The following paper was originally published in the
Proceedings of the USENIX 1997 Annual Technical Conference
Anaheim, California, January 6-10, 1997

An Analytical Approach to File Prefetching

Hui Lei and Dan Duchamp
Computer Science Department
Columbia University

An Analytical Approach to File Prefetching

Hui Lei and Dan Duchamp

Computer Science Department
Columbia University
New York, NY 10027
{lei, djd}@cs.columbia.edu

Abstract

File prefetching is an effective technique for improving file access performance. In this paper, we present a file prefetching mechanism that is based on on-line analytic modeling of interesting system events and is transparent to higher levels. The mechanism, incorporated into a client's file cache manager, seeks to build semantic structures that capture the intrinsic correlations between file accesses. It then heuristically uses these structures to represent distinct file usage patterns and exploits them to prefetch files from a file server. We show results of a simulation study and of a working implementation. Measurements suggest that our method can predict future file accesses with an accuracy around 90%, that it can reduce cache miss rate by up to 47% and application latency by up to 40%. Our method imposes little overhead, even under antagonistic circumstances.

1 Introduction

This paper reports the effectiveness of a predictive file prefetching technique that operates automatically and without any sort of information supplied by applications or users. The technique, which is incorporated into the client's cache manager, makes extra requests to the server, hopefully in advance of the actual need for the data. Prefetched data is then placed in the client's cache.

We hypothesize that there is pronounced regularity in file access patterns and relatively simple algorithms can identify an access pattern and quickly spot it when it re-emerges later during another run of the application. In particular, we build semantic data structures, called *access trees*, that capture potentially useful information concerning the interrelationships and dependencies between files. An access tree for a program records all the files referenced during one execution of the program. For each program, we maintain a number of access trees in virtual memory that represent distinct file usage patterns. When a program is re-executed, we compare the access tree being formed by current activity against saved access trees to determine which usage pattern, if any, is recurring; we then prefetch the files remaining in the saved access tree.

File prefetching brings two major advantages. First, applications run faster because they hit more in the file cache. Second, there is less "burst" load placed on the network because prefetching is done only when there is network bandwidth available rather than on demand. On the other hand, there are two main costs of prefetching. The first is the CPU cycles expended by the client in determining when and what to prefetch. Cycles are spent both on overhead in gathering the information necessary to make prefetch decisions, and on actually carrying out the prefetch. The second cost is the network bandwidth and server capacity wasted when prefetch decisions inevitably prove less than perfect.

We have conducted our study in the UNIX environment, which is ubiquitous in academia and the research community. It has been well recognized that on UNIX most files are accessed in their entirety and sequentially [19, 2]. We take advantage of this phenomenon in two ways. First, we effectively model file system events at the level of whole files, and look for access patterns across files. Second, if a file is predicted, we prefetch the initial portion of the file only, letting the standard sequential readahead mechanism bring in the rest when the file is demanded.

Section 2 details the prefetching mechanism. Section 3 reports results from a trace-driven simulation. Section 4 describes the implementation and presents some initial performance data. Section 5 discusses related work.

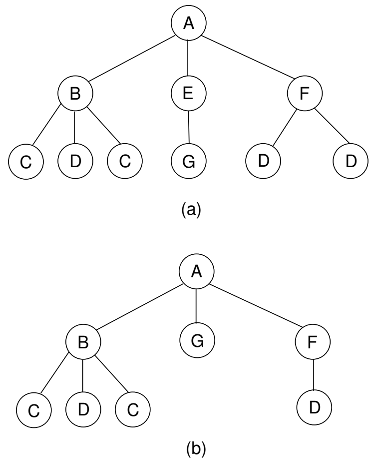


Figure 1: **Example Access Trees.** These graphs show the access trees generated from the events described in Section 1.2. Graph (a) shows the version before compression. Graph (b) that after both vertical and horizontal compression, assuming E is a shell program.

2 Mechanism

2.1 Data Abstraction: Access Tree

Every program can invoke other programs. In UNIX-style operating systems, this is usually realized by the executing program forking child processes, which in turn execute other programs. The programs may also open some data files. All the file references in an application can be formulated into a tree data structure, dubbed an access tree. Files (program files and data files) are the nodes, and an edge is drawn from parent A to child B if either (1) program A invokes program B or (2) program A opens data file B. The order of siblings reflects the chronology of file accesses.

Figure 1(a) depicts the access tree for an application A that includes the following activities:

1. Program A invokes program B
2. B opens data files C and D, in that order
3. B opens C again
4. A invokes programs E and F
5. F opens D
6. E invokes program G
7. F opens D again

An access tree is subject to two kinds of compression. Vertical compression draws edges through UNIX shells, effectively cutting them out of the access tree. This is necessary because of the role shells play as command interpreters. Shells are invoked in a variety of circumstances and can generate a large number of file usage patterns. Since we perform file prefetching for each program file in the access tree and it is infeasible to prefetch accurately for shells, we choose to ignore them. Horizontal compression removes consecutive accesses of the same data file. The detail of consecutive accesses offers no help in prefetching and can be safely omitted. However, non-consecutive accesses of the same data file are preserved for use in a later phase. Assuming program E is a shell in the previous example, Figure 1(b) shows the access tree after compressions.

An access tree describes the context in which file references are incurred in an application. We feel that this context information, if made available to the prefetching mechanism, would decrease the apparent randomness of file system events and permit more accurate predictions of future file references.

As an application proceeds, we construct an access tree for it by intercepting `fork`, `execve`, `open`, `chdir` and `exit` system calls. `Execve` and `open` calls deal with references of program files and data files respectively. `Fork` provides information on how processes, or program executions, are associated with each other. Information on `chdir` calls is used to resolve relative pathnames of files. The access tree is completed upon `exit` of the program execution. An access tree that is being constructed by current activity is called a *working tree*; a finished access tree that is saved to exemplify a file usage pattern is called a *pattern tree*.

2.2 The Prefetch Algorithm

We use a heuristic function `compatibility(w_tree, p_tree)` which returns an indication between 0 and 1 of the degree to which a working tree `w_tree` and a pattern tree `p_tree` resemble each other. When the function's value is above constant `MATCH_THRESHOLD`, we say that the two trees *match*, meaning that they are similar enough that they are considered to belong to the same access pattern. We shall explain the definition of `compatibility` function in more detail after we discuss the basic operation of the mechanism.

