Appears in *ACM Operating Systems Review*, V 27(2), April, 1993, pp.21-34.

A Status Report on Research in Transparent Informed Prefetching

R. Hugo Patterson†, Garth A. Gibson, M. Satyanarayanan**
†Department of Electrical and Computer Engineering
**School of Computer Science*
Carnegie Mellon University, Pittsburgh PA 15213
email: rhp@cs.cmu.edu

Abstract

This paper focuses on extending the power of caching and prefetching to reduce file read latencies by exploiting application level hints about future I/O accesses. We argue that systems that disclose high-level knowledge can transfer optimization information across module boundaries in a manner consistent with sound software engineering principles. Such Transparent Informed Prefetching (TIP) systems provide a technique for converting the high throughput of new technologies such as disk arrays and log-structured file systems into low latency for applications. Our preliminary experiments show that even without a high-throughput I/O subsystem TIP yields reduced execution time of up to 30% for applications obtaining data from a remote file server and up to 13% for applications obtaining data from a single local disk. These experiments indicate that greater performance benefits will be available when TIP is integrated with low level resource management policies and highly parallel I/O subsystems such as disk arrays.

1 Introduction

Today, file read latency is the most significant bottleneck for high performance input and output. Other aspects of I/O performance benefit from recent advances in disk bandwidth and throughput resulting from disk arrays [Kim86, Salem86, Livny87, Patterson88, Reddy89], and in write performance derived from buffered write-behind and the Log-structured File System [Rosenblum91]. The development of distributed file systems operating over networks with diverse bandwidth [Spector89, Satyanarayanan85, Nelson88] only exacerbates the problem. In this paper, we argue that prefetching based on application level information is a feasible and effective strategy for reducing file access read latency in both local and network file systems.

This material is based (in part) upon work supported by the National Science Foundation under grant number ECD-8907068, the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597, an IBM Corporation Research Initiation Grant, and a Digital Equipment Corporation External Research Project Grant. 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 the funding agencies. The government has certain rights in this material.

This paper presents *Transparent Informed Prefetching (TIP)* as a powerful and flexible mechanism promising to reduce application execution time in two ways.

1. By exposing concurrency in the I/O workload, TIP can
    a) service multiple I/O requests concurrently and convert the high throughput of disk arrays and other new peripheral technologies into low read latency,
    b) overlap I/O with computation or user think time, and
    c) optimize I/O accesses over the larger number of outstanding requests.
2. With knowledge of future I/O requests, TIP can make informed cache management decisions that lower cache miss rates.

TIP derives its power from hints given by application levels of the system that disclose future access patterns rather than advise lower-level policies or actions. Such disclosures should be made early, as soon as knowledge becomes available. Lower levels of the system use the hints to transparently prefetch data and improve resource management, but are free to ignore or defer acting upon any hint.

In addition to a discussion of TIP, this paper reports the results of preliminary experiments intended to demonstrate potential benefits and obstacles of TIP. These experiments show a 13% reduction in the execution time of the *make* of an X windows application with ideal hints accessing a single disk, and a 20% reduction in the same program's execution time when accessing data remotely in the Coda distributed file system [Satyanarayanan90]. A second example, the *grep* text search for a simple pattern in 58 files stored remotely in Coda, achieves a 30% reduction in its execution time when the shell issues the command arguments as a hint in parallel with initiating the search. These results were obtained on systems with only a single disk. Greater benefits are expected on systems with the higher throughput of a disk array.

Our argument in favor of TIP begins with a review of the increasingly important I/O bottleneck and existing mechanisms for combatting it. Then we introduce the TIP approach, describe its benefits and discuss its dependence on hints that disclose instead of advise. After describing our experiments and results, we give an overview of our research plans and opportunities for further study of transparent informed prefetching and finish with a discussion of related work.

2 Technology Trends

2.1 The I/O Bottleneck

In recent years, systems researchers have begun to highlight the need for improved I/O performance. The speed of computations using only primary memory is increasing five to ten times faster than the speed with which blocks on magnetic disk can be accessed [Patterson88, Ousterhout89]. In addition, some new technologies further increase the gap between processor and I/O performance. Distributed and wide area [Cate92] file systems slow I/O with substantial network transmission and server delays. Portable computers, which frequently spin down their disks and are only weakly connected to networks, suffer additional latencies [Kistler92]. Applications that depend on the massive storage of optical disk jukeboxes or robotic tape libraries must wait while the desired media is fetched and mounted. Amdahl's Law [Amdahl67] tells us that unless I/O subsystem performance keeps pace with improvements in the rest of the system, I/O will increasingly constrain the performance of the system as a whole.

	Latency	Throughput
Read	demand caching prefetching	disk arrays
Write	buffered writes LFS	disk arrays buffered writes LFS

Table 1. Range of Solutions. This table shows the mechanisms most effective in combating the growing I/O bottleneck. Write-behind buffering and log-structured file systems (LFS) eliminate write latency for many applications and help throughput. The parallelism of disk arrays increases both read and write throughput. Unfortunately, the long read latencies of slow peripherals are only partially masked by caching and prefetching. This paper focuses on extending the power of caching and prefetching by exploiting hints from high-levels of the system to more effectively reduce read latency.

2.2 Range of Solutions

On the bright side, a number of innovations have relieved certain aspects of the I/O bottleneck. Table 1 summarizes the most commonly used techniques to address the throughput and latency components of both read and write I/O performance.

