Goddard Conference on Mass Storage Systems and Technologies, Greenbelt, MD, September, 1992, pp. 329-342.

Using Transparent Informed Prefetching (TIP) to Reduce File Read Latency

R.H. Patterson, G.A. Gibson, M. Satyanarayanan
Carnegie Mellon University

Outline

I/O performance is lagging

No current solution fully addresses read latency

TIP to reduce latency

- **exploits high-level hints that don't violate modularity**
- **converts throughput to latency**

Preliminary TIP test results

As processor performance gains continue to outstrip Input/Output gains, I/O performance is becoming critical to overall system performance. File read latency is the most significant bottleneck for high performance I/O. Other aspects of I/O performance benefit from recent advances in disk bandwidth and throughput resulting from disk arrays [Patterson88], and in write performance derived from buffered write-behind and the Log-structured File System [Rosenblum91]. The access gap problem limiting improvements in read latency is exacerbated by distributed file systems operating over networks with diverse bandwidth [Spector89, Satyanarayanan85]. In this paper, we focus on extending the power of caching and prefetching to reduce file read latencies by exploiting hints from high-levels of a system. We describe such Transparent Informed Prefetching, TIP, and its benefits. We argue that hints that disclose high level knowledge are a means for transferring optimization information across, without violating, module boundaries. We discuss how TIP can be used to convert the high throughput of new technologies such as disk arrays and log-structured file systems into low latency for applications. Our preliminary experiments show reductions in wall-clock execution time of 13% and 20% for a multiple module compilation tool (make) accessing data on a local disk and remote Coda file server, respectively, and a reduction of 30% for a text search (grep) remotely accessing many small files.

Solutions to I/O Bottleneck

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

But, cache effectiveness is declining

This table shows the mechanisms most heavily used to combat the growing I/O bottleneck. Written data benefits from write-behind buffering and log-structured files systems, while I/O throughput is directly increased by parallelism in disk arrays. Read latency, however, is only reduced by caching and prefetching. As will be shown next, caches will not, by themselves, be able to relieve the I/O bottleneck, and prefetching will emerge as a critical approach to the problem.

Effective I/O Performance with Caching

$$T_{I/O} = MC_M + (1-M)\,C_H \approx MC_M$$

$$T_E = T_C + N_A T_{I/O} \approx T_C + N_A M C_M$$

$T_{I/O}$ = I/O time
C_M = cost of a miss
T_E = execution time
N_A = number of I/Os
M = cache miss ratio
C_H = cost of a hit
T_C = computation time

Miss ratio for effective I/O performance to scale with CPU performance

CPU/IO Perf.	Current=1	10	100
Miss Ratio	40%	4%	0.4%

Caches reduce the average I/O service time by reducing number of I/O requests that must be serviced by slow peripheral devices. The ratio of requests thus serviced to the total number of requests is the miss ratio. For caches to compensate for the growing gap between CPU performance and I/O peripheral performance, they must reduce their miss ratios. This simple model quantifies this relationship.

The average I/O service time, $T_{I/O}$, is the weighted sum of the service times for requests that miss in the cache and must be serviced by the I/O subsystem, C_M, and for requests that hit in the cache, C_H. The cache miss ratio, M, weights the sum. Since $C_H << C_M$, the average I/O service time is roughly MC_M. The execution time for a program, T_E, is the sum of the time spent on computation, T_C, and the total time spent on I/O. Time spent on I/O is, in turn, the product of the number of I/O requests, N_A, and the average time to service a request. As processor improvements reduce T_C relative to C_M, the miss ratio, M, must be reduced to achieve corresponding reductions in the time spent on I/O. The table shows the improvement needed in the cache miss ratio for the effective I/O performance to keep pace with processor gains. A cache that currently has a 40% miss ratio must improve to 4% to match a ten-fold increase in processor performance and to 0.4% to match the 100 fold increase expected in the next ten to fifteen years. As the next slide shows, such miss ratios are most unlikely.

Cache Miss Ratios

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

- **Diminishing returns from larger caches**
- **Disappointing performance over time**
  - **> growing file sizes**

Clearly, caching alone cannot provide the needed performance improvements

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 workstations running Sprite. The Sprite 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 concluded 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 the extremely low miss ratios needed to improve effective I/O performance. This leaves us with prefetching as a tool for improving I/O read latency.