As mentioned earlier, a number of pattern trees are saved in virtual memory for each program; a working tree is constructed in the course of every program execution. Whenever a program references a file, a new child node is added in the working tree for the program, and some analysis is performed to find out whether any saved pattern trees can be prefetched. Our analysis follows a simple guideline: if there is no previously prefetched pattern tree or the current working tree no longer matches the prefetched pattern tree, compute the compatibility of the working tree and each of the pattern trees for the program. If any match is found, then prefetch the pattern tree with the highest compatibility. If more than one pattern tree bears the highest compatibility, then prefetch the one most recently saved.

The above analysis is carried out for each executable file inside the working tree whenever the executable initiates a file reference. At this point, a pattern tree may have already been prefetched for the executable, either as the result of prefetch analysis incurred by earlier file accesses or in the form of a subtree of the larger tree we prefetched at a higher level. The executable can always prefetch another pattern tree, based on the analysis result. This effectively allows minor prefetch corrections to be made, reducing the cost of a bad guess.

Two complications may arise when the pattern tree selected for prefetching is large: prefetched files may be evicted from the cache before they are actually referenced, and the cache is considerably destroyed when the prefetching guess is bad. To diminish the extent of these problems, we place an upper limit (`PREFETCH_CAPACITY`) on the number of files from a pattern tree we shall prefetch at one time. When the pattern tree selected is too large, we prefetch only those files in the initial portion of the tree so that the prefetch limit is not exceeded. We also record the immediate child node of the pattern tree we prefetch last. When later the working tree extends to this child, we will prefetch the remaining portion of the pattern tree, if the latest working tree still matches it. We measure `PREFETCH_CAPACITY` in number of files, rather than in number of bytes, because such a measure goes well with the access tree structure, where each tree node represents a file. Further, as illustrated later, only the initial portion of a predicted file will be prefetched, hence the cost of an incorrect prediction is proportional to the number of prefetched files.

On program exit, we compare the newly completed working tree with the saved pattern trees. If it doesn't match any of the pattern trees, the working tree is saved as new information. Otherwise, it is substituted for the pattern tree that it matches best.

We set the two algorithmic parameters to the following values:

- `MATCH_THRESHOLD`: 0.4
- `PREFETCH_CAPACITY`: 15

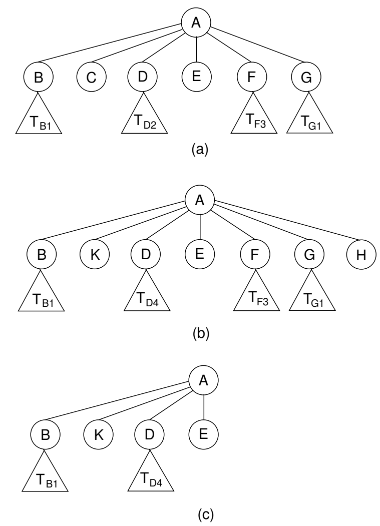


Figure 2: **Computation of Compatibility.** Graph (a) shows a pattern tree. Graph (b) shows a finished working tree, bearing a 0.575 compatibility with the given pattern tree. Graph (c) shows a unfinished working tree, so far bearing a 0.5 compatibility with the pattern tree; E is the pivot in the pattern tree, since it corresponds to the latest child node in the working tree being formed.

We have found that the behavior of our mechanism is insensitive to the value of these parameters. We shall illustrate this later.

2.3 Compatibility Computation

We now take a closer look at the `compatibility` function. This function, essentially a similarity metric between access trees, abstracts out the complexities of prefetch analysis and pattern tree maintenance. We illustrate the definition with the example in Figure 2. A pattern tree is shown in 2(a).

Let us first consider the case that the working tree is a finished one. We try to pair up the immediate child nodes in the working tree with identical child nodes in the pattern tree, preserving the order of the nodes. Recall that a child node can represent either an executable file, which may root another access tree, or a data file. For the two trees to be considered to describe the same file access pattern, we require that there be a one-to-one correspondence between all the executable files, but not all the data files. However, the same executable file may root a different access subtree in the working tree than in the pattern tree. We define C_d as the percentage of data files that can be paired up, C_e as the percentage of pairs of executables that root the same access subtree. Intuitively, C_d suggests how "compatible" the data files are, C_e suggests how "compatible" the executable files are. We choose to use the average of the two values as the compatibility of the two trees in question.

Let us consider the finished working tree in Figure 2(b). There are three data files in the working tree and two in the pattern tree. Out of these five files, only the two appearances of E can be paired up, making C_d 0.4. Three out of four executable pairs (B, F and G) root identical access subtrees, so C_e is 0.75. The compatibility is thus 0.575.

The case of an unfinished working tree is similar except that one child in the pattern tree is first determined to be the *pivot* node. The pivot corresponds to the most recently added child node in the working tree. Only the child nodes in the pattern tree that appear before the pivot are involved in compatibility computation. If the pattern tree is selected, we prefetch those files that follow the pivot in the sequence of pattern tree preordering, since those before the pivot probably have already been accessed. Given the unfinished working tree in Figure 2(c), node E is the pivot in the pattern tree. C_d and C_e are both 0.5, giving rise to a 0.5 compatibility. If we decide to prefetch this pattern tree, only the files in subtrees T_{F3} and T_{G1} will be prefetched.

Since the compatibility function is invoked often, it is important that it not be expensive. That is why we examine only immediate child nodes. The time complexity of the computation is proportional to the number of child nodes.

3 Simulation

Our initial assessments of the mechanism were obtained by trace-driven simulation.

3.1 Method

We gathered several file traces [24] on SunOS 4.0.3c. This version of SunOS offers a "C2 secure computing facility" that includes the ability to produce a system call audit trail. Using this feature, we gathered three traces of a volunteer user performing his normal work activity over a period of two weeks. The first trace contains 25,441 invocations of previously enumerated system calls captured over 72 hours. The second trace contains 23,858 invocations captured over 52 hours, while the numbers for the third trace are 85,962 and 86, respectively. During these hours activity varied widely and included compilations, document production, data analysis and display, large file searches, news reading, printing, and other operations.

Our simulated cache manager used an LRU replacement policy. It stepped through the traces, maintaining the cache in accordance with the user file accesses. Since our trace data lacks file sizes, we defined cache size by number of files. We varied the cache size and for each cache size, we compared the results of prefetching against those without prefetching.