Redundant Arrays of Inexpensive Disks (RAID) take advantage of steadily decreasing disk diameter and cost per byte to address I/O throughput [Gibson91]. They provide parallel transfer to speed servicing of large requests. They can also service many small requests simultaneously to provide high service rates for workloads consisting of many concurrent, randomly distributed, small accesses. RAIDs do not perform quite as well for small writes as they do for reads, but the Log-Structured File System (LFS) [Rosenblum91] can help by combining small writes into large ones that RAIDs handle effectively. Thus enhanced, RAIDs increase I/O subsystem throughput dramatically for both large accesses and concurrent small accesses. And, they do so in a manner that can scale with increasing processor performance.

Unfortunately, RAIDs cannot reduce seek and rotational latencies. Thus, a disk array cannot service a series of modest sized requests to a moderately utilized I/O subsystem much, if any, faster than a single disk. Such requests do not greatly benefit from a disk array's high data transfer rate nor do they have the concurrency needed to take advantage of concurrent servicing of requests on an array. Yet, these are the characteristics of many important workloads. Even for those workloads that do benefit from the increased throughput and data bandwidth of a disk array, the mechanical latencies of a disk put a lower bound on the service time of any request. Thus, disk arrays, by themselves, do not address the latency aspect of the I/O problem.

In the broader context of I/O that includes network transmission and tertiary storage, the situation is similar. New technologies can generally increase throughput, but often fail to reduce latency especially for small requests whose service time is dominated by factors other than simple data transmission.

Fortunately, there are techniques that address at least the write side of the latency problem. Write performance is in some sense an easier problem than read performance because in most cases, an application does not have to wait for a write to finish. Buffering writes while allowing the application to continue successfully masks write latency. Thus, write buffering effectively converts the write latency problem into the write throughput problem of emptying write buffers.

	1985 BSD Study						1991 Study
Cache Size	390KB	1MB	2MB	4MB	8MB	16MB	7MB (avg)
Miss Ratio	49.2%	36.6%	31.2%	28.0%	26.2%	25.0%	41.4%

Table 2. Comparison of caching performance in 1985 and 1991. The numbers in this table are drawn from [Ousterhout85] and [Baker91]. The 1985 tracing study of the UNIX 4.2 BSD file system predicted cache performance for a range of cache sizes assuming a 30 second flush back policy for writes. The 1991 study measured cache performance on a number of workstations running Sprite. The cache size varied dynamically, but averaged 7MBytes. The diminishing returns from increasing cache size are evident in the 1985 results. Also striking is the difference between the predicted and measured performance of a large cache. The large cache was not nearly as effective as expected. The authors of the study conclude that growing file sizes were to blame for the disappointing cache performance. This result is strong evidence that we cannot rely on increased cache sizes to give us arbitrarily low cache miss ratios.

The counterpart on the read side to write buffering is file caching. But, file caches only mask read latency when the desired data is immediately available from the cache. Thus, their effectiveness in reducing read latency depends on having low miss ratios. For caches to compensate for the growing gap between processor and I/O performance, their miss ratios will have to drop proportionately. Is such improvement likely?

Table 2 compares the performance predicted for a variety of operating system file cache sizes in 1985 [Ousterhout85] with that observed in 1991 [Baker91] by a group at Berkeley. The first observation, based on the 1985 data, is that increasing the size of an already large cache does not reduce the miss ratio much. Some data sets just don't cache well, either because they are too large or because they are accessed too infrequently. In fact, the situation is even worse as the 1991 data shows. Since data sets are growing, it is necessary to increase the size of the cache just to maintain miss ratios. Thus, it will not be possible to compensate for the growing performance gap simply by increasing the cache size.

An alternate approach to reducing cache miss ratios is to anticipate cache misses and initiate I/O accesses before the data are needed. This technique, known as prefetching, typically masks read latency with computation. Prefetching can take several forms including asynchronous I/O and readahead.

Asynchronous I/O requires the user to explicitly request data before it is needed. Thus, using asynchronous I/O can require substantial programmer effort both for the initial implementation and for subsequent tuning to each new system configuration. From a systems perspective, a drawback of asynchronous I/O is that it does not facilitate global optimization of resources. There is no provision for balancing competing demands for buffers and I/O bandwidth from concurrently running applications.

More commonly, file systems provide prefetching based on the assumption of sequential file access. Organizing and fetching data in large file blocks [McKusick84] effectively prefetches unrequested data in the latter portions of the block. More explicitly, the file system can "readahead" sequential blocks of a file [Smith85, Feiertag71]. Recently, other researchers have explored prefetching for more complex access patterns by inferring future requests from a user's I/O request stream [Kotz91, Tait91, Palmer91, Korner90]. For well behaved applications, these techniques can effectively prefetch data and reduce cache misses. However, for the many applications whose access patterns appear random or that touch data only once, such techniques are ineffective.

The greater issue, though, is how well these techniques scale with improving processor performance. As I/O latencies grow in terms of processor cycles, prefetches must begin ever farther in advance if they are to complete in time. Unfortunately, current prefetching techniques are unable to confidently infer accesses far into the future.

Caching and prefetching are important and valuable techniques for reducing read latency. But, they are insufficient today, and they will not scale with processor performance. Thus, read latency remains an important problem, and it is the focus of this work.

