Detection and Exploitation of File Working Sets

Carl D. Tait
tait@cs.columbia.edu

Dan Duchamp
duchamp@cs.columbia.edu

Computer Science Department
Columbia University
New York, NY 10027

CUCS-050-90

Abstract

The work habits of most individuals yield file access patterns that are quite pronounced and can be regarded as defining working sets of files used for particular applications. This paper describes a client-side cache management technique for detecting these patterns and then exploiting them to successfully prefetch files from servers. Trace-driven simulations show the technique substantially increases the hit rate of a client file cache in an environment in which a client workstation is dedicated to a single user. Successful file prefetching carries three major advantages: (1) applications run faster, (2) there is less "burst" load placed on the network, and (3) properly-loaded client caches can better survive network outages. Our technique requires little extra code, and — because it is simply an augmentation of the standard LRU client cache management algorithm — is easily incorporated into existing software.

This work is supported by the New York State Science and Technology Foundation Center for Advanced Technology in Computer and Information Systems, the AT&T Foundation, IBM Corporation, and Hewlett-Packard Corporation.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any of the sponsoring agencies.

1. Introduction

This work investigates how to automatically prefetch files from a file server to a single-user client workstation. Another form of prefetching, prepaging, is an old idea in computer architecture. Prepaging has not had major impact in computer architecture because of the tight time and complexity constraints on paging hardware and software. However, prefetching of files is a more promising endeavor for several reasons. First, since file 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. Further, the penalty for faulty prefetching is not so severe — wasting space in the client's file cache rather than in physical memory. Finally, the resource most needed to arrange intelligent file prefetching, namely client CPU cycles, is the resource most in excess in distributed systems now and in the likely future. In addition, we hypothesize that the work patterns of most individuals yield file access patterns that are quite pronounced and can be regarded as defining "working sets" of files used for particular applications. If the hypothesis is true, then it should be possible to capitalize on the promise of file prefetching.

In Section 2 we describe an algorithm for automatically detecting an application's working set of files as it forms and then prefetching the working set later should the application be executed once again. Our method can in fact detect and exploit multiple distinct working sets generated by different executions of the same application. This capability is important when the same application is executed frequently but with different input; e.g., when a compiler is run on several different but similar modules during a system build. Successful file prefetching carries three major advantages:

1. Applications run faster because they experience a higher cache hit rate.
2. There is less "burst" load placed on the network because prefetching is done in background rather than on demand.
3. Properly-loaded client caches can better survive network outages.

These advantages have extra impact if the client has a small cache and/or if the network is slow. Therefore, we expect our technique to be particularly useful for small portable workstations connected by a radio network, which is slow relative to modern wired networks. Indeed, design of a file system for this environment was the motivation for our work.

While we have not yet implemented our prefetching algorithm in the portable environment (or any other), Section 3 reports encouraging results from simulations driven by real file access traces. Section 3.5 argues that — despite lack of a demonstration implementation — the algorithm is straightforward and quick to implement, and imposes little overhead on regular file system activity. Thus, our technique is desirable (in small-cache environments), effective, and practical.

2. The Algorithm

In UNIX-style operating systems, every program gives rise to a tree of forked (child) processes, and all of these programs access (i.e., open or create) some files. Because the same files may be accessed or executed by many different programs, the graph that results will not necessarily be a pure tree: it may be riddled with cross-edges and back-edges. For ease of expression, however, we will refer to this tree-like structure as “a tree.”

Our algorithm is based on a simple idea: if we save the distinct trees generated by each program, and if we compare trees being built by current activity with such saved trees, then we can detect which saved tree is being executed and prefetch the remainder of its files.

Many subtle problems creep into this seemingly simple algorithm. What happens if the saved tree is too large for the cache? What about the cross-edges and back-edges within “trees” that destroy the pure tree structure? This paper describes how to address such practical details and argues that the resulting algorithm can be effectively used for file prefetching in a distributed file system.

2.1. Data Structure - The Working Graph

As a user does work, a *working graph* is built that reflects current file access patterns. Each file (program or data) is a node in the graph. Conceptually, there are two types of arcs:

1. If program A forks program B, we draw an arc from A to B.
2. If program A accesses or creates file B, we draw an arc from A to B.

Files A and B need not be distinct. In fact, we have observed many cases in which program A accesses itself as a data file. Although (1) and (2) represent distinct arc types, we have found that there is no need to distinguish between these types in the graph. We detect the two cases in different ways, but all edges are considered equivalent once they are in the graph.

In order to reflect the chronology of file accesses, arc order is preserved and we allow multiple arcs to exist from A to B. Consecutive accesses are ignored, but all non-consecutive accesses are preserved for use in a later phase of the algorithm. We do place an upper bound (currently 200) on the number of links between nodes. Subsequent links between the same two nodes are ignored. Since the working graph is constantly being pruned and rebuilt, 200 links are sufficient in virtually every case that we have seen. A form of mark-and-sweep garbage collection is periodically invoked to reclaim dead nodes in the working graph.