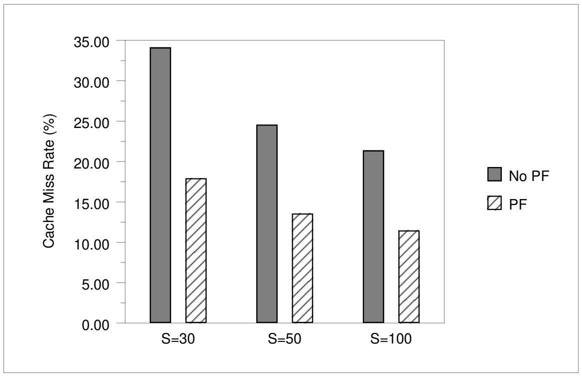
3.2 Results

Our first metric was the cache miss rate. The results for the three traces appears in Figure 3. Prefetching delivers a substantially better miss rate with all cache sizes. The results are worse for the second trace because it includes a recursive directory traversal (i.e., the UNIX `find` program) over a hierarchy that includes thousands of files that are never accessed again.

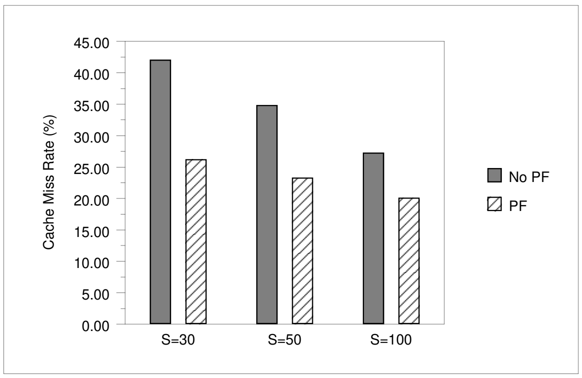
In addition, we monitored the cache behavior at a finer grain: how well our mechanism worked over each run of 500 accesses. At the end of each run, we checked to see whether prefetching had beaten no-prefetching by measuring the number of misses during that run. As shown in Table 1, prefetching won this comparison easily and consistently. This shows that our mechanism's superiority is steady and stable, and not simply a result of a few exceptionally fruitful prefetch sequences. The few losses due to bad prefetching guesses are more than offset by the many wins.

Both Figure 3 and Table 1 indicate that the increased intelligence of our mechanism is more effective in smaller caches. This was expected. Consider that, in the extreme case of an infinitely large cache, any file appearing in any access tree is still in the cache. There is no room for improvement from any prefetching algorithm that is based solely on information about the past.

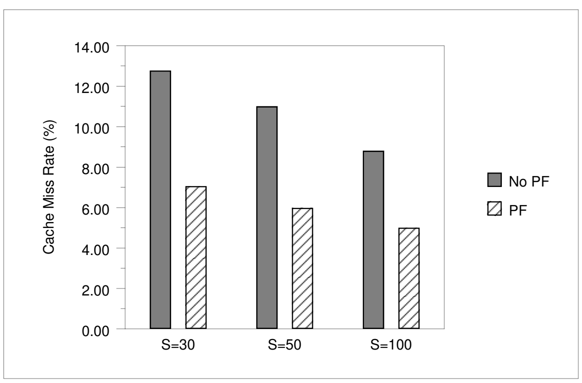
We measured the accuracy of our prefetch decisions, defined as the percentage of file access predictions that were actually used. We also calculated one overhead of the mechanism, i.e., the percentage of the file fetches that were initiated due to bad prefetch decisions. This reflects the network bandwidth and server capacity that were wasted. Table 2 shows these results. (We shall address another overhead, the CPU cycles expended by the prefetcher, in Section 4.) Because of the LRU policy, bigger caches give better accuracy results because prefetched entries have a better chance to survive cache entry replacements. Similarly, bigger caches also give better overhead results because bad predictions have a bet-



(a) Trace 1



(b) Trace 2



(c) Trace 3

Figure 3: **Cache Miss Rate.** These charts compare the accumulative cache miss rate with prefetching (PF) against that without prefetching (No PF). The comparison is performed under three different cache sizes (S), measured by number of files.

Trace	Cache Size	Wins	Ties	Losses
1	30	31	6	0
	50	29	6	2
	100	23	12	2
2	30	31	4	0
	50	30	4	1
	100	23	9	3
3	30	88	45	8
	50	82	54	5
	100	69	65	7

Table 1: **A Finer-grained Comparison.** At the end of each run of 500 file accesses, we determined whether prefetching had beaten no-prefetching by measuring the number of cache misses during that run.

Trace	Cache Size	Accuracy (%)	Overhead (%)
1	30	94.15	5.13
	50	95.17	4.61
	100	96.04	3.83
2	30	88.09	8.60
	50	90.92	6.93
	100	93.07	5.34
3	30	90.68	12.51
	50	93.40	9.63
	100	95.36	7.25

Table 2: **Prefetching Accuracy and Overhead.** The *accuracy* is the percentage of the predictions that were actually used. The *overhead* is the percentage of the file fetches due to bad guesses.

ter chance to have already existed in the cache; in such a case, no fetch is performed.

The `compatibility` function plays a critical role in our mechanism. The assumption underlying this function is that if the initial portions of two access trees bear a high compatibility, so will the two trees in their entirety. In order to test this assumption, we examined the *number of loads*, *i.e.*, the total number of times we prefetched pattern trees. We further classified these loads as either *proper* or *improper*. A load became improper when a different pattern tree was selected by the prefetcher to take the place of the current one, or when the finished working tree did not match the pattern tree loaded. Otherwise, the load was proper. Our results in Figure 4 suggest that the compatibility function is effective.

Finally, we note that the algorithmic parameters should be stable: small variations in the settings must not produce large performance degradations.

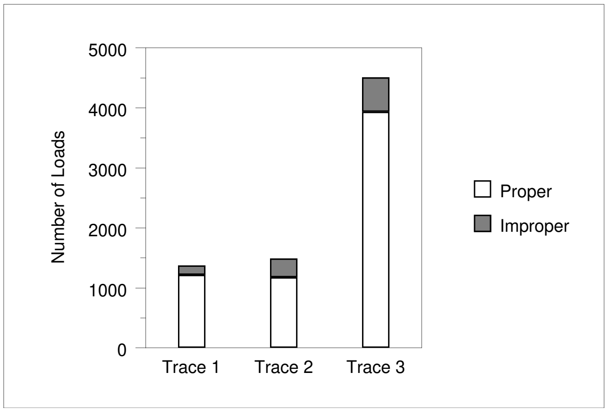


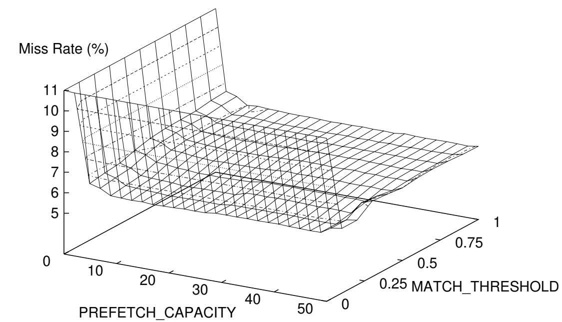
Figure 4: **Effectiveness of the** `compatibility` **Function.** This figure illustrates, for each trace, the split of pattern tree loads into proper ones and improper ones.

To illustrate the stability of the algorithmic parameters, we measure the mechanism over a wide range of parameter values. In Figure 5, we show the results for Trace 3 and a cache size of 50. Three measurements are presented: cache miss rate, predication accuracy and prefetch overhead. The results for the other two traces and cache sizes are similar.

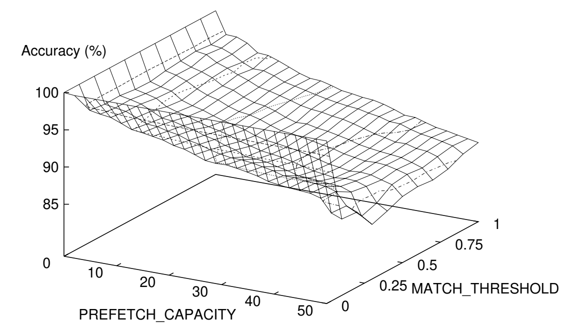
The simulation demonstrates that our algorithm can accurately predict future file accesses based on past file usage. The major limitation of the simulation study is that it does not account for the relative timing of events due to the absence of such information in the traces used. As a result, prefetched files are assumed to appear in the cache instantaneously. It remains to be determined whether a real system will have the resources to exploit the information on future file accesses. The limitations of the simulation motivated us to conduct a full implementation and further evaluations.

4 Implementation

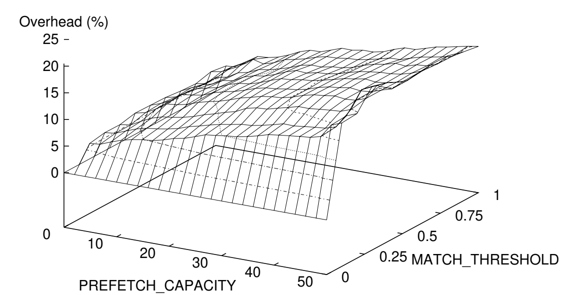
We have implemented our mechanism in UX42 [8], a BSD UNIX server running on Mach 3.0 [1]. UX42 resides in user space and is organized as a collection of C threads [5]. Most threads handle BSD system calls. Among the others are NFS [23] *async daemons*, which handle asynchronous NFS block I/O requests. Since we expect that network file accesses would be the performance bottleneck in a client-server architecture, we prefetch only NFS files opened for read.



(a) Cache Miss Rate



(b) Prediction Accuracy



(c) Prefetch Overhead

Figure 5: **Mechanism Stability.** This figure presents measurements of the prefetching mechanism over a wide range of parameter values. `MATCH_THRESHOLD` was varied from 0 to 1. `PREFETCH_CAPACITY` from 0 (no prefetching) to 50. The measurements were taken for Trace 3 and a cache size of 50.

4.1 Structure

Figure 6 shows the basic structure of the implementation, where each box stands for a C thread and the shaded area constitutes the prefetcher. The prefetcher consists of two pieces of code. The first is the system independent prefetch engine, which handles prefetch analysis, working tree construction and pattern tree maintenance. The same code was used in the simulation. In the implementation, the BSD service threads were modified to provide the prefetch engine information on each `fork`, `execve`, `open`, `chdir` and `exit` system call. The prefetch engine processes this information, makes prefetch decisions and enters the files to be prefetched in a queue. The second piece of prefetcher code is an added thread called the prefetch daemon. The prefetch daemon consumes the file queue and produces block read requests in another queue. These requests are then satisfied by the async daemons. Unlike a user-initiated read operation, a prefetch ends when the block is placed in the system buffer cache. No copy to user space is necessary.

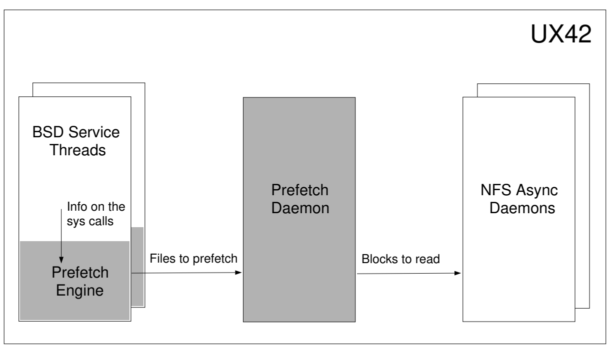


Figure 6: **Structure of Implementation.** A prefetch daemon is added to the collection of C threads in UX42. Each BSD service thread is extended to call the prefetch engine when a relevant system call is serviced.

The prefetch daemon takes advantage of the NFS readahead logic by prefetching only the first block of a file. When a requested block number is one more than the number of the block last read (`r_lastr`), NFS readahead logic speculates that the file is being accessed sequentially, and initiates an asynchronous read on the next block together with the requested block. At least two requested blocks are needed to establish the sequential access pattern before NFS starts readahead. Accordingly, when it removes an entry from the file queue, the prefetch daemon generates a read request for only the first file block (block 0). It also sets `r_lastr` to -1. Later when block 0 is actually accessed, the request will hit in the cache, and, since `r_lastr` is -1, a readahead on block 1 will be issued. This scheme moves on as the file is read block by block, which is the norm. Since the prefetch daemon issues only one block read for each file to be prefetched, the cost of prefetching is minimal.

We have ensured that prefetch I/O yields to regular NFS asynchronous I/O. All the NFS async daemons can be used towards regular I/O, but only up to a certain number of them towards prefetch I/O. Prefetch I/O will be started only if this limit has not been reached and there is no pending regular I/O. Thus, regular asynchronous I/O's can be serviced promptly and when there are too many of these, the prefetcher will refrain from issuing any prefetch I/O's. This ensures that prefetching halts when the system capacity limit is reached, and does not add extra load to an overload.

The implementation consists of approximately 3380 lines of C code. Of these, 90 lines have been added to the existent UX42 source files, 2280 lines are in separate ".c" files and 1010 lines in ".h" files.