3 Transparent Informed Prefetching (TIP)

To be most successful, prefetching should be based on *knowledge* of future I/O accesses, not inferences. We claim that such knowledge is often available at high levels of the system. For example, the file access patterns of many programs are determined by simple, single-flow control structures well understood by their programmers. These programmers could give hints about their programs' accesses to the file system. Thus informed, the file system could transparently prefetch needed data and optimize resource utilization. We call this *Transparent Informed Prefetching (TIP)*.

3.1 Application Examples

Here are a few examples of applications that could give hints to a TIP system.

An example from the Unix domain is the shell expansion of '*' to a list of files. For example, the command 'grep foobar *' says to search (grep) for all occurrences of the string 'foobar' in the files in the current directory. Either *grep* or the shell could give a hint about all of the files that *grep* will access.

The *make* program orchestrates the compilation of program modules and their linking with standard libraries. *Make* determines its actions according to a 'makefile' of instructions. After parsing a 'makefile' and checking the status of all modules to be built, *make* constructs a sequence of commands that it passes to a shell for execution. *Make* could give hints about the whole process to a TIP system.

Hints for more complex, non-sequential access patterns are also possible. An example from the supercomputer domain is stride access to large matrices [Miller91]. Data base applications often can predict their accesses to satisfy a query [Chou85, Stonebraker81, Selinger79].

Interactive applications, too, could provide hints. For example, when a bank customer inserts their card in an automatic teller machine, a hint about the customer's identity could be given, and the customer's account records retrieved while the customer enters their Personal Identification Number.

An important part of our research will be to expose hints in important, I/O-dependent applications. Initially, programmers will explicitly give hints. Eventually, we expect that compilers will be able to generate hints automatically for a wide range of applications.

3.2 The Benefits of TIP

User hints provide a TIP system with two new levers to apply to the I/O bottleneck: exposure of I/O concurrency and knowledge of future resource demands. TIP can exploit the I/O concurrency to convert the high throughput of new I/O technologies to the lower access latencies these new technologies cannot provide. TIP systems can exploit knowledge of future resource demands to better manage resources such as file cache buffers.

In the simplest exploitation of I/O concurrency, a system can start slow I/O accesses early and overlap their latency with ongoing computation to achieve a concurrency of two. Where data is accessed over a network, it is possible to overlap disk accesses, network transmission and computation to achieve a concurrency of three. Such concurrencies are both the goal and the limit of current prefetching techniques. As processors get faster, the computation time available to overlap with I/O shrinks, making it harder to achieve even these modest concurrencies. TIP can ease the problem by providing more reliable information about what to prefetch farther in advance, giving the system more time to prefetch. Interactive applications may even be able to give hints based on user actions that allow TIP to overlap I/O with a large amount (by computer standards) of user think time.

The greatest benefit of TIP comes when there is not much time to prefetch. Where applications dribble requests one at a time into the I/O subsystem, hints can inform the system of access patterns for whole files or even multiple files. Thus, hints can expose the concurrency of many I/O requests. On systems with disk arrays, TIP can service these requests with a concurrency bounded only by the number of disks in the array, potentially tens or hundreds instead of two or three. All of the files for the *grep* example could be retrieved concurrently. The application may not be spared the latency of the first access, but the latency of all subsequent requests will be masked behind that first access. The challenge becomes one of delivering data to the file cache as fast as the application consumes it. Thus, TIP uses the I/O concurrency that hints expose to convert the read latency problem into a read throughput problem in a manner analogous to the way write buffers work. TIP systems can use RAID throughput to provide low read latency. Conversely, for workloads characterized by small, serial I/O accesses that are otherwise unable to take advantage of disk arrays, TIP can use the lever of I/O concurrency to fully exploit the high throughput of RAIDs.

As a side effect of concurrent prefetching, TIP fills normally short I/O queues with prefetch requests. This creates new opportunities for storage subsystem optimizations. Deep queues of prefetch requests will not encumber demand requests with queuing delays if demand requests have higher priority. For disks, deeper queues allow better arm and rotation scheduling [Seltzer90]. For network I/O, multiple prefetch requests can be batched together, reducing network and protocol processing overhead. In both cases, deeper queues increase throughput that TIP uses to reduce read latency and application execution time.

Finally, TIP uses its knowledge of future I/O requests to improve cache management and reduce cache miss ratios. It is often possible to outperform an LRU page replacement algorithm, even without prefetching [Chou85, Korner90]. Unneeded blocks can be released early and needed blocks can be held longer. For example, knowing that a large file will be read sequentially twice, a TIP cache manager could hold onto the first few blocks of the file for use at the start of the second access while immediately flushing others and reusing the buffers for further prefetching. This strategy would minimize the amount of other data that is flushed from the cache.

3.3 Hints for TIP

As the previous section shows, TIP is much more than simple prefetching; it is a strategy for optimizing I/O. Powerful optimizations such as these require TIP to be applied where detailed knowledge of the system's static and dynamic state is available. The special advantage of TIP, however, comes from the additional power provided by high-level knowledge of future I/O activity. Therefore, application hints to TIP should disclose knowledge of high-level behavior rather than give advice about low-level resource policy decisions.