See Figure 2-1 for a diagram of the working graph after the following series of accesses:

1. Program A is executed

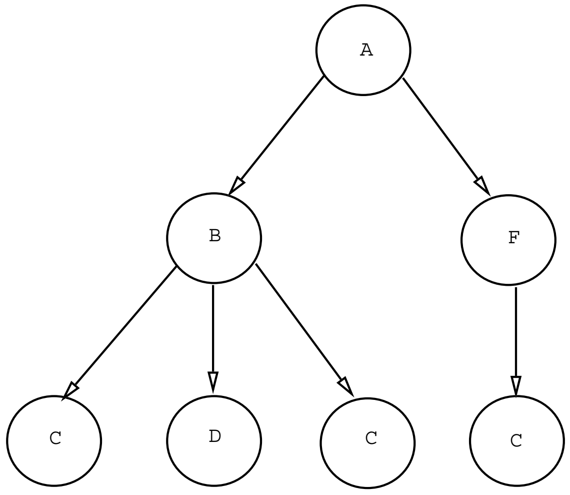

Figure 2-1: Sample Program Tree

2. Program A forks program B
3. B accesses files C and D, in that order
4. B accesses C again
5. A forks F
6. F accesses C

Note that all of the references to file C actually draw arcs to the same node in the graph. Figure 2-1 shows the conceptual structure of the tree, not how it is actually represented in memory.

Because UNIX shells are invoked in such a wide variety of circumstances, they are treated differently from other programs. When a shell uses a file, we draw an arc from the closest non-shell ancestor of the given shell to the file now being accessed or executed. In essence, we draw arcs *through* shells, effectively cutting them out of the working graph. The justification for ignoring shells is that the shell, acting as a command interpreter, is simply an extension of the program that forked it.

Two more special cases need to be mentioned. Temporary files (whose names begin with “/tmp/” or “/usr/tmp/”) are never included in the working graph. By their nature, most of these files are not likely to occur in repeated patterns of computation — prefetching such files would likely be a waste. Further, devices (“files” whose names begin with “/dev/”) are not really conventional files at all, so they also have no place in the working graph.

2.2. Saving and Loading Trees

Whenever a program is executed, the working graph is checked for a tree (formed by a previous execution of the program) rooted at that program. If such a tree exists, it is copied from the working graph onto the top of a stack of trees saved for that program. The links emanating from the program in the working graph are then deleted. Up to ten different trees can be saved for each program. The result of this process is that, when the N+1st execution of a program begins, the tree formed by the Nth execution is unlinked from the working graph and moved to that program's stack. The moment the N+1st execution begins is when the working set of the Nth execution is defined. Notice that while the moment of definition may seem "late," in fact it is the first time at which the working set information could be exploited.

As the N+1st execution begins to form another tree within the working graph, its tree is compared to all trees saved on the stack. If it seems the tree in the working graph is likely to match one of the saved trees, then prefetching of the saved tree is initiated.

In order to choose which tree, if any, to prefetch as a program executes, the algorithm follows a simple guideline: wait until the observed file usage pattern matches only one of the saved trees. If several trees match, wait for more file accesses. If no trees match, do not prefetch for this execution. In order for the forming tree in the working graph to match a saved tree, its initial file references must match exactly: all of the files that the program has directly touched so far must also have been touched by the root of the saved tree.

What happens if the guess is wrong? We may prefetch a saved tree, only to discover that it was a terrible match for the files that were actually accessed next. So when the program executes the next time, we must compare the tree that was actually built with the tree that we prefetched. If the match is "close enough," we discard the saved tree and replace it with the tree we just built; we assume that newer is better, if the match was otherwise reasonable. But if the constructed tree is nothing like the tree we prefetched, we must save both of them on the stack of trees for that program.

The definition of "close enough" is a heuristic that is effective for the data we have seen: if at least 40% of one tree's files are found in the other tree, we deem the match to be successful. This allows for imperfect matches, which is what we want — rarely will two trees touch exactly the same files.

Figure 2-2 contains an example from one of our traces. There are two trees rooted by "cc," the UNIX C compiler. The first tree is responsible for creating object files; the second links object files into executables. They both start the same way: cc executes, and then reads a file containing language information. Our algorithm waits until the set of files that the current cc execution touches is found in exactly one of the saved trees. If cc should access some previously unknown file before a tree is chosen, we give up on prefetching for cc on this execution — apparently there is no matching