4.2 Controlled Experiments

We started the evaluation of the implementation with two simple, controlled experiments. The following questions motivated our experiments:

- How much are the potential benefits of prefetching?
- Under antagonistic circumstances, what is the bearing of prefetching on performance?
- What is the CPU overhead due to prefetching?

The first experiment is a shell script that consists of tens of filter programs, each of which reads one parametric input file, performs some transformation, and writes one output file. Many well known UNIX programs are filters; included in our script are: `awk`, `compress`, `sed`, `sort`, `strings`, `uniq`, `uuencode`, and members of the `grep` family. All input files are the same size and reside remotely.

The second experiment is composed of several program builds. Although builds may seem ideally suited to prefetching, they are antagonistic, for two reasons. First, on our hardware platform a build is CPU intensive, leaving the prefetch mechanism relatively little excess CPU cycles with which to make prefetch decisions. Second, compilations often access header files rapid-fire, meaning that the time between a prefetch decision and the actual need for the

File Size (blocks)	Cache Miss Rate				Latency		
	Buffer Cache		Name Cache				
	No PF	PF	No PF	PF	No PF	PF	Speedup
1	41.67%	0.00%	68.47%	5.87%	20.34 (0.29)	12.21 (0.52)	1.67
2	52.63%	0.00%	68.47%	4.26%	25.92 (0.17)	17.43 (0.53)	1.49
4	30.30%	0.00%	68.47%	4.64%	34.24 (1.18)	24.69 (0.56)	1.39
8	16.39%	0.10%	68.47%	4.45%	46.39 (0.27)	37.47 (0.82)	1.24

(a)

File Size (blocks)	Cache Miss Rate				Latency		
	Buffer Cache		Name Cache				
	No PF	PF	No PF	PF	No PF	PF	Speedup
1	41.67%	1.04%	68.47%	8.14%	23.94 (0.54)	14.33 (0.89)	1.67
2	52.63%	0.88%	68.47%	6.72%	30.07 (1.24)	20.34 (0.66)	1.48
4	30.30%	1.07%	68.47%	6.63%	38.76 (1.10)	28.38 (1.24)	1.37
8	16.39%	0.07%	68.47%	4.92%	54.04 (1.28)	42.94 (1.10)	1.26

(b)

Table 3: **Filters Experiment.** This table summarizes the performance results of the filters experiment. Part (a) presents the results with the 10 Mb/sec wired link; part (b) with the 2 Mb/sec wireless link.

data may not be sufficient to complete the prefetch I/O.

The experiments were run standalone. Both the client and the server are 486 processors with 16MB of memory. The client dedicates 1.57MB to its UNIX buffer cache. Since we are interested to find out how our mechanism behaves in relation to different network bandwidth, we ran the tests using both hardwired and wireless links directly connecting the client and server. The hardwired connection is an Ethernet (10 Mb/sec), while the wireless link is an NCR "WaveLAN" [26] radio link with a maximum 2 Mb/sec data rate. For each combination of experiment and network link, we ran the tests both with and without prefetching. Each number reported below is the mean of three trials.

Table 3 shows the results when the filters experiment is run with a wired link. We varied the work load by using different input file sizes, as given in the "File Size" column. Size is measured in multiples of the server's preferred NFS block size, which is 4KB. For each work load, we list the UNIX buffer cache miss rate as well as the directory name lookup cache miss rate. Since our mechanism deals exclusively with network files, the cache miss rates are those of remote entries. The reduction of the miss rates due to prefetching is substantial. We also present the measurements of application latency, or total elapsed time, which we couldn't do in the simulation. From a user's standpoint, latency is the most important performance metric. The time measurements are in seconds, with standard deviations included in parentheses. The "Speedup" column is the ratio of no-prefetching latency to prefetching latency. The speedups are significant, particularly when the input files are relatively small, since the delay caused by a sequential file access mainly lies in the first one or two block reads. The remaining blocks, if any, will be brought into cache by NFS readahead logic before they are needed.

Table 3(b) shows the results when the same experiment is run over a wireless link. The application latency is larger because of the slower link, but the speedups are comparable to those in a wired setup. The cache miss rates with prefetching are slightly higher over a wireless link than over a wired link since a small amount of prefetch operations cannot be completed in time for demanded use. Nevertheless, the bandwidth of the wireless link is adequate to perform NFS readahead on a timely basis, so it still suffices for the prefetcher to read only the first file block for each prediction it makes.

Figure 7 graphically illustrates the application latency of the filters experiment.

Our second experiment consists of builds of several UNIX utility programs. Table 4 contains the results of these tests. The application latency is also illustrated in Figure 8. When a large number of header files are opened in quick succession — common during compilation — the prefetcher often does not have

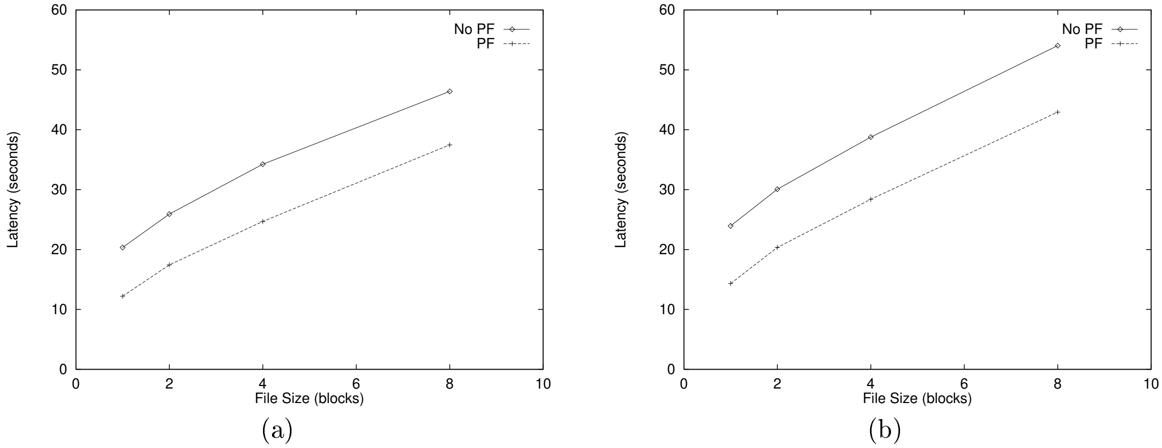


Figure 7: **Filters Experiment: Comparison of Application Latency.** These graphs illustrate the latency data in Table 3. Part (a) compares latency in the wired setting; part (b) in the wireless setting. Prefetching results in significant speedup in this experiment.

enough time or available CPU cycles to make a fruitful prefetch even though it can speculate accurately which files will soon be referenced. The relatively small name cache miss rate improvements confirm this. However, prefetching still manages to reduce the buffer cache miss rate by 65% to 85%. There is no significant latency enhancement, suggesting that the total file read time is not dominant in the application latency. We are pleased to see that no negative effects are observed in an adverse case such as this.