Hints that disclose	Hints that advise
I will read file F sequentially with stride S	cache file F
I will read these 50 files serially & sequentially	reserve B buffers & do not read-ahead

Table 3. Disclosure vs. Advice. This figure contrasts hints that disclose knowledge of future I/O activity with hints that give advice about low-level resource management policies. Hints that disclose use the same abstractions and semantics that the application later uses for I/O requests, whereas hints that advise use terms meaningful only to the system's implementation. It is important to use hints that disclose for portability, the flexibility needed to support global resource optimizations, and for adherence to sound software engineering principles of modularity.

Table 3 presents examples of hints that disclose knowledge of application behavior in contrast to hints that advise lower levels how to make policy decisions. Hints that advise do not give much usable knowledge to the file system. For example, if the file system is asked to cache a file, what should the system do if it cannot cache the whole file? Should it cache a part of the file? Which part? If, instead, the application discloses how it will access the file, the file system can determine how best to accommodate that access pattern given its current resource constraints. Without such disclosure, the file system may be needlessly forced to guess what to do.

Even if advice were always useful, a modular, portable application (that is, by many standards, well written) is not in a good position to give advice to a file system. It is unlikely to know anything of other demands being made on the file system, of its data's layout on disk, of its data's location on the network, or of the network's physical characteristics and current level of congestion. Such an application does not have the system state knowledge necessary to make good optimization decisions, so advice about such decisions is of limited value.

A simple rule of thumb distinguishes between good and bad hints. Good hints are specified using the same abstractions and semantics that an application later uses to demand access to its files, whereas bad hints are stated in terms meaningful only to the system's implementation. Thus, hints to the Unix file system should be in terms of file names and byte ranges not inodes, file blocks, network packets, or cache buffers. They should refer to reads and writes, not prefetching or caching. It is not a coincidence that good hints are compatible with modular software design. They are a means for transferring optimization information across module boundaries without violating those boundaries.

Giving hints is optional, but doing so encourages better system performance. Acting on hints is also optional; the system may ignore hints it cannot use or trust. Hints may be vague or imprecise, e.g. 'file *foo* will be touched.' But, inaccurate hints that incorrectly predict accesses should be avoided. Applications should give hints, even if imprecise, as soon as knowledge becomes available. Such imprecise hints may be supplemented later with more precise information, e.g. 'file *foo* will be read with stride *n*.' One TIP system may choose to ignore imprecise hints, but another, running on a more powerful machine, may be able to exploit them.

4 Demonstration of Principle

As we have argued, Transparent Informed Prefetching has the potential to reduce read latency, the most difficult I/O bottleneck. Our research into a TIP approach began with simple, controlled experiments demonstrating the potential benefits and obstacles of informed prefetching. Our goals with these experiments were to validate TIP as a tool for reducing read latency, to determine if more than a simple, user-level mechanism is needed, to uncover implementation problems, and to develop experience incorporating hints into applications.

Our experiments evaluated TIP for accesses both to a local disk and over a network to the Coda distributed file system [Satyanarayanan90]. In both cases, there was only a single request server, either a local disk or a remote file server. This fact limited the potential benefits we could expect to achieve because there was little or no opportunity for concurrency in the I/O subsystem. In addition, since the prefetching was carried out at the user level, there was no opportunity for cache management gains and little opportunity for I/O subsystem optimizations. Our results are thus conservative with respect to what could be realized by a full-fledged TIP system. In the experiments we describe below, virtually all benefits came from overlapping computation and I/O. The maximum benefit from such prefetching is a 50% reduction in execution time, and this is only possible when, without prefetching, an application spends exactly 50% of its time computing without overlapping any I/O, and with prefetching, it completely overlaps I/O and computation. Thus, while we expect to see significant reductions in execution time, the rarity of ideally balanced applications limits these reductions to well under 50%.

4.1 Test Description

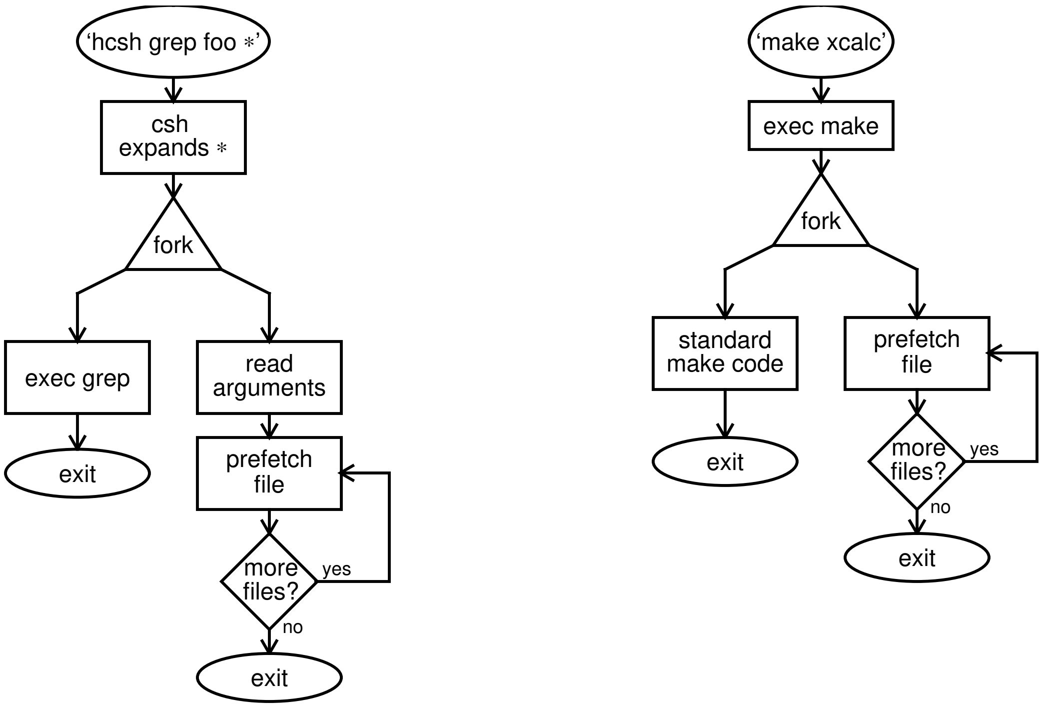
We used two hardware platforms for our tests. The local disk tests were conducted on a Sun Sparcstation 2 running Mach/BSD Unix 4.3. The remote tests were run on two Decstation 5000/200 also running Mach, one of them the client, and the other the server for the Coda File System. Coda is a descendent of the Andrew File System [Satyanarayanan85], but with enhancements for availability and mobility. In this system, whole files are transferred to requesting client machines which then cache them on local disks. Reads and writes are applied through the client's buffer cache to its local disks' copy. Writes are transferred back to the server when the file is closed.

Our experiments concentrate on two applications, a *make* program doing a program build, and the Unix shell expanding '*' to a list of files. Figure 1 shows the control flow in these experiments. In both experiments, a separate user-level process prefetched files while the program under test ran unmodified in its own process.

Figure 1a, on the left, illustrates the shell experiments. A wrapper program, called hcsh for hint-csh, handles the prefetching. The experimenter invokes hsch sending it a command to execute. The shell does any file name expansion before hsch executes. Hcsh spawns a prefetching process, passing it the arguments to the command. This process assumes that all arguments without a leading '-' are names of files that the command will read and starts to prefetch them serially and sequentially into the buffer cache. Meanwhile, the parent hcsh process invokes the command whose reads hopefully hit the prefetched data in the buffer cache. The command and the prefetch process then run concurrently.

Figure 1b shows the flow of events for *make* experiments. The *make* program was modified slightly so that the first thing it does is fork off a prefetching process. The parent process then simply continues on, executing the standard *make* code. The files to prefetch were determined in advance using a file system tracing facility [Mummert92] and hard-wired into the prefetcher. Thus, the prefetching was based on accurate knowledge of the accesses to be performed.

The prefetch mechanism differs in the local and remote cases. In the local case, the prefetch process sequentially reads files in 64K chunks using standard system calls. This had the effect of loading them into the buffer cache where they were available to the application. For files on the Coda server, the prefetch



(a) Prefetching for shell expansion **(b) Prefetching for *make***

Figure 1. Flow chart for the two test programs. Diagram (a) shows the configuration for exploiting the ‘*’ expansion of file names for shell commands. The command runs down the left side of the fork while the prefetch process runs down the right. The prefetch process determines what to prefetch from the command line arguments. Diagram (b) shows the configuration for the *make* example. It is similar to the previous example except that the files to prefetch are determined in advance and wired into the prefetcher.

process took advantage of an asynchronous prefetch ioctl built into Coda to fetch the files over the network to the client workstation. Since the transfer went through the client’s buffer cache on its way to the local disk, the file data was available to the application in the buffer cache.

4.2 Test Results

Table 4 compares the elapsed times to run applications with and without prefetching on both the local disk and the Coda File System. The first application, *make xcalc*, compiles and builds the X window calculator tool. The second, *grep foobar* *, searches 58 files containing a total of 1 MB all stored in (the cache of) a remote Coda file server.

In this table, numbers in parentheses are the standard deviations for the measurements. Since the local disk and Coda tests were performed on different hardware platforms, the numbers are not directly comparable. In the ‘hot cache’ runs, all data read throughout the job was preloaded into the local buffer cache, so the job never blocked for the disk. These numbers represent a lower bound on the elapsed time. At the start of the ‘cold cache’ runs, there was no data in the buffer cache or client disk cache, though, in the distributed case, the server’s buffer cache was not flushed between runs. Also, executables were not flushed from virtual memory. The ‘cold cache w/prefetching’ runs were started just like the ‘cold cache’ runs, but they used prefetching to speed access to the files. The ‘% reduction’ represents the benefits of

Application	Local Disk				Distributed File System (Coda)			
	hot cache	cold cache	cold cache w/prefetch	% reduction	hot cache	cold cache	cold cache w/prefetch	% reduction
make xcalc	9.17 (0.03)	14.19 (0.13)	12.40 (0.07)	12.6	18.29 (2.00)	40.41 (3.63)	32.20 (2.74)	20.3
grep foobar *	1.22 (<0.01)	3.29 (0.13)	3.30 (0.04)	0	1.85 (0.01)	7.86 (0.77)	5.55 (0.68)	29.4

Table 4. Results of Tests. This table summarizes the results of the experiments. Times for all runs are the average total elapsed execution time in seconds. Numbers in parentheses are the standard deviation of the measurements. For the 'hot cache' runs, all data read during the run is preloaded into the buffer cache. These numbers represent an absolute lower bound on execution time. For the 'cold cache' runs, all data was flushed from the buffer cache and, for the Coda runs, from the local disk cache. Executables were not flushed from virtual memory, and data was not flushed from the Coda server's buffer cache. The 'cold cache w/prefetch' runs had the same initial conditions as the 'cold cache' runs, but used prefetching to speed data retrieval. The '% reduction' is the reduction in execution time due to prefetching in the 'cold cache w/prefetch' runs compared to the 'cold cache' runs. No gain was observed for the *grep* example running on the local disk because, with little computation time, the disk was running flat out with or without prefetching.

prefetching.

4.3 Lessons from Tests

Although our experiments were preliminary, they nevertheless served their purpose of demonstrating the benefits of informed prefetching and educating us about implementation pitfalls. Among the performance pits we fell into were high overhead, terrible disk scheduling, and runaway prefetching that overran the cache and starved the user. To avoid these pitfalls, TIP must be able to track consumption to throttle prefetching and reuse buffers, and to prefetch efficiently without delaying the user. In this section, we expand on these requirements and what happens when TIP doesn't meet them.

Our experiments' prefetching technique, especially from the local disk, was inefficient. It paid a high price in CPU overhead, and sometimes in disastrously poor disk scheduling. We incurred unnecessary overhead in a number of ways. For both the local and remote tests, we had to support the overhead of an independent prefetch process and its concomitant context switching, scheduling, and system calling costs. The obvious solution to this problem is to put the prefetching mechanism into the file system.

In local disk tests the prefetch process could not move data directly into the cache. Instead, it did so indirectly by making standard read system calls and paying the cost of copying the data from the cache up to the prefetch process. A more serious drawback was that, because the read calls were blocking, there could only be one outstanding prefetch request at a time. Thus, we did not have the deep queues needed for efficient disk scheduling. In some cases, the queue depth was zero between read calls, leaving the disk idle.

For some applications, such as *compress*, which are constantly writing out data, writes would be interleaved with prefetch requests in the disk queue. Because the BSD Unix file system located the output files far from the input files, these interleaved prefetches and writes resulted in a series of very long seeks. It is hard to imagine worse disk scheduling. From this experience, it is clear that latency tolerant accesses such as prefetches and write-behinds should be either deeply queued so that scheduling can be employed or queued in batches to encourage locality.

Tests employing the Coda distributed file system had the benefit of an asynchronous ioctl which allowed the prefetch process to flood the I/O subsystem with prefetch requests. This led to its own unique problem, thread starvation. Because Coda used a fixed number of worker threads to handle user requests, the prefetch process monopolized these threads and starved the user's demand reads. Clearly, prefetches must operate at lower priority than demand fetches. We fixed the problem in Coda by limiting the number of threads that could be devoted to prefetching before making the measurements in Table 4.

We anticipate a similar problem at the local disk once we fill disk queues with prefetch requests. A large, efficient, prefetch request could stall a demand request behind it. New functionality in disk controllers that could suspend a low priority request to service a demand request immediately could help resolve this problem. A less attractive alternative is to keep prefetch requests small so that they never take long to complete.

The voracious Coda prefetcher created another problem. Sometimes it got so far ahead of the user process that it flushed unused prefetched data out of the cache. This can have dramatic consequences. One job initially experienced a 17% slowdown with prefetching because it had to go to disk to retrieve data that the prefetcher had flushed out of the cache. The local disk prefetcher occasionally had the same problem. For these preliminary tests, we avoided this problem by using a buffer cache big enough to hold all of the data. The long-term solution is to track user reads of prefetched data and throttle prefetching accordingly.

Actually, this cache overrun problem had an additional, ironic twist. The cache held on to already used and no longer needed data even as it was ejecting prefetched data that had yet to be used. The used data had been more recently accessed and so was higher in the LRU list. This phenomena highlights the importance of combining prefetching with buffer management.

5 Work in Progress and Future Work

The experimental results support our contention that TIP offers a powerful mechanism for overcoming the looming I/O crisis. They also demonstrate the need to integrate TIP with low-level resource management. To this end we are implementing TIP in our Mach/Unix research environment and instrumenting important applications, including the ones already tested, to provide hints. Since we feel that the greatest gains of TIP occur with high levels of concurrency in the I/O subsystem, we are also building a disk array on which our implementation of TIP can flex its muscles. In the rest of this section, we describe a few possibilities for future work on even more powerful TIP mechanisms.

Although we feel that often programers can easily give good hints, automatic hint generation would be even easier and would provide a much broader base of applications providing hints. Given a hint interface, why not extend optimizing compiler techniques to use it? While file names and access patterns could be concealed in complex data structures, we believe that many programs access files simply; file names come as arguments, access calls are made from outer loops, and offsets are computed simply from previous offsets or loop indices. With such straightforward programming styles, we expect that precise hints can be extracted automatically.

In some cases, even programmers may not fully understand the file accesses their programs make. An access pattern profiler could be built on top of an efficient file system tracing facility [Mummert92]. The profiler could help programmers improve the quality of their hints, or it could be used directly to generate hints for future runs.

A variation on giving hints for other programs is giving hints for library routines such as a fast-fourier transform (FFT) procedure in a math library. Since the programmer may not be familiar with the implementation of FFT, they may be unable to give good hints. The math library could export the routine *fft.hint* to facilitate hints. A user program could call *fft.hint* and let it give the appropriate hint to the file system. Effectively, the application gives a hint to the library which repackages it and sends it to the file system. This is an example of how hints can be used to pass information through multiple layers of software.

6 Related Work

The idea of giving hints is not new. For example, Trivedi suggested using programmer or compiler generated hints for prepaging [Trivedi79]. Hints are now widely enough understood that they appear in various existing implementations. For example, Sun Microsystems' operating system provides two "advise" system calls that instruct the virtual memory system's policy decisions [SunOS-vadvise].

Database systems researchers have long recognized the opportunity to accurately prefetch based on application level knowledge [Stonebraker81]. They have also extensively examined the opportunity to apply this knowledge through advice to buffer management algorithms [Sacco82, Chou85, Cornell89, Ng91] and for I/O optimizations [Selinger79]. We hope to extend these techniques to prefetching. Also, our work emphasizes a more solid partitioning of function between application and operating system.

Many researchers have looked into prefetching based on access patterns inferred from the stream of user I/O requests [Kotz91, Tait91, Palmer91, Korner90]. Our view of the problem is perhaps most similar to Korner's who recognized the value of high-level hints as a means of bridging levels of abstraction from files to disk blocks. Her characterizations of access patterns, like ours, are at a high level of abstraction. But, we carry this approach further and ask for hints from applications themselves to bridge the gap between applications and the operating system.

Recently, researchers have proposed an object-oriented file system layered on top of the Unix file system called ELFS [Grimshaw91]. ELFS has knowledge of file structure and high-level file operations that allow it to help prefetch and caching operations. However, ELFS emphasizes user control over file activity. It would be possible instead for users to give hints to ELFS which would translate them into hints for the low-level file system. Thus, hints could be used to bridge layers of the system at the application level. In such a context, ELFS and TIP would complement each other well.

Our work differs from all previous prefetching work in one important respect. We do not view the overlapping of I/O with computation as the major benefit of prefetching. The success of such overlapping is extremely sensitive to the ratio of time spent on I/O relative to computation and that ratio changes constantly as processor performance increases. Instead, we believe that the greatest benefit of prefetching will come from the exposure of I/O concurrency that can take advantage of new high-throughput technologies. Thus, we believe that the high I/O concurrency provided by early, accurate user-supplied hints is critical to the ultimate success of prefetching.

7 Conclusion

Transparent Informed Prefetching, TIP, extends the power of caching and prefetching to reduce file read latency by exploiting application-level knowledge of future access patterns. These access patterns are expressed to TIP in the form of hints that disclose rather than advise and serve to expose concurrency in the I/O workload. TIP systems can then cooperate with resource management policies to increase the utilization and efficiency of high-throughput network and storage systems. This effectively converts the high throughput of new peripheral technologies into low read latency for application programs.

In this paper, we have introduced informed prefetching as the key to reducing read latencies for application programs whose files do not cache well because of their large size, non-sequential access patterns, or inherently read-once nature. The key to informed prefetching is knowledge of future application file references conveyed by hints expressed in terms of operations on files, not resource management policy options. However, TIP should not operate independent of resource management because it can squander resources if not properly checked.

To demonstrate our ideas and develop experience instrumenting programs with hints and incorporating prefetching into caching access patterns, we have conducted a few preliminary experiments. The results of these experiments are quite promising. Applications obtaining data from a remote file server may be able to reduce their execution time by up to 30% and applications obtaining data from a single local disk may see 13% reductions in their execution time.

8 Acknowledgment

We would like to thank Lily Mummert for her important contributions to this paper. In addition to many helpful discussions, she constructed the original file tracing facility, later adapted it to our special needs, and modified the Coda File System to support low-priority prefetch requests.

We also thank Mark Holland for his constructive comments about an early draft of this paper.

9 References

[Amdahl67] Amdahl, G.M., "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. AFIPS 1967 Spring Joint Computer Conference*, V 30, Atlantic City, New Jersey, April 1967, pp. 483-485.

[Baker91] Baker, M.G., Hartman, J.H., Kupfer, M.D., Shirriff, K.W., and Ousterhout, J.K., "Measurements of a Distributed File System," *Proc. of the 13th Symp. on Operating System Principles*, Pacific Grove, CA, October 1991, pp. 198-212.

[Cate92] Cate, V., "Alex --- A Global Filesystem," *Proceedings of the Usenix File Systems Workshop,* Ann Arbor, MI, May 1992, pp. 1-11.

[Chou85] Chou, H. T., DeWitt, D. J., "An Evaluation of Buffer Management Strategies for Relational Database Systems," *Proc. of the 11th Int. Conf. on Very Large Data Bases*, Stockholm, 1985, pp. 127-141.

[Cornell89] Cornell, D. W., Yu, P. S., "Integration of Buffer Management and Query Optimization in Relational Database Environment," *Proc. of the 15th Int. Conf. on Very Large Data Bases*, Amsterdam, Aug. 1989, pp. 247-255.

[Feiertag71] Feiertag, R. J., Organisk, E. I., "The Multics Input/Output System," *Proc. of the 3rd Symp. on Operating System Principles*, 1971, pp 35-41.

[Gibson91] Gibson, G. A., *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*, Ph.D. dissertation, University of California, Berkeley, technical report UCB/CSD 91/613, April 1991. Available in MIT Press' 1991 ACM distinguished dissertation series, 1992.

[Grimshaw91] Grimshaw, A.S., Loyot Jr., E.C., "ELFS: Object-Oriented Extensible File Systems," Computer Science Report No. TR-91-14, University of Virginia, July 8, 1991.

[Kim86] Kim, M.Y., "Synchronized Disk Interleaving," *IEEE Trans. on Computers*, V. C-35 (11), November 1986.

[Kistler92] Kistler, J.J., Satyanarayanan, M., "Disconnected Operation in the Coda File System," *ACM Trans. on Computer Systems*, V10 (1), February 1992, pp. 3-25.

[Korner90] Korner, K., "Intelligent Caching for Remote File Service, *Proc. of the Tenth Int. Conf. on Distributed Computing Systems*, 1990, pp.220-226.

[Kotz91] Kotz, D., Ellis, C.S., "Practical Prefetching Techniques for Parallel File Systems," *Proc. First International Conf. on Parallel and Distributed Information Systems*, Miami Beach, Florida, Dec. 4-6, 1991, pp. 182-189.

[Livny87] Livny, M., Khoshafian, S., Boral, H., "Multidisk Management Algorithms," *Proc. of the 1987 ACM Conf.*

on Measurement and Modeling of Computer Systems (SIGMETRICS), May 1987.

[McKusick84] McKusick, M. K., Joy, W. J., Leffler, S. J., Fabry, R. S., "A Fast File System for UNIX," *ACM Trans. on Computer Systems*, V 2 (3), August 1984, pp. 181-197.

[Miller91] Miller, E., "Input/Output Behavior of Supercomputing Applications," University of California Technical Report UCB/CSD 91/616, January 1991, Master's Thesis.

[Mummert92] Mummert, L., Satyanarayanan, M., "Efficient and Portable File Reference Tracing in a Distributed Workstation Environment," Carnegie Mellon University, manuscript in preparation.

[Nelson88] Nelson, M. N., Welch, B. W., Ousterhout, J. K., "Caching in the Sprite Network File System," *ACM Trans. on Computer Systems*, V 6 (1), February 1988.

[Ng91] Ng, R., Faloutsos, C., Sellis, T., "Flexible Buffer Allocation Based on Marginal Gains," *Proc. of the 1991 ACM Conf. on Management of Data* (SIGMOD), pp. 387-396.

[Ousterhout85] Ousterhout, J.K., Da Costa, H., Harrison, D., Kunze, J.A., Kupfer, M., and Thompson, J.G., "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," *Proc. of the 10th Symp. on Operating System Principles*, Orcas Island, WA, December 1985, pp. 15-24.

[Ousterhout89] Ousterhout, J., Douglis, F., "Beating the I/O Bottleneck: A Case for Log-Structured File Systems," *ACM Operating Systems Review*, V 23 (1), January 1989, pp. 11-28.

[Palmer91] Palmer, M.L., Zdonik, S.B., "FIDO: A Cache that Learns to Fetch," Brown University Technical Report CS-90-15, 1991.

[Patterson88] Patterson, D., Gibson, G., Katz, R., A, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proc. of the 1988 ACM Conf. on Management of Data (SIGMOD)*, Chicago, IL, June 1988, pp. 109-116.

[Reddy89] Reddy, A.L.N., Banerjee, P., "Evaluation of Multiple-Disk I/O Systems," *IEEE Tans. on Computers*, December 1989.

[Rosenblum91] Rosenblum, M., Ousterhout, J.K., "The Design and Implementation of a Log-Structured File System," *Operating Systems Review (Proceedings of the 13th SOSP*), Volume 25 (5), October 1991, pp 1-15.

[Sacco82] Sacco, G.M., Schkolnick, M., "A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model," *Proc. of the Eighth Int. Conf. on Very Large Data Bases*, September, 1982, pp. 257-262.

[Salem86] Salem, K. Garcia-Molina, H., "Disk Striping," *Proc. of the 2nd IEEE Int. Conf. on Data Engineering*, 1986.

[Satyanarayanan85] Satyanarayanan, M., Howard, J. Nichols, D., Sidebotham, R., Spector, A., West, M., "The ITC Distributed File System: Principles and Design," *Proc. of the Tenth Symp. on Operating Systems Principles*, ACM, December 1985, pp. 35-50.

[Satyanarayanan90] Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H., Steere, D. C., "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Trans. on Computers*, V C-39 (4), April 1990.

[Selinger79] Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G., "Access Path Selection in a Relational Database Management System," *Proc. of the 1979 ACM Conf. on Management of Data (SIGMOD)*, Boston, MA, May, 1979, pp. 23-34.

[Seltzer90] Seltzer, M. I., Chen, P. M., Ousterhout, J. K., "Disk Scheduling Revisited," *Proc. of the Winter 1990 USENIX Technical Conf.*, Washington DC, January 1990.

[Smith85] Smith, A.J., "Disk Cache--Miss Ratio Analysis and Design Considerations," *ACM Trans. on Computer Systems*, V 3 (3), August 1985, pp. 161-203.

[Spector89] Spector, A.Z., Kazar, M.L., "Wide Area File Service and The AFS Experimental System," *Unix Review*, V 7 (3), March, 1989.

[Stonebraker81] Stonebraker, Michael, "Operating System Support for Database Management," *Communications of the ACM*, V 24 (7), July 1981, pp. 412-418.

[Tait91] Tait, C.D., Duchamp, D., "Detection and Exploitation of File Working Sets," *Proc. of the 11th Int. Conf. on Distributed Computing Systems*, Arlington, TX, May, 1991, pp. 2-9.

[Trivedi79] Trivedi, K.S., "An Analysis of Prepaging", *Computing*, V 22 (3), 1979, pp. 191-210.