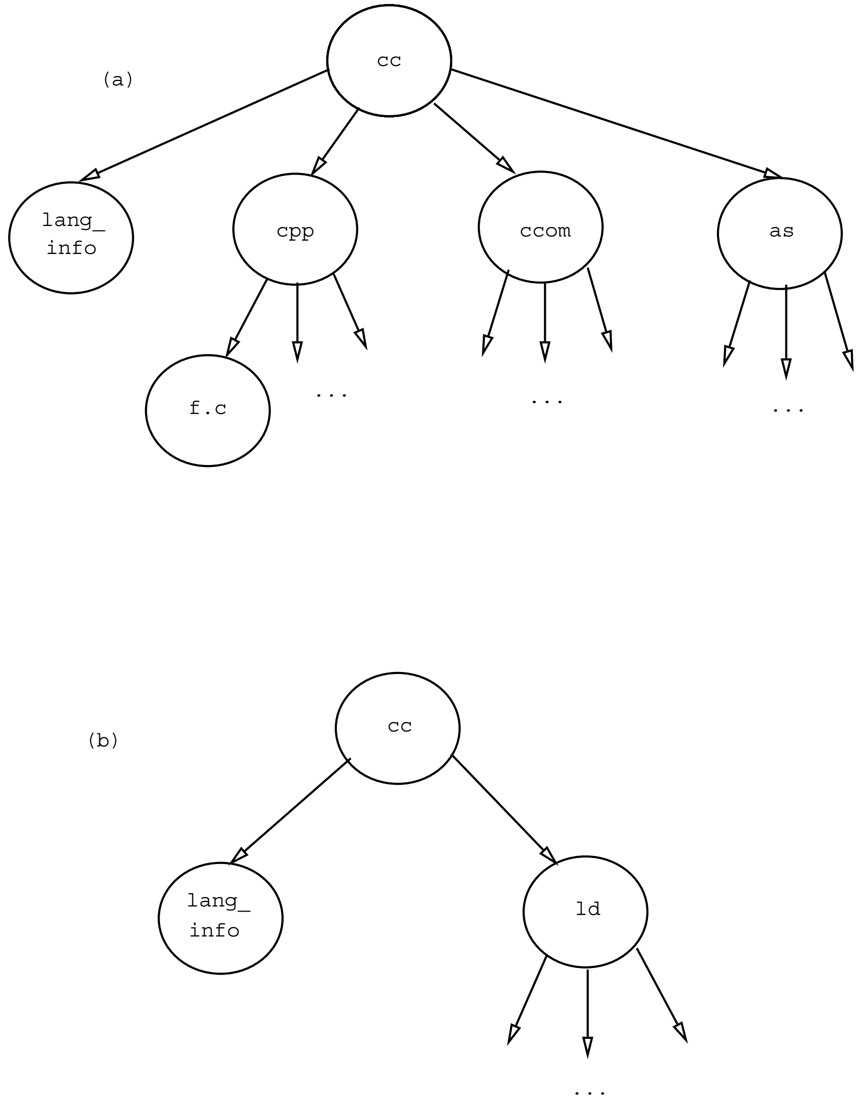

Figure 2-2: Two Different ‘cc’ Trees

tree.

Note that we do not consider access order important here. If, for some reason, cc had touched ccom before doing anything else, tree (a) would have been selected on the spot.

It is important to note that this entire process is repeated for each program within the tree that we have loaded. The idea is that each program prefetches its own tree, and the tree thus selected may or may not be a subtree of the larger tree we prefetched earlier. This effectively allows minor prefetch corrections to be made at every level in a tree, which reduces the cost of a bad guess at a high level. It also means that there is no concept of “the tree currently being prefetched”: the prefetch set is constantly being modified as each constituent program is executed. That is, there is no coupling needed between the working graph and the saved trees; this fact substantially eases implementation.

2.3. Tree Splitting

If the prefetched tree is too big to fit into the cache, we resort to a tree-splitting scheme. First, we step through the tree in preorder — the order in which files are likely to be used — until we have found a subtree that will fit comfortably into the cache.

Now for the hard part. Stepping through the tree again, we take all of the nodes and subtrees that were severed from the original tree, and point them directly to a single new root, thus creating a single subtree to be saved. (This is actually a form of path compression.) When a file is accessed that has one of these pointers to a saved subtree root, we load the subtree, splitting it again if necessary. Note that this does not take precedence over the tree-selection scheme described earlier; we load saved subtrees only when there is nothing else we can do.

Figure 2-3 shows how the tree in Figure 2-1 is broken up when using a cache that can hold three files. Note that D and C are pointed directly to the new root due to path compression.

2.4. Common Prefix Trees

Assuming that there is no unique saved tree to load, and that there is no split subtree from a previous load available, then we immediately compute and prefetch a *common prefix tree* for the program. The prefix tree is a 2-level tree whose root is the program itself; the children are the root’s file accesses that are common to all the saved trees. We require strict matching in this case: if the root of our prefix tree has N children, this means that the first N children of the root in every saved tree are identical.

We added the prefix tree feature to our algorithm after discovering that oftentimes the first few file accesses of all saved trees (i.e., the common prefix) are the same. In this case, the unoptimized algorithm would typically miss on every file access in the prefix, despite the fact that the prefix is perfectly predictable.

Once the prefix tree is computed, it is loaded like any other tree, and then destroyed. To avoid the problem of tree splitting, our construction ensures that prefix trees are never too large for the cache: extra files in the common prefix are ignored.

2.5. Cycle Trees

A case requiring special attention is that of long-running programs that establish repeated file access patterns within a single execution. The basic algorithm cannot handle this case at all, since the tree-matching code is triggered only when a program

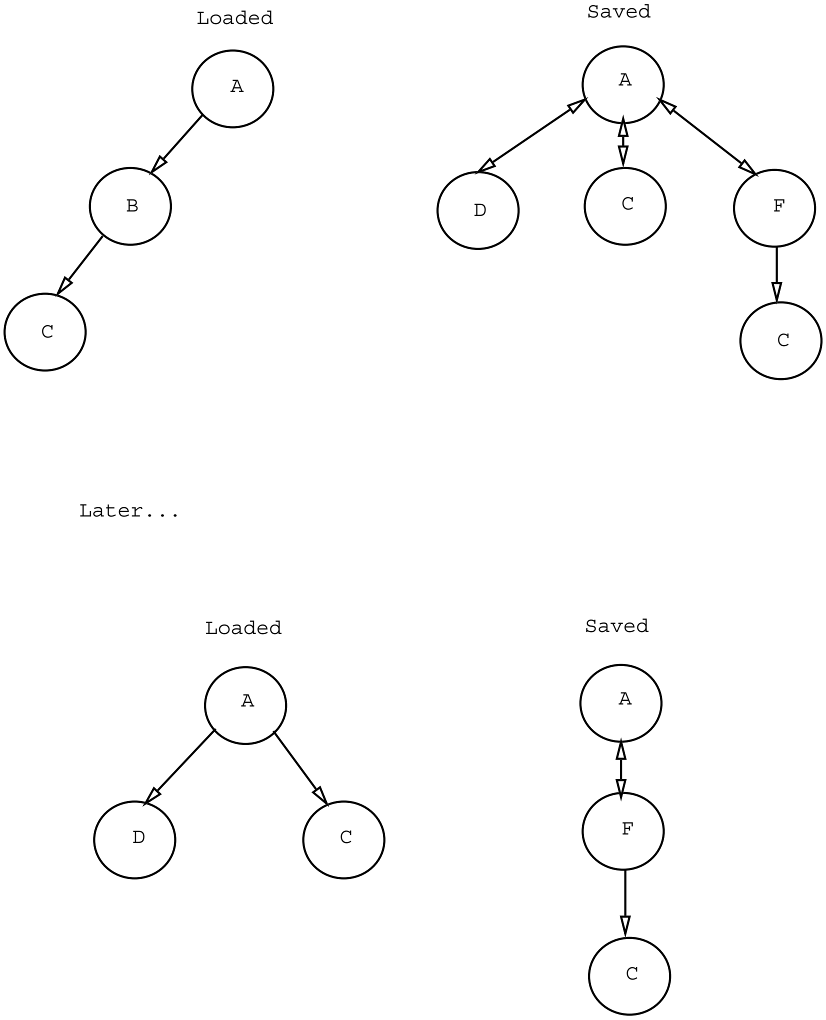

Figure 2-3: Tree Splitting

begins an execution.

To get around this problem, we use *cycle trees*. Such a tree consists of a cyclic pattern of file accesses that has been detected within a running program. Our algorithm checks for cycles whenever it cannot load a tree of any other type. It compares the two most recent access patterns that begin with the file just touched, and builds a 2-level tree that includes the longest common prefix of the two patterns.

Cycle trees are very much like common prefix trees. They include only the immediate children of the program in question; they contain exactly those file accesses common to previous executions or access patterns; and by construction, they are never too large for the cache. They are loaded like any other tree, and destroyed immediately. Figure 2-4 shows how a cycle tree is constructed from a program's pattern of file accesses.

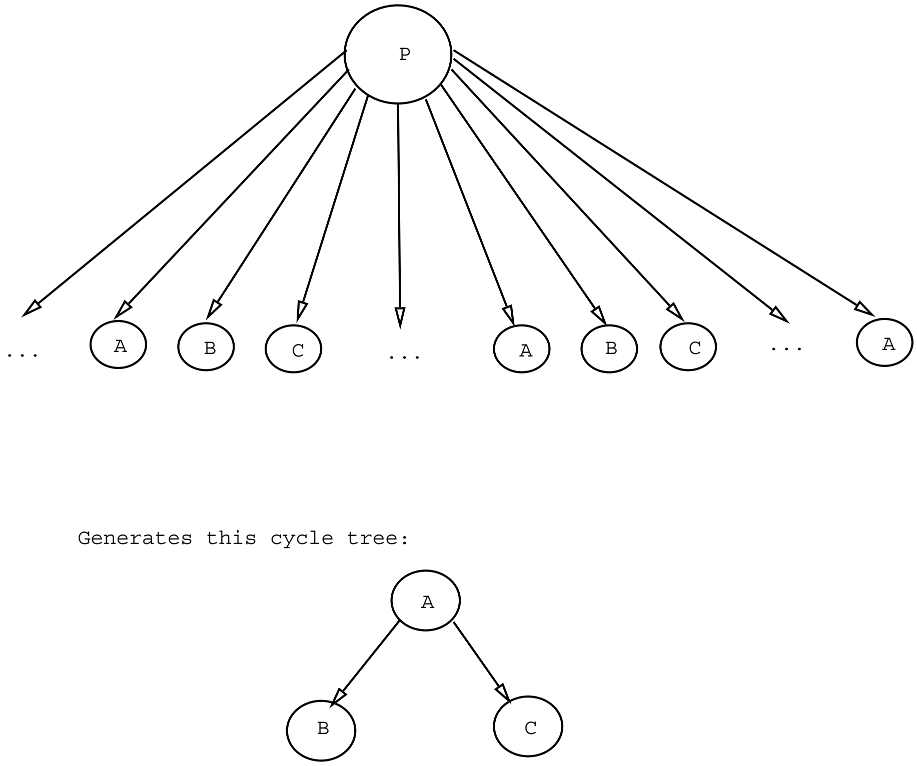

Figure 2-4: Cycle Tree Construction

2.6. Prefetch Confidence

One final point about loading trees of any sort: in order to avoid destroying the entire cache due to a bad prefetching guess, we never fill more than 80% of the cache when loading a tree. The figure of 80% for “prefetch confidence” is arbitrary. Varying it has little effect on the hit rate unless the number is made small.