We have used simulation traces to demonstrate the prefetch overhead in terms of wasted network bandwidth and server capacity. The implementation enables us to collect data on another type of cost, extra CPU consumption. Table 5 presents the results for the two experiments. The CPU time includes the system time and user time. Again the time measurements are in seconds and the standard deviations are given in parentheses. The CPU time overhead is negligible in all cases. We also show the ratio of CPU time to application latency (*CPU/latency*). It comes as no surprise that prefetching increases this ratio, substantially in the first experiment. A prefetcher reduces the total elapsed time by increasing the parallelism between CPU processing and I/O.

4.3 Discussion

There are three necessary conditions for prefetching to be useful. First, there must be spare capacity in the whole "data pipe" that extends from server's disk to client's cache. This pipe consists of the client's network I/O interface, the network, the server's CPU, and the server's disk and network I/O paths. Second, there must be system resources on the client side for the prefetcher's use. Of special importance are CPU cycles. The prefetcher cannot fulfill its duty if it doesn't acquire CPU cycles promptly, even though the amount of CPU time needed is low. Third, the workload must allow some interval between file accesses so that the prefetched I/O can be started and completed ahead of the demand.

Prefetching proves feasible on both wired and wireless network connections. Although our experiment results suggest that the speedups for these two situations are comparable, the speedup may be even better appreciated by the client at the end of a wireless link. With the slower link, a job lasts longer and more real time can be saved via prefetching. As the ratio of CPU speed to network speed increases, prefetching should provide more benefit. However, there is some subtlety with our implementation scheme. When the ratio is lowered to a certain point, we will no longer be able to depend on the NFS readahead function to bring in subsequent file blocks quickly enough. This difficulty can be tackled by a simple generalization of our initial approach. The prefetcher can always prefetch the first N file blocks (N growing with the CPU/network ratio) and the NFS readahead code can be modified to read the next N-th file block, instead of the immediately next block. Currently, N is simply set to 1.

Utility	Cache Miss Rate				Latency		
	Buffer Cache		Name Cache				
	No PF	PF	No PF	PF	No PF	PF	Speedup
hash_info	32.09%	5.30%	8.39%	6.92%	21.59 (0.32)	19.53 (0.53)	1.11
snames	17.15%	2.85%	9.03%	7.84%	30.93 (0.44)	29.81 (0.49)	1.04
machid	10.57%	3.17%	10.35%	9.83%	64.11 (1.05)	63.63 (0.68)	1.01
machipc	23.42%	3.36%	8.48%	6.90%	25.48 (0.78)	24.70 (2.51)	1.03

(a)

Utility	Cache Miss Rate				Latency		
	Buffer Cache		Name Cache				
	No PF	PF	No PF	PF	No PF	PF	Speedup
hash_info	32.09%	5.61%	8.46%	6.99%	26.22 (0.66)	24.79 (0.47)	1.06
snames	17.20%	2.96%	9.09%	7.91%	38.17 (1.17)	33.89 (1.64)	1.13
machid	9.15%	3.21%	10.39%	9.86%	77.31 (3.25)	74.42 (2.13)	1.04
machipc	23.27%	4.03%	8.55%	7.02%	29.10 (0.36)	27.17 (0.30)	1.07

(b)

Table 4: **Builds Experiment.** This table summarizes the performance results of the builds experiment. Part (a) presents the results with the 10 Mb/sec wired link; part (b) with the 2 Mb/sec wireless link.

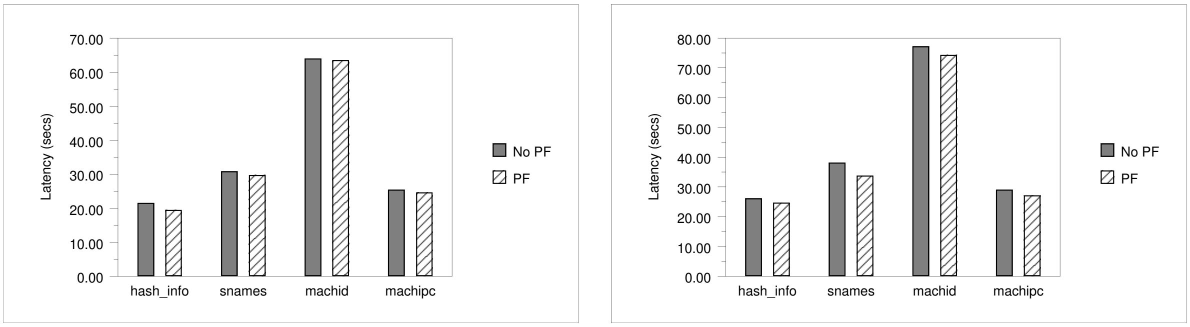
5 Related Work

Prefetching is an old idea. It has been studied extensively in various areas, including prepaging, prefetching of files and prefetching of database objects. Prepaging has not had a major impact in computer architecture because of the tight time and complexity constraints on paging hardware and software. However, prefetching of files and database objects is a more promising endeavor for two reasons. First, since file and database accesses are less frequent than page accesses, the speed with which the decision to prefetch must be made is not so much of the essence. Secondly, the resource most needed to arrange intelligent prefetching, namely client CPU cycles, is the resource most in excess in distributed systems now and in the likely future.

Some researchers have looked into prefetching blocks *within* files. Sequential readahead [7, 17] is the most primitive and yet successful approach. The work of Kotz and Ellis (see, for example, [14]) focuses on the uses of prefetching to increase I/O bandwidth on MIMD shared-memory architectures. Their prefetching methods are geared to the patterns of many scientific and database applications running on multiprocessors. In contrast to these methods, our work looks for access patterns *across* files.

