERIAL$	1100.0 (42.6)	2123.4 (13.0)	3143.8 (46.8)	4154.7 (37.9)	5176.6 (59.3)
	$SETS$	414.1 (16.0)	1425.0 (6.3)	2460.9 (51.5)	3493.8 (67.5)	4504.7 (50.4)

Table 2: Average Elapsed time to process a set for 3 values of $band$.

Table 2 shows how the increasing cost of network communication affects the cost to process a set of objects. Each number shows the cost to fetch and process a set of 40 objects stored on 4 servers, where each object is 1000 bytes long. The numbers were obtained using a slow-network emulator which has been shown to be an adequate predictor of actual networks[7].

		Think time (msec/byte)				
n		0	25	50	75	100
4	$SERIAL$	111.0 (4.7)	221.9 (6.3)	332.8 (10.0)	426.6 (7.2)	528.1 (9.4)
	$SETS$	59.4 (6.3)	157.8 (4.7)	262.5 (6.3)	360.9 (4.7)	462.5 (7.7)
16	$SERIAL$	460.9 (12.6)	860.9 (13.0)	1275.0 (26.3)	1665.6 (15.9)	2073.4 (28.0)
	$SETS$	167.2 (7.2)	587.5 (31.4)	984.4 (9.9)	1390.6 (12.1)	1832.8 (58.1)
64	$SERIAL$	2196.9 (665.2)	3562.5 (29.7)	5151.6 (35.0)	6857.8 (110.1)	8690.6 (684.8)
	$SETS$	679.7 (17.5)	2320.3 (18.8)	3960.9 (51.5)	5607.8 (74.4)	7373.4 (387.9)
232	$SERIAL$	9120.3 (236.3)	14728.2 (172.1)	21120.4 (794.2)	26681.3 (298.4)	32864.0 (573.7)
	$SETS$	3260.9 (573.3)	9206.3 (257.3)	15090.7 (96.0)	21131.3 (237.4)	27640.5 (1085.0)

Table 3: Average Elapsed time to process set increasing size of set

Table 3 shows how the number of objects in the set affects the cost to process a set of objects. Each number shows the cost to fetch and process a set of 1000 byte objects stored on 4 servers, across an Ethernet (ours can effectively deliver data at 4 Mbps). Note how the increase in $think$ decreases the relative benefit of sets. This is due to the increasing importance of processing time to total elapsed time; dynamic sets can only reduce I/O costs.

Table 4 shows how the size of the objects in the set affects the cost to process a set of objects. Each number shows the cost to fetch and process a set of 40 objects stored on 4 servers across an Ethernet (ours can effectively deliver data

		Think time (msec/byte)				
$size$		0	25	50	75	100
1000	$SERIAL$	1315.6 (62.8)	2396.9 (96.9)	3342.2 (43.3)	4386.0 (124.4)	5961.0 (1109.7)
	$SETS$	489.1 (7.2)	1510.9 (25.2)	2515.6 (23.2)	3548.5 (35.3)	4632.8 (179.4)
10000	$SERIAL$	3490.6 (33.7)	12845.4 (366.4)	23100.1 (343.5)	33128.2 (556.5)	43209.5 (305.4)
	$SETS$	1542.2 (7.2)	11637.5 (248.2)	21847.0 (271.3)	31857.9 (309.2)	42012.6 (468.8)
100000	$SERIAL$	32348.5 (877.5)	121810.8 (1370.0)	228832.3 (3836.0)	323741.1 (3260.5)	431325.3 (8410.4)
	$SETS$	18665.7 (152.0)	119126.3 (1649.0)	225519.1 (8904.0)	323006.5 (4891.8)	425243.8 (6351.4)

Table 4: Average Elapsed time to process set increasing size of members

at 4 Mbps). The effective benefit of SETS was limited by the prototype's transfer buffer size of 4KB. Section 5.3.2 argues that this implementation choice can substantially affect the achievable benefit from using dynamic sets.