3. Results

Our results were obtained by trace-driven simulation.

3.1. Trace Data

Traces were gathered on a Sun 3/80 running 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 large traces of a single user’s file access activity. The monitored operations were those for process creation (`fork`, `execv`/`execve`) and file access and creation (`open`, `creat`, `mkdir`). These are the operations needed to build the working graph.

The traces report the file access activity of a volunteer user performing his normal work activity over a period of two weeks. The first trace contains 10,182 events captured over 33 hours. The second trace contains 12,472 events captured over 72 hours, while the numbers for the third trace are 25,440 and 86, respectively. During these hours activity varied widely and included compilations, document production, data analysis and display, large searches for certain files (i.e., UNIX “find” commands), Usenet news reading, printing, and other operations.

3.2. Simulation Methodology

The task of the algorithm simulator is to step through the trace files, managing its cache in accordance with the graphical machinations prescribed by our algorithm. As a basis for comparison, the simulator manages a separate cache using a simple LRU replacement policy at the same time. Since our trace data provides no information concerning file sizes, we simply define cache size by number of files; for example, 20 files, not 200 kilobytes.

In SunOS 4.0.3, five particular files involved in the dynamic linking process are opened by every program that executes. These files are accessed so frequently that any reasonable method would certainly keep them locked in the cache. Since we want to see how our method compares with LRU in the cases where the two algorithms might reasonably be expected to differ, we filtered these heavily-used files out of our trace data — i.e., *before* the simulation. In order to present fair comparisons between our algorithm and LRU despite this adjustment, we report *miss ratio* rather than hit ratio. The reason is that of the three significant numbers summarizing a simulation run (number of hits, number of misses, number of total accesses), only the number of misses is unaffected by removing certain accesses that are assumed to always hit in the cache. So the ratio of miss ratios of the two algorithms remains the same despite the adjusted data (because the ratio of miss ratios is simply the ratio of misses).

The current and parent directory files (‘.’ and ‘..’) are also accessed very frequently. Although these names do not represent fixed files, it is a simple matter to cache the current and parent directories, and so these file references are filtered out as well. We are thus left with the interesting cases that provide the most reliable measure of our algorithm’s effectiveness vis-a-vis LRU.

3.3. Simulation Results

We expected that the increased intelligence of our method would be more effective versus LRU in smaller caches. After all, in the extreme case of an infinitely large cache, it doesn’t matter what “replacement” policy one uses, since no files are ever replaced. The “optimal” column in Table 3-1 shows the miss ratios that one would obtain in such a situation.

Bearing that in mind, we ran our simulator using cache sizes of 20, 40, and 100 files. We kept track of the cache misses for both LRU and our tree-based algorithm. A summary of the results for the three traces appears in Table 3-1. The tree-based algorithm has a substantially better miss ratio than LRU with a cache size of 20 files. Even with larger caches, however, the tree method out-performs LRU.

TRACE	CACHE SIZE	LRU	TREE	OPTIMAL	LRU/TREE
1	20	58.9%	39.1%	18.0%	1.51
	40	51.3%	36.1%		1.42
	100	34.2%	29.6%		1.16
2	20	67.2%	45.5%	14.9%	1.48
	40	51.2%	40.7%		1.26
	100	37.8%	36.9%		1.02
3	20	55.5%	41.2%	13.3%	1.35
	40	43.9%	36.8%		1.19
	100	33.4%	31.6%		1.06

Table 3-1: File Cache Miss Rate

In practice, no algorithm can approach the optimal miss ratio because of the repeated `find` operations.

In addition, we monitored the burst hit ratios: how well each method did over each burst of 512 accesses. At the end of each run, we checked to see which method had won the most bursts. As shown in Table 3-2, the tree method won this comparison easily and consistently. This shows that the tree algorithm's superiority over LRU is steady and stable, and not simply the result of a few exceptionally fruitful prefetch sequences.

Ties at the burst level are attributable to a large number of new files suddenly being brought into the cache in both methods — as will happen with a large `find`, for example. The tree method's few losses due to bad prefetching guesses are more than offset by its many wins.

TRACE	CACHE SIZE	WINS	LOSSES	TIES
1	20	18	0	1
	40	18	0	1
	100	13	5	1
2	20	20	1	3
	40	18	2	4
	100	14	5	5
3	20	44	1	4
	40	37	6	6
	100	31	10	8

Table 3-2: Tree vs. LRU over Bursts of 512 Accesses

Finally, we measured the miss rate during those periods immediately following a tree prefetch. After loading a tree of size N, we monitored the miss ratios of both methods for the next N-1 accesses (the root was not counted since it initiated the prefetch). Table 3-3 shows the results of these measurements. As desired, the miss rate for the tree method is quite low in the periods following prefetches, indicating that our algorithm is effective in choosing which files to prefetch.

TRACE	CACHE SIZE	LRU	TREE	LRU/TREE
1	20	56.9%	15.5%	3.66
	40	44.8%	13.6%	3.31
	100	18.1%	8.8%	2.06
2	20	62.5%	12.8%	4.87
	40	34.0%	9.1%	3.74
	100	10.4%	8.1%	1.28
3	20	45.9%	17.7%	2.60
	40	26.2%	13.7%	1.91
	100	12.9%	9.5%	1.36

Table 3-3: Miss Rates following Tree Prefetches

One important note. We assume that prefetching is done in the background while the user makes use of the single file that triggered the prefetch. Hence, none of the prefetched files is ever treated as a cache miss. In reality, the user might need some of these files before the background prefetch has brought them into the client cache. We do not anticipate that this will be a significant problem; however, only a working implementation will tell us with certainty.

The entire simulator consists of approximately 2200 lines of C. Table 3-4 gives the name, purpose, code size, and approximate running time of each of the major pieces of the simulator. Several of these routines are called quite infrequently, so two timing figures are given: milliseconds per call, and average milliseconds per file access when calling frequency is taken into account.

CODE SECTION	LINES	CALL	AVG.
Add links to working graph	200	0.09	0.09
Compare graph to saved trees, decide to prefetch	470	1.45	1.45
Determine if prefetch succeeded	90	1.21	0.06
Split saved tree into cache-sized pieces	190	2.56	0.02
Garbage collection of old graph nodes	190	144.09	0.04
Miscellaneous utility routines	540	---	---
Simulator skeleton - not part of the algorithm	380	---	---
Declarations & constants	110	---	---
TOTAL	2170	---	---

Table 3-4: Size and Running Time of Code Sections

Determining if a prefetch succeeded occurs about 5 times in every 100 accesses. Tree splitting is rarer, and is only required about 75 times in 10,000 accesses. Garbage collection is performed once every 4096 accesses. The simulation was timed on a Sun 4/490, a 22 MIPS machine.

3.4. Limitations

Our algorithm depends on a UNIX-style model of process forks in order to build trees. Fork information is vital to our method, since it is used to define a hierarchical chronology of file accesses, leading to the use of tree-based working sets of files. Our dependence on fork information mandates that the algorithm be run on the client side. This is not really a problem: in the interests of scalability, it is certainly desirable to minimize the use of server resources.

Certain file access patterns limit the effectiveness of our algorithm. In trace #1, more than half of all misses were caused by a data analysis program that, across repeated executions, accessed different subsets of files within the same directory or else accessed the same files but in a different order. Our simulator deals poorly with this case. In a working implementation, however, this problem could be reduced or eliminated by prefetching entire directories; measurements we have made indicate that, if a user’s directory is used at all, then a very high percentage of its files are accessed.

Our simulator is implicitly based on whole-file caching. This is perhaps acceptable, since studies indicate that the great majority of files are read in their entirety [4, 7]. Furthermore, there is a minor trend toward file systems that use whole-file caching

[5, 10]. Very large files and shared files should remain at the server, however, and we have not yet addressed the problem of how to deal with these files.

Finally, we would like to gather more trace data from different environments. The traces that we are currently using, however, provide encouraging evidence for the effectiveness of our algorithm.

3.5. Implementation Issues

An implementation of our algorithm would function quite like the simulation does. That is, the data structures described above would be maintained within the file cache manager, and the same analyses of these data structures would take place at the same times. The most common data structure occupies about 70 bytes, and thousands of these can be created during long runs. On our particular data, garbage collection had the effect of inducing an equilibrium on the number of nodes in existence, with the result that total program size remained relatively constant around 300KB. Smaller caches required more program space than large ones due to the greater number of trees that had to be split.

Note that the prefetch recommendations produced by analysis of the working graph can and should be treated as a hint. That is, the cache management policy remains LRU. Either the working graph or the stacks of saved trees could at any time be reduced or thrown away, with the only negative effect being a reduced hit rate, bounded below by the hit rate that LRU would produce by itself. Thus, the algorithm is not sensitive to any particular choices for the size of data structures, and performance and resource use can be traded off against each other as necessary. Nevertheless, it is desirable to maintain as much of this information for as long as possible. To avoid incurring the "learning curve" costs after every new login or reboot, the data structures could be checkpointed to disk during idle periods.

The cache manager would need no extra mechanism to perform prefetching, although it would be advantageous if the interface between client cache and server were augmented to include a way to ask, in one call, for many files to be prefetched.

A disadvantage of our algorithm is that it requires information from two parts of an operating system that are increasingly being separated: process management and the file system. Fortunately, the information needed by the cache manager is minimal: the user id of any individual whose file access behavior is being "snooped," and notice of every `fork` and `exec` call made by any of that individual's processes. Since our algorithm only provides hints, this information could be batched and/or provided late to the file system, with reduced performance the only cost. The other information needed to construct trees — namely the user id, and process id of each `open`, `creat`, or `mkdir` — is typically available within a file system implementation, where it is needed to check permissions and establish open file tables.