Some file prefetching methods require that each application inform the operating system of its future demands. This includes the TIP project by Patterson *et al.* [21, 22] and the work of Cao, Felton, Karlin and Li [3, 4]. These researchers consider the interaction between prefetching and caching more carefully. TIP uses application-disclosed access patterns to dynamically allocate file buffers between the competing demands of prefetching and caching, based on a cost-benefit model. Cao *et al.* allow applications to pass down both prefetching hints and caching hints. They then employ an integrated algorithm for prefetching and caching, which is shown to be theoretically near-optimal. These "informed" approaches possess an advantage over ours in that prefetching is driven not by deductions made after snooping, but rather by certain knowledge provided in advance by higher levels. There is no danger that disastrously incorrect speculative prefetching might trash the cache. On the other hand, these approaches require re-coding applications. Also, the prefetching mechanism must act within the interval between when the higher level learns of the need to do I/O and when it actually initiates I/O; this interval may not always be sufficient to perform the prefetch I/O.

Like ours, a number of other prefetching methods are completely transparent to clients. They use past accesses to predict future accesses. While we seek to build semantic structures, *i.e.*, access trees, that are endowed with application-level meaning, most of the other approaches use a probabilistic method to model the user behavior. In its most general form, a probabilistic method regards file references as a string of events, and uses information about the last



(a) (b)

Figure 8: **Builds Experiment: Comparison of Application Latency.** These graphs illustrate the latency data in Table 4. Part (a) compares latency in the wired setting; part (b) in the wireless setting. There are no observable negative effects in this antagonistic experiment.

C events to estimate the frequency distributions of the next L events (C and L are called *context order* and *lookahead size*, respectively). For simplicity, however, existing methods have either single context order or single lookahead. Unlike our tree-based approach, there is no attempt to build in an "understanding" of why files are likely to be referenced together.

Some recent examples of work based on probabilistic methods are a paper by Curewitz *et al.* [6] on prefetching objects in an object-oriented database, work by Griffioen and Appleton [9, 10] and by Kroeger and Long [15] that prefetch whole files. Both [6] and [15] adapt context modeling techniques used in data compression to predict the next access. Their work is inspired by the idea that a good data compressor should be able to predict future data well. Griffioen and Appleton's work, in comparison, employs a "probability graph" that for each file accumulates frequency counts of all files that are accessed within the lookahead window.

In the initial stages of our work, we considered probabilistic modeling. We implemented (in the simulator only) two extremely simple probabilistic methods, which we called "stupid pairs" and "smart pairs:"

- **Stupid pairs:** When file F is accessed, prefetch the file that was accessed immediately following F at the last time F was accessed.

  Sample series of accesses: F, G (remember F-G), F (prefetch G, remember G-F), H (remember F-H), F (prefetch H, remember H-F), H (cache hit).

- **Smart pairs:** Keep track of all files that are accessed immediately after F, and when F is accessed the next time, choose one according to the frequency distribution.

  Sample series of accesses (only F's pairs are shown): F, G (F-G1), F (prefetch G), H (F-[G1, H1]), F (prefetch G or H with 1:1 weighting), H (F-[G1, H2]), F (prefetch G or H with 1:2 weighting).

The stupid pair scheme worked better than the smart pair approach when applied to our traces. This unexpected result is explained by considering the locality of many accesses: often, a user works on one file or one group of files for some time, then moves on to similar operations with different files. Stupid pairs are well-equipped to handle this usage pattern, since they invariably prefetch the most recently used successor file. The seemingly superior intelligence of smart pairs actually becomes a liability when locality is strong; files no longer in active use may be given undue weight if they were heavily accessed in the past.

The phenomenon, that less information is better provided that it is more recent, identifies one common weakness of existing probabilistic methods: they don't address the problem of aged information. Instead they make predictions based on a total history of accesses. In addition, single lookahead methods like [6] and [15] are less likely to be widely applicable: as I/O latencies grow in terms of CPU cycles, prefetches must begin ever further in advance if they are to complete in time. On the other hand, single context order methods like that of Griffioen and Appleton fail to make full use of historical file access information, and thus are unable to confidently in-

File Size	Wired Link (10 Mb/sec)				Wireless Link (2 Mb/sec)			
	CPU Time		CPU/latency		CPU Time		CPU/latency	
	No PF	PF	No PF	PF	No PF	PF	No PF	PF
1	3.82 (0.24)	4.50 (0.24)	18.80%	36.87%	3.69 (0.07)	4.37 (0.07)	15.41%	30.47%
2	5.39 (0.71)	5.80 (0.48)	20.81%	33.29%	4.77 (0.09)	5.83 (0.14)	15.86%	28.65%
4	7.88 (0.49)	8.78 (0.79)	23.00%	35.58%	7.39 (0.22)	8.60 (0.35)	19.07%	30.29%
8	13.24 (1.03)	14.45 (0.53)	28.54%	38.58%	13.14 (0.08)	14.58 (0.08)	24.32%	33.95%

(a)

Utility	Wired Link (10 Mb/sec)				Wireless Link (2 Mb/sec)			
	CPU Time		CPU/latency		CPU Time		CPU/latency	
	No PF	PF	No PF	PF	No PF	PF	No PF	PF
hash_info	5.19 (0.21)	5.84 (0.37)	24.03%	29.91%	5.17 (0.11)	5.79 (0.21)	19.72%	23.36%
snames	10.36 (0.67)	11.25 (0.80)	33.48%	37.73%	10.69 (0.10)	11.16 (0.40)	28.00%	32.94%
machid	30.29 (0.92)	32.08 (1.28)	47.24%	50.41%	30.96 (1.82)	32.58 (1.70)	40.04%	43.78%
machipc	6.74 (0.22)	7.63 (0.45)	26.45%	30.90%	6.91 (0.27)	7.80 (0.34)	23.73%	28.72%

(b)

Table 5: **CPU Time Consumption.** This table presents the CPU time with and without prefetching, over two different network links. Also given is the ratio of CPU time to application latency. Part (a) give the results for the filters experiment; part (b) for the builds experiment.

fer accesses far into the future.

Similar to our work, Palmer and Zdonik's work on Fido [20] also explicitly recognizes and maintains access patterns. Several important aspects make their work different. First, their work was conducted in the context of object-oriented database systems, whereas our context is file systems. Second, they represent access patterns with strings of object identifiers, with no semantics involved. We represent patterns with access trees. Third, they employ specialized pattern memory, while we store pattern trees in virtual memory. Finally and most notably, Fido requires a separate training phase after each user session, while our mechanism is more on-line: there is no off-line computation or periodic analysis needed.