Transparent Informed Prefetching (TIP)

1) Encapsulate programmer knowledge about future I/O requests in a hint

2) Transfer hint to file system

3) File system uses hints to transparently prefetch data and manage resources

Prefetching can pre-load the cache to reduce the cache miss ratio, or, at least reduce the cost of a cache miss by starting the I/O early and thereby improve effective I/O performance. While there have been a number of approaches to prefetching [Kotz91, Smith85, McKusick84, Feiertag7], it is often difficult to know what to prefetch, and prefetching incorrectly can end up hurting performance [Smith85].

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. 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).

Obtaining Hints

Early knowledge of serial file access

Access patterns part of code algorithm

- **large matrix supercomputing: read by row, read by column**

Hints generated by: programmer, compiler, profiler

Critical to the success of informed prefetching is the availability of accurate and timely hints. An important part of our research will be to expose such hints in important, I/O-dependent applications. However, we don't think this will be as hard as it might seem. After all, the success of sequential readahead is largely the product of "discovering" that an application is sequentially accessing its files; this is really known a priori because a programmer has chosen to do so. Often, it is known well in advance that many files will be thus accessed. It is a simple step to have programmers notify the I/O system, through a hint, of sequential access patterns.

In addition to the simplest hints about sequential accesses, programmers could give hints about more complex, non-sequential access patterns. An important beneficiary of this approach will be the large scientific programs that execute alternating row and column access patterns on huge matrix data files [Miller91]. At least one of these access patterns will not be sequential in the file's linear storage, yet the pattern is easily and obviously specified by a programmer.

In addition to programmer-generated hints, compilers could automatically generate hints, or a profiler could be used to generate hints for future runs of a program.

Application Examples

grep foo *

- Shell expands '*' to a list of filenames.
- Grep searches for a string, 'foo,' in all of the files in the list.
- From invocation, it is known that all of the files on the list will be read sequentially.
- Give a hint about all of the files at once.

make

- makefile specifies all files to be touched from the start
- make generates hints for binaries it will invoke and the files they will touch.

While we believe that scientific applications will be major beneficiaries of TIP, common Unix applications can also benefit. Her are two examples.

Given the command 'grep foo *,' the shell expands the '*' into a list of all files in the current directory and invokes the 'grep' program which searches for the string 'foo' in all the files. Grep, or even the shell if it knows a little about grep from a command registry, can issue a hint notifying a TIP system that all the files in the list will soon be read. If the system has stored these files on an underutilized disk array, many or all will be fetched concurrently.

We expect programs issuing hints on behalf of other programs, such as the shell on behalf of grep, to be a common occurrence. Another example is the 'make' program which 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 set of command sequences that it will pass to a shell for execution. These commands or the shell itself can issue hints about their I/O accesses. Pursuing a TIP approach more aggressively, 'make' can use the same command registry as the shell to issue hints even before it issues the commands.

TIP Converts High Throughput to Low Latency

Use excess storage bandwidth to pre-load caches with future accesses and overlap I/O with computation

Expose concurrency to pack low-priority queue with prefetch requests

- **Optimize seek scheduling**
- **High-throughput disk arrays simultaneously service multiple requests**
- **Multiple network requests may be batched together**

Cache management superior to LRU

Armed with knowledge of future file accesses, a system employing TIP can improve performance in three important ways.

1) At the most basic level, TIP, as for all prefetching, can overlap slow I/O accesses with other useful work so that applications spend less time idly waiting for these accesses to complete. But, because TIP systems know what to prefetch, they can prefetch more aggressively to pre-load the cache with future accesses and further reduce cache misses.

2) Using TIP, normally short I/O queues can be filled with low-priority prefetch requests giving more opportunities for low-level I/O optimizations. For an individual disk, deeper queues allow better arm and rotation scheduling [Seltzer90]. For a disk array, deeper queues mean more requests are available for concurrent servicing by independent disks. On a network, prefetch requests can be batched together, reducing network and protocol processing overhead.

3) TIP improves cache management to further reduce cache miss ratios. If it is known what data will be needed in the future, it may be possible to outperform an LRU page replacement algorithm, even without prefetching. Unneeded blocks can be released early, and needed blocks can be held longer.

The first two benefits make TIP an excellent mechanism for exploiting the high throughput of emerging storage technology to provide the low latency that these technologies cannot provide. Combined with improved cache management, these three benefits make TIP a powerful tool for overcoming the widening access gap.

Hints are Disclosure not Advice

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

- **Users not qualified to give advice**
- **Advice not portable, disclosure is**
- **Disclosure allows more flexibility**
- **Disclosure supports global optimizations**
- **Disclosure hints consistent with sound SWE principles**

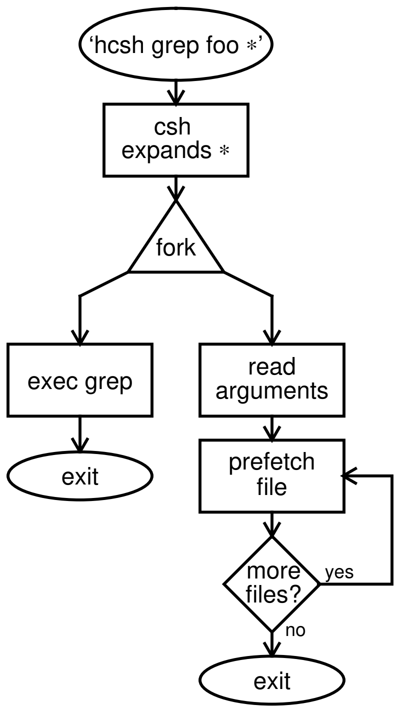
As the previous slide showed, TIP is much more than simple prefetching; it is a strategy for optimizing I/O. For a number of reasons, such powerful optimizations depend on having hints that disclose knowledge of future I/O operations instead of hints that give advice about I/O subsystem operation.

Advice about low-level operations depends on detailed system-specific knowledge. Even if a user had such knowledge of a system's static configuration, they could not know about the system's dynamic state. Thus, the user is not qualified to give advice on how to optimize the dynamic operation of the system. Furthermore, such system-specific knowledge would not apply to other systems, and so, advice that exploits it would not be portable to other systems.

Additionally, hints that advise, such as, 'cache this file,' do not give much usable knowledge to the TIP system. What should the TIP system do if it cannot cache the whole thing? Should it cache a part of the file? Which part? If, instead, the application discloses how it will access the file, the TIP system has the flexibility to respond appropriately. This flexibility is crucial for balancing competing demands for global resources.

Good hints that disclose are specified using the same semantics that an application later uses to demand access to its files, whereas bad hints which advise concern themselves with a system's implementation. 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.

Preliminary Test



Prefetching for shell expansion. **Prefetching for 'make.'**

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, determine if more than a simple, user-level mechanism is needed, uncover implementation problems, and develop experience incorporating hints into applications.

We used two hardware platforms for our tests. The local disk tests were conducted on a Sun Sparcstation 2 running Mach 2.5/BSD Unix 4.3. The remote tests were run on two Decstation 5000/200 running Mach 2.5, one of them the client, and the other the server for the Coda distributed file system [Satyanarayanan90].

We tested the two applications previously mentioned, shell expansion of '*' for 'grep,' and 'make' building a program called 'xcalc.' Flow charts for the two test programs are above. The chart on the left shows the configuration for exploiting shell expansion of '*.' A fork operation splits the program into two processes. The command runs down the left side of the fork, while an independent prefetch process runs down the right side of the fork. The prefetch process uses the expanded list of filenames to determine what to prefetch. The right-hand chart shows the configuration for the 'make' example. It is similar to the previous example except that a tracing facility [Mummert92] is used to determine in advance the files to prefetch.

To prefetch from the local disk, the prefetch process simply read the appropriate files, indirectly causing the data to be moved into the cache. To prefetch remotely from Coda, the prefetch process used a special prefetch ioctl to explicitly and asynchronously transfer the file to the local machine.

Test Results

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

- **make xcalc: compile & link X window calculator**
- **grep foo *: 58 files, 1 MB**
- **Results limited by lack of parallelism in I/O subsystem**

This table compares the elapsed times to run two applications with and without prefetching on both the local disk and the Coda distributed 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 MByte all stored in (the cache of) a remote Coda file server.

The numbers in parentheses are the standard deviations for the measurements. Since the local tests were performed on a Sun Sparcstation 2 whereas the Coda tests were performed on Decstation 5000/200, the numbers are not directly comparable. In the 'hot cache' runs, all data read throughout the job were in 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 cleared between runs. 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 prefetching.

TIP systems will only be able to approach the lower bound represented by the 'hot cache' numbers when combined with high-throughput I/O subsystems unavailable for these tests. In the grep test on the local disk, the execution time is dominated by I/O. The disk is already running flat out, so there is no time for prefetching. Grep with a disk array would still keep one disk busy and would run in about the same amount of time, but grep with TIP and a disk array would keep many disks busy. The total time spent on I/O would drop and performance approaching the 'hot cache' lower bound should be possible.

Lessons from Tests

- Independent prefetch process overhead too high
- Single prefetch process ⇒ no deep prefetch queues
- Coda ioctl allowed too much prefetching
  > thread starvation - need low-priority prefetching
  > premature cache flushing - need to track consumption
- Poor cache buffer replacement performance
- Disk write scheduling often very inefficient

Although our experiments were preliminary, they served their purpose of demonstrating the benefits of informed prefetching and educating us about implementation pitfalls.

Using independent prefetch processes incurred a lot of extra overhead, especially in the local disk tests. Context switching, process scheduling inefficiencies, system call cost, and, on the local disk, data copy costs all reduced the performance of the prefetch tests. But, the most serious hindrance to prefetching from the local disk was that, because the read system calls used are blocking, there was never more than one prefetch request in the queue at a time. Thus, we did not benefit from the scheduling advantages offered by deeper queues.

The coda tests avoided this problem with the asynchronous prefetch ioctl. They suffered instead from over-prefetching. Until we reduced the priority of the prefetches, they interfered with demand fetches, reducing performance. Also, prefetches sometimes got ahead of the actual job and caused prefetched data that had not yet been used to be replaced in the cache by newly prefetched data. Clearly, a real system will need to track data consumption to avoid this problem. This was an extreme example of the cache manager making uninformed decisions. The cache held onto data that had just been used in preference to prefetched data that was about to be used. Integrating TIP with the cache manager should greatly improve performance. In the tests, we avoided this problem by using a very large cache that could all of the data.

Writes of whole blocks were not buffered and thus were interleaved with both prefetch and demand reads which led to very poor disk scheduling. This highlighted the importance of buffered writes.

Summary

TIP uses hints to convert high throughput storage to low latency where caching fails

Hints that disclose, not advise, provide the best information and are consistent with sound SWE principles.

Applicable to local disk and network file servers

Immediate Plans

- **modify Coda/BSD/Mach to accept and exploit correct hints**
- **find & instrument applications**
  - **> make, search, visualization, simulation**

Transparent Informed Prefetching, TIP, extends the power of caching and prefetching to reduce both local and remote file read latency by exploiting application-level knowledge of future access patterns. TIP systems can cooperate with resource management policies to increase the utilization and efficiency of high-throughput network and storage systems. Many future accesses become current accesses that can exploit the parallelism of disk arrays or may batched to reduce network overheads. Disk accesses and buffer allocation may be improved with foreknowledge of future accesses. TIP effectively converts the high throughput of new peripheral technology into low read latency for application programs.

Informed prefetching depends on hints from applications that disclose their future I/O accesses in terms of operations on files. Hints should not give advice about I/O subsystem operation nor be expressed in terms of resource management policy options. This distinction is important for hint portability and consistency with software engineering principles of modularity, and for the TIP system to be able to effectively manage global resources.

Preliminary tests have confirmed the potential benefits of informed prefetching and highlighted some of the potential pitfalls of implementation.

Our next step is to implement TIP in a Coda/BSD/Mach operating system. Then we will identify and instrument applications to provide the required hints to the system.

References

[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.

[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.

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

[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.

[Miller91b] Miller, Ethan, private communication.

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

[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.

[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.

[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.

[Satyanarayanan85] Satyanarayanan, M., Howard, J. Nichols, D., Sidebotham, R., Spector, A., West, M., "The ITC Distributed File System: Principles and Design," *Proceedings of the Tenth Symposium 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 Transactions on Computers*, V C-39 (4), April 1990.

[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.