The only way in which a user needs to be involved at any point in the process is to turn on the cache manager's snooping. Thus, a system call should be made available to turn snooping/prefetching on and off.

In summary, an implementation might include the following changes to a simple LRU cache manager:

1. The interface between process manager and file system should include a way for the process manager to pass parent/child information to the file system.
2. The user interface to the file system should include a call to turn snooping/prefetching on or off.
3. The client/server interface within the file system should include a way to specify prefetching of several files at one time.
4. The client cache manager should be able to checkpoint its data structures.

The first two changes are essential, the third change is desirable, and the last is useful.

4. Related Work

File prefetching is not a new idea. Once design of distributed file systems reached a basic level of sophistication, designers began to contemplate increasing client cache hit rates through various means that can all be labeled as "prefetching." Also, the infrequent but annoying loss of file servers led to the related desire to make client sites more independent of their servers. Previous practical work on the topic is discussed below. We argue that our work has substantial advantages over previous work in this direction.

An early discussion of the concept of *stashing* occurred in a 1988 workshop [3, p. 16]. The idea is simple: to gather at the client most or all of the files it will need in the near future so as to survive communication failures. A very large cache would help in this, but clearly anticipatory fetches play a role as well.

Shortly afterward, Alonso's group listed a number of stashing methods that they contemplated for inclusion in their Face file system [1, 2]. In order of increasing sophistication, they are:

1. Make users responsible for which files are stashed. Various methods were proposed. One is enumerating the files that are to be stashed a ".stashrc" file. Another is making an explicit "stash" command available to the user. The last is to have the user tell the system to record and stash all files used between two selected points in time.
2. Have each application stash what it needs.
3. Engineer the file system to read and understand certain key files that indicate which other files are needed to perform a task; e.g., "'makefiles."
4. Have the system stash files used in the last several commands given by a user.

The notions of performing per-application stashing (assuming that stashing is implicit in opens, rather than requiring the application to be altered to include some ‘stash’ call) and of saving the result of the last few commands are both like our idea. But the Face report does not explain how either idea might be achieved.

We regard a stash mechanism that depends on users for correct and timely information or action to be unsatisfactory, for two reasons. First, many users will refuse to do the necessary work on the regular basis required to keep a stash primed with the correct contents. Second, users will be unable to fully describe what should be in the stash. In some cases users will forget that certain input files are required; more often, users will be ignorant of certain files required by the *implementation* of an application, which is necessarily unknown to them. It is the task of modern file systems to remove this sort of burden from the user.

The proposal to use specification files (like “makefile” and “.Xdefaults”) as a guide to stashing is intuitively appealing, but limited. First, only so many applications employ such files. Second, usually these files specify which other files are needed as input, but once again cannot know which files the application’s implementation will make use of.

Coda [8] is a highly available file system that, as part of its approach to availability, uses a brute-force sort of prefetching. Coda’s cache management software performs “volume fetching” — an entire volume is brought into a user’s cache whenever that user begins his work. A volume is a portion of the file system that in Coda forms an administrative unit. In the Coda environment, a user’s files are typically all placed in a single volume, and so Coda’s prefetching scheme boils down to retrieving all of a user’s files at one time. This sort of prefetching has two obvious disadvantages: it depends on the existence of a (very) large client cache, and the method is tremendously wasteful. Our own studies (and some published studies [9]) have shown that an overwhelming percentage of a user’s files have not been touched within the preceding month.

Though wasteful, volume prefetching clearly does ensure that a user’s personal files are never inaccessible to his workstation. The other half of Coda’s high-availability strategy is to widely replicate read-only copies of “system volumes.” Thus provided that all its assumptions about the operating environment are met, Coda does reliably give access to most needed files. To summarize, the Coda assumptions are:

- Applications rarely need files lying outside both the user’s volume and the system portion of the file space that has been made read-only.
- Applications can function properly with read-only copies of system files.
- Non-read-only sharing of user files is uncommon.
- Network connectivity permits constant contact with a system-file volume.
- The client cache is large enough to accommodate the whole volume of its user(s) as well as any other needed files.

The work most closely related to our own is Korner's work on detecting and exploiting the *block access patterns of individual files* [6]. The "style" of this work is similar to ours in that patterns are detected and exploited. However, several important aspects make Korner's work different:

1. The detection of patterns is an off-line process performed by two expert systems, whereas our method provides on-line real-time detection and exploitation.

2. Whereas our method is an adjustment to LRU, many of the access patterns Korner discusses (e.g., MRU) are alternatives to LRU; thus, a cache manager would require substantial re-coding to benefit from the noted access patterns.

3. Korner's objective is the more limited problem of detecting and exploiting the block access patterns within a file, not file access patterns within an application. Since a large proportion of files are accessed in their entirety [7], it is not clear what percentage of files could benefit from Korner's prefetch rules. Our method benefits any application whose file access behavior is predictable.

In addition, Korner's proposed exploitation is for the file server's in-memory cache of disk blocks, not for a client-side cache of server contents. Below we propose what might be a better use of Korner's techniques.

Korner's method would seem to provide impressive speedups in prefetching file blocks; however, the method offers no help in prefetching future files. But Korner's work can complement ours in two ways. One way would be to use our file prefetching method in the client (file) cache while using Korner's technique for the server-side (block) cache. Better yet would be to couple the techniques for use in "remote open" file systems that move blocks, and not files, between client and server. In this case, a client-side block cache would use our technique to determine which files should have blocks in the cache, and Korner's technique would pick the right blocks of those files.

5. Conclusion

We have presented an algorithm that detects and exploits file working sets. It is used to add prefetching intelligence to the basic LRU cache management strategy. The technique is applicable to any UNIX-style system, is independent of the cache consistency algorithm, imposes little overhead, can easily be added to existing standard software, and simulation suggests that — especially for small caches — it is effective in improving the client cache miss rate that would be obtained by LRU alone. Decreasing cache miss rate speeds applications and renders clients more independent of file server loss.

Simulation results must always be interpreted carefully, and we have identified a total of five ways in which our simulation does not accurately mirror the events of a plausible real-life implementation:

1. References to a few commonly-used files are filtered from trace data.

2. References to certain other commonly-used files are ignored by our algo-

rithm. Thus files in /tmp hit sometimes when the LRU method is used, but rarely when our method is used. Names of files in /tmp are highly structured, and it would not be hard to distinguish per-process temporary files (certain to never again be used) from per-session temporary files (likely to be used).

3. It is assumed that prefetching can be accomplished in background without cache misses.

4. Our tree-based algorithm suffers from a learning curve. Results not included in Section 3 suggest that, for our traces, the learning-curve penalty raises our algorithm's miss rates by up to 2%.

5. Cache size is measured in files, not bytes.

Simulation inaccuracies (1) and (3) tend to over-value our algorithm; inaccuracies (2) and (4) tend to under-value it.

The promise shown by the simulations and the existence of these inaccuracies motivate us to start an implementation. The algorithm's trait of spending client cycles (and, to a lesser extent, client memory) in return for more effective use of client cache space and fewer on-demand network operations leads us to believe it would be most effective in an environment of radio-connected "notebook" workstations. This environment is characterized by these features:

- Limited client cache space (because of constraints on size, weight, and power consumption).
- Plenty of client cycles (because of microprocessors).
- A slow network (present commercially-available state-of-the-art is only 2Mb/sec).
- Occurrence of transient network outages — "fades" occurring in radio null spots.

With the strong industry trend toward portable "notebook" computers, we expect the portable, wireless environment to be increasingly common and important. Our algorithm seems to offer especially impressive advantages in these circumstances.

5.1. Future Work

Our prefetching technique depends on repeated executions of application programs exhibiting similar file reference behavior. We suppose that this condition is more likely to be satisfied on a personal workstation than on a timeshared machine, and that is why we have aimed our current effort toward the case wherein the client is a single-user workstation. It is an open question how much our technique would help or hurt the file system of a many-user machine.

A more substantial concern for the future is that the algorithm as it now stands has a limited definition of what constitutes a "computation" — namely, a tree of forked processes. The strong trend in operating systems is toward providing a multiplicity of ways to spread a computation across processes: message passing between clients and servers, shared memory, multiple threads of control within a process, etc. As this trend continues, it will become harder to track all the locations and actions of a computation. This uncertainty will confound our method. Consider the difficulty of tracing the file accesses of a computation that is accomplished partly by calling a multi-threaded server on another machine. We expect the tracking and control of distributed computations (i.e., the distributed computing equivalent of ‘‘network management’’) to be an area of research in the near future.

Acknowledgements

Bill Schilit’s labor produced the traces used in our study.

References

[1] R. Alonso, D. Barbara, and L. L. Cova.
Face: Enhancing Distributed File Systems for Autonomous Computing Environments.
Technical Report CS-TR-214-89, Princeton Univ., March, 1989.

[2] R. Alonso, D. Barbara, and L. L. Cova.
Augmenting Availability in Distributed File Systems.
Technical Report CS-TR-234-89, Princeton Univ., October, 1989.

[3] A. Birrell.
Position given in *Autonomy and Storage* section of Third European SIGOPS Workshop.
Operating Systems Review 23(2):3-19, April, 1989.

[4] R. A. Floyd and C. S. Ellis.
Directory Reference Patterns in Hierarchical File Systems.
IEEE Trans. on Knowledge and Data Engineering 1(2):238-247, June, 1989.

[5] J. H. Howard, et. al.
Scale and Performance in a Distributed File System.
ACM Trans. on Computer Systems 6(1):51-81, February, 1988.

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

[7] J. Ousterhout, et. al.
A Trace-Driven Analysis of the UNIX 4.2 BSD File System.
In *Proc. Tenth ACM Symp. on Operating System Principles*, pages 15-24. December, 1985.

[8] M. Satyanarayanan, et. al.
Coda: A Highly Available File System for a Distributed Workstation Environment.
IEEE Trans. Computers 39(4):447-459, April, 1990.

[9] A. J. Smith.
Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms.
IEEE Trans. Software Engineering SE-7(4):403-417, July, 1981.

[10] C. Staelin and H. Garcia-Molina.
File System Design Using Large Memories.
Technical Report CS-TR-246-90, Princeton Univ., June, 1990.