An issue that is related to prefetching is hoarding [12, 11, 16, 25]. Both prefetching and hoarding involve anticipatory file fetches: bringing files from remote servers into a local cache before they are needed. These are not exactly the same techniques, however. Hoarding is a scheme designed to increase the likelihood that a mobile client will be able to continue working during periods of total disconnection from file servers. Since hoarding is a relatively infrequent operation performed only at a client's request prior to disconnection, timing is not critical. On the other hand, prefetching is mainly concerned with improving performance and timing is important. With prefetching, the file server is assumed to be still accessible, although the network connectivity may be weak. A cache miss is much more catastrophic in disconnected operations, hence hoarding is typically willing to overfetch substantially in order to enhance the availability of files. Despite the differences, our idea of uncovering and exploiting the semantic structure underlying file references also applies to the hoarding problem, as shown in [25].

6 Conclusion

We have presented a technique for transparent online file prefetching. The technique analytically models interesting system calls and builds semantic structures that capture the intrinsic correlations between file references. It makes accurate predictions of future file accesses, imposes little CPU overhead, defers to demand I/O, and delivers substantially lower client cache miss rates and elapsed time for I/O-intensive applications.

One central trait of the algorithm is that it spends client CPU cycles in return for more effective use of client cache space and fewer on-demand network operations. Another distinguishing aspect is that the algorithm's lookahead ability is potentially much greater than that of previous work. Both of these traits help to couple application I/O performance more closely to CPU speed than to I/O device speed, thereby addressing a fundamental and longstanding problem in operating systems [18].

Our initial performance evaluation has been encouraging. We intend to extend the evaluation along several directions. First, we would like to run experiments that model user behavior more realistically. We are in the process of synthesizing workloads based on actual file system traces. Second, we plan to conduct experiments over a much wider spectrum of network and client capacities. We hope to reach a good understanding on when prefetching is feasible. Third, we would like to examine the applicability of our prefetching algorithm to different kinds of file systems: local file systems, remote file systems with caching of file blocks in main memory only, and remote file systems with whole file caching on client disks.

7 Acknowledgements

This work was supported in part by the Advanced Research Projects Agency, ARPA order number B094, under contract N00014-94-1-0719, monitored by the Office of Naval Research; and in part by the Center for Telecommunications Research, an NSF Engineering Research Center supported by grant number ECD-88-11111.

We wish to thank Carl Tait for many constructive conversations and insightful comments. We also wish to thank the anonymous reviewers and our USENIX shepherd, Carl Staelin, for their remarks.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. 1986 USENIX Summer Conf.*, pages 93–112, June 1986.

[2] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. In *Proc. Thirteenth Symp. on Operating System Principles*, pages 198–212. ACM, October 1991.

[3] P. Cao and E. W. Felten and A. Karlin and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *Proc. 1995 ACM SIGMETRICS*, pages 171–182, June 1995.

[4] P. Cao and E. W. Felten and A. Karlin and K. Li. Implementation and Performance of Integrated Application-Controlled Caching, Prefetching and Disk Scheduling. In *Proc. First USENIX Symposium on Operating Systems Design and Implementation*, pages 165–178, November 1994.

[5] E. C. Cooper and R. P. Draves. C Threads. Technical Report CMU-CS-88-154, Carnegie Mellon University, June 1988.

[6] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical Prefetching via Data Compression. In *Proc. 1993 ACM SIGMOD*, pages 257–266, May 1993.

[7] R. J. Feiertag and E. I. Organisk. The Multics Input/Output System. In *Proc. Third Symp. on Operating System Principles*, pages 35–41. ACM, 1971.

[8] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an Application Program. In *Proc. Summer 1990 USENIX Conf.*, USENIX, pages 87–95, June 1990.

[9] J. Griffioen and R. Appleton. Reducing File System Latency Using a Predictive Approach. In *Proc. 1994 USENIX Summer Conf.*, pages 197–207, June 1994.

[10] J. Griffioen and R. Appleton. Performance Measurements of Automatic Prefetching. In *Parallel and Distributed Computing Systems*, pages 165–170, IEEE, September 1995.

[11] L. B. Huston and P. Honeyman. Disconnected Operation for AFS. In *Proc. First USENIX Symp. on Mobile and Location-Independent Computing*, pages 1–10, August 1993.

[12] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proc. Thirteenth Symp. on Operating System Principles*, pages 213–225. ACM, October 1991.

[13] K. Korner. Intelligent Caching for Remote File Service. In *Proc. Tenth Intl. Conf. on Distributed Computing Systems*, pages 220–226. IEEE, May 1990.

[14] D. Kotz and C. S. Ellis. Practical Prefetching Techniques for Parallel File Systems. In *Proc. First Intl. Conf. on Parallel and Distributed Information Systems*, pages 182–189. ACM, December 1991.

[15] T. M. Kroeger and D. D. E. Long. Predicting Future File-System Actions from Prior Events. In *Proc. 1996 USENIX Annual Technical Conf.*, pages 319–328, January 1996.

[16] G. H. Kuenning. The Design of the Seer Predictive Caching System. in *Proc. Workshop on Mobile Computing Systems and Applications*, ACM/IEEE, pages 37–43, December 1994.

[17] M. K. McKusick, W. J. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for Unix. *ACM Trans. on Computer Systems*, 2(3):181–197, August 1984.

[18] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast As Hardware? In *Proc. 1990 USENIX Summer Conf.*, pages 247–256, June 1990.

[19] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proc. Tenth Symp. on Operating System Principles*, pages 15–24. ACM, December 1985.

[20] M. Palmer and S. Zdonik. Fido: A Cache That Learns to Fetch. In *Proc. 17th Intl. Conf. on Very Large Data Bases*, pages 255–264, September 1991.

[21] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan. A Status Report on Research in Transparent Informed Prefetching. *Operating Systems Review*, 27(2):21–34, April 1993.

[22] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky and J. Zelenka. Informed Prefetching and Caching. In *Proc. Fifteenth Symp. on Operating System Principles*, pages 79–95. ACM, December 1995.

[23] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proc. 1985 USENIX Summer Conf.*, pages 119–130, June 1985.

[24] C. D. Tait and D. Duchamp. Detection and Exploitation of File Working Sets. In *Proc. Eleventh Intl. Conf. on Distributed Computing Systems*, pages 2–9. IEEE, May 1991.

[25] C. D. Tait, H. Lei, S. Acharya and H. Chang. Intelligent File Hoarding for Mobile Computers. In *Proc. First Intl. Conf. on Mobile Computing and Networking*, pages 119–125, ACM, November 1995.

[26] B. Tuch. An Engineer's Story of WaveLAN. In *Proc. First Virginia Tech Symp. on Wireless Personal Communications*, June 1991.