Appears in *Proc. Third International Conf. on Parallel and Distributed Information Systems,* Austin, TX, Sept. 28-30, 1994, pp. 7-16

Exposing I/O Concurrency with Informed Prefetching

R. Hugo Patterson†, Garth A. Gibson*

†Department of Electrical and Computer Engineering
*School of Computer Science
Carnegie Mellon University, Pittsburgh PA 15213

Abstract

Informed prefetching provides a simple mechanism for I/O-intensive, cache-ineffective applications to efficiently exploit highly-parallel I/O subsystems such as disk arrays. This mechanism, dynamic disclosure of future accesses, yields substantial benefits over sequential readahead mechanisms found in current file systems for non-sequential workloads. This paper reports the performance of the Transparent Informed Prefetching system (TIP), a minimal prototype implemented in a Mach 3.0 system with up to four disks. We measured reductions by factors of up to 1.9 and 3.7 in the execution time of two example applications: multi-file text search and scientific data visualization.

1: Introduction

Reducing program execution time is commonly the reason for purchasing new, faster processors. However, for programs that process stored data, faster processors do not linearly decrease execution time unless they are coupled with proportionately faster storage systems. Because storage performance is increasing more slowly than processor performance, data-intensive programs do not benefit as much as one might expect from a faster processor. To directly combat this limitation, new storage systems are increasing disk parallelism, usually in the form of Redundant Arrays of Inexpensive Disks (RAID) [Patterson88, Gibson91].

RAID subsystems exploit data striping to provide high throughput: high data rate for large parallel transfers and I/O concurrency and disk load balancing for large numbers of small accesses [Kim86, Livny87]. Unfortunately, RAID subsystems cannot reduce the access latency of isolated small reads and can increase small write latencies [Chen90, Stodolsky93]. The situation is analogous to that of parallel processors which are effective when applied to large jobs distributed over the processors and to many independent small jobs running in parallel. But, parallel processors do not reduce the execution time of a serial program any more than RAIDs reduce the latency of a small access. Since serial streams of small accesses dominate many important workloads, access latency is an increasingly important component of overall system performance. Distributed file systems further increase its importance by adding transfer and server overheads to the storage access time [Sandberg85, Satya85].

Caching recently used file blocks can provide fast access when a program's workload is small or has high locality. But, growing file sizes and the large volume of read-once data limit the effectiveness of file caching [Baker91]. Beyond caching, prefetching soon-to-be-needed file blocks is the best method of reducing storage access time [Feiertag71, McKusick84].

To be most successful, prefetching should be based on the *knowledge* of future accesses often available within applications. By passing hints to the file system, applications can disclose this information to lower levels of the system. There, it may be combined with global knowledge of the competing demands for system resources. Thus informed, a file system can transparently prefetch needed data and optimize resource utilization. We call this *informed prefetching*.

As presented previously, informed prefetching reduces application execution time through three mechanisms [Patterson93].

Exposure of an application's I/O concurrency: The primary advantage of informed prefetching is its ability to perform multiple I/O accesses in parallel so that applications and users spend less time waiting for these accesses to complete. Because informed prefetching systems know what to prefetch, they can utilize resources less timidly than uninformed prefetching systems. It is this ability of informed prefetching to expose and exploit I/O concurrency that enables it to convert the high throughput of parallel I/O technologies to the lower access latencies these storage technologies cannot directly provide.

This work was supported in part by the National Science Foundation under grant number ECD-8907068 and by an IBM Graduate Fellowship.

Increased storage efficiency: Because informed prefetching systems prefetch aggressively, they can fill otherwise empty I/O queues with low-priority accesses and create opportunities for storage subsystem optimizations [Seltzer90].

Informed cache management: Knowledge of future I/O requests can be used to hold on to needed blocks and avoid disk accesses altogether. Thus, an informed cache can outperform a standard LRU cache even without prefetching [Chou85, Korner90, Cao94].

To obtain the full benefit of these mechanisms, application hints to an informed prefetching system should not advise particular lower-level policies or actions. Instead, they should *disclose* knowledge of future accesses using the same abstractions and semantics that the application later uses for I/O requests. Such disclosure does not violate sound software engineering principles of modularity and, as will be shown in Section 3.2.2, allows hints to be passed through multiple software layers. Further, in contrast to advice, disclosure provides portability and the flexibility needed to support global resource optimizations.

In this paper, we present a prototype *Transparent Informed Prefetching* (TIP) system. The primary goal of our prototype, and the thrust of this paper, is to explore the large benefits arising from the exposure of I/O concurrency. We do not address storage efficiency or cache management further in this paper.

We evaluate our prototype with two I/O-intensive, cache-ineffective applications: multi-file text search and data visualization. They share the following key features:

- the amount of data they manipulate are large and growing,
- they usually do little computation per data byte accessed,
- they are easily and generally coded as sequential (single-threaded) applications,
- they do not reuse blocks or they flush even huge caches between reuse,
- they perform significant numbers of non-sequential accesses, either because they access blocks from many files or because they access non-sequential blocks in a large file.

Together these features indicate that multi-file text search and data visualization do not benefit much from disk arrays, large buffer caches or traditional per-file and per-disk readahead. Informed prefetching file systems, however, see the accesses of even these applications as sequential in the "hint address space" and exploit this sequentiality to overlap multiple accesses and fully utilize disk parallelism.

Section 2 describes the implementation of our TIP prototype. Section 3 describes our test applications in detail and reports their performance in our system. Sections 4 and 5 discuss related work and conclusions.

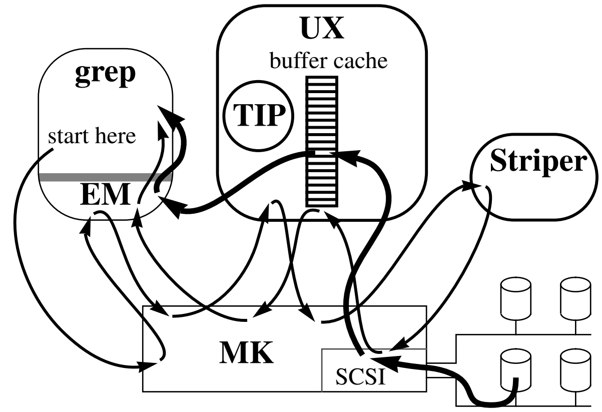

Figure 1: Experimental system. The prototype was implemented in the Mach 3.0 operating system which is decomposed into a user-level Unix server, *UX*, and a microkernel, *MK*. To maintain binary compatibility with earlier operating systems, an emulator, *EM*, is added in the address space of applications. A separate user-level Striper process intercepts disk requests from *UX* and redirects them to the appropriate disk in an array of up to four disks. The flexibility of the Mach 3.0 operating system made this possible, but this flexibility comes at the cost of considerable overhead. Thin arrows indicate the control path that a read system call traverses to get data from disk, while thick arrows represent data copies.

2: The TIP prototype

As Figure 1 shows, we built our prototype Transparent Informed Prefetching system (TIP), in a Mach 3.0 system [Accetta86, Golub90] augmented with disk striping software (UX version 42, MK version 83). We installed TIP in the file buffer cache of the 4.3BSD Unix Fast File System (FFS) [McKusick84] in the UX server where it has the opportunity to execute whenever a buffer or disk was accessed.

The system ran on a DECstation 5000/200 with 32 megabytes of RAM, two SCSI strings and up to four IBM 0661 Lightning disks formatted with a file block size of 8Kbytes. The Striper process striped data across the disks with a stripe unit of 128 512-byte sectors, or eight 8 Kbyte file blocks. There were 400 8 Kbyte buffers in the file cache.

Although Mach 3.0 may be inefficient in terms of instruction counts and memory cache behavior [Chen93] particularly when an application's primary activity is moving lots of data [Druschel93], it allowed us to add TIP and the disk striping functionality outside of the kernel. Moreover, operating system overheads have little impact on the non-TIP execution time of disk-bound applications, though they do limit speedup once TIP has removed the disk bottleneck. For this reason, and because our DEC 5000/200 testbed is about four times slower than current fast workstations, we believe our results are pessimistic for today and even more so for the future.

Our TIP implementation is deliberately minimal. It defines only enough functionality to exploit the types of precise disclosure easily generated by our test applications. While we are aware of the hint language richness and prefetching control problems that could be explored with synthetic workloads, we prefer to develop primitive mechanisms and allow higher level layers to build richer functionality on these primitives [Steere94]. Accordingly, an important goal in the development of our TIP prototype was to provide a platform for experimenting with I/O-intensive applications.

For ease of implementation, hints are passed to a pseudo-device named "/dev/tip" through the Unix I/O control (ioctl) mechanism. Each hint indicates a sequence of accesses within one file. If multiple files will be accessed, a hint for each should be given in the order these files will be accessed. Currently, hints may specify either sequential or non-sequential access.

A *tipio_seq* hint indicates that a file will be read sequentially in its entirety. Its only parameter is the name of the file to be read.

A *tipio_seg* hint delivers a list of subsequences of a file that will be accessed in the given order. Its parameters are the file name and a list of <offset, length> couples.

Note that these hints disclose an application's future behavior without reference to specific file system behavior such as caching or prefetching. Additionally, they are specified using the same abstractions that will later be used to access the file: file name, byte offset and byte count.

The simplest way to describe the structure of the TIP file system is follow the actions (possibly) triggered by a hint. This description assumes familiarity with the BSD filesystem [Leffler89].

When /dev/tip receives an application's hint, it adds it to the end of a prefetch queue. The *tip_prefetch()* routine draws a hint from this queue and uses the standard FFS routines to resolve the file name to an inode or file handle. TIP resolves file names lazily, on dequeue rather than enqueue, to avoid pinning more than one entry in the inode cache.

Given a resolved file name, *tip_prefetch()* iterates through the requested blocks allocating a buffer for each block not found in the buffer cache, marking each buffer "prefetch", and enqueuing the necessary disk access. When a prefetch access completes, the buffer containing its data is treated as though it was a FFS readahead block, and may be aged out of the cache. If the block becomes the target of a read system call, its prefetch mark is cleared and it becomes indistinguishable from blocks demand-fetched into the cache.

We use the prefetch flag as an accounting measure to limit the number of buffers containing prefetched but unread data. Without this throttling, TIP could prefetch too far ahead and flush its own prefetch buffers before they can be read. The *tip_prefetch()* routine, once initiated, iterates through the prefetch queue until it runs out of hints or it reaches a ceiling on the number of prefetch buffers. To reinitiate prefetching, *tip_prefetch()* is called whenever the count of prefetch buffers or hints changes. In particular, it is called whenever a buffer containing prefetched data is read or whenever a prefetch buffer ages out from the cache unread. In the experiments described in the next section, we set the prefetch ceiling at 150 out of the 400 cache buffers.

3: Informed prefetching case studies

To supply our TIP prototype with hints, we have instrumented two applications. The multi-file text search application, *grep*, searches many files with a simple sequential access pattern. We have instrumented grep to give hints across all the files it will search. At the other extreme is *XDataSlice*, a data visualization package that accesses blocks non-sequentially from a single very large file. We have instrumented it to give hints about the blocks within the file it will access. XDataSlice also serves to illustrate how hints may be passed through layers of software without violating software modularity.

The primary performance measure of our system is the elapsed execution time of applications giving hints. Thus, we report the elapsed execution times in seconds when running each application with and without giving hints. The times reported are averages of a number of runs and the sample standard deviation is reported in parentheses. As an indication of the benefits of I/O concurrency, the tests were run on arrays of one, two, three, and four disks. The speedup reported is simply the ratio of the average non-TIP to TIP-enhanced execution time.

3.1: *Grep*: prefetching across files

The simplest and most common file access pattern of Unix applications is whole file sequential read. A good example is grep which searches files for a specified pattern of text. Typically, grep is asked to search all the files on a list which is passed as an argument to grep at invocation.

Since the arguments to grep completely determine the file accesses grep will make, it is simple to loop through the file list and pass a *tipio_seq* hint to TIP for each file. Because FFS readahead heuristics work well for sequen-

Files searched		Time (seconds)			
		1 disk	2 disks	3 disks	4 disks
X11 includes 139 files 187 blocks	no TIP	3.21 (0.15)	2.98 (0.11)	2.92 (0.08)	3.02 (0.22)
	with TIP	2.35 (0.08)	1.90 (0.04)	1.85 (0.04)	1.85 (0.03)
	Speedup	1.37	1.57	1.58	1.63
bboard, mail 158 files 218 blocks	no TIP	3.62 (0.27)	3.40 (0.13)	3.19 (0.15)	3.29 (0.13)
	with TIP	2.68 (0.11)	2.17 (0.05)	2.12 (0.05)	2.15 (0.03)
	Speedup	1.35	1.57	1.50	1.53
source code 57 files 152 blocks	no TIP	2.11 (0.09)	2.16 (0.09)	2.02 (0.08)	2.24 (0.07)
	with TIP	1.66 (0.11)	1.24 (0.06)	1.21 (0.04)	1.20 (0.03)
	Speedup	1.27	1.74	1.67	1.87
man pages 834 files 892 blocks	no TIP	16.80 (0.37)	15.26 (0.27)	15.17 (0.34)	14.75 (0.26)
	with TIP	13.71 (0.37)	10.72 (0.14)	10.46 (0.19)	10.44 (0.44)
	Speedup	1.23	1.42	1.45	1.41

Table 1: Time to search a number of small files for a text string. This table shows the total execution time for searches through four sets of files. The numbers were collected by performing each search in turn and repeating the sequence thirty times both with and without TIP. The numbers in parentheses are sample standard deviations. The tests were repeated with the data striped over an array of 1, 2, 3, and 4 disks. The speedup is the ratio of non-TIP to TIP-enhanced execution time. The results show that, without TIP, *grep* performance is nearly flat no matter how many disks are in the array. By overlapping CPU with I/O, TIP achieves speedups of from 1.2 to 1.4 on a single disk. By performing I/Os concurrently, TIP achieves speedups of from 1.4 to 1.9 on four disks. TIP performance might have been higher, except that the CPU becomes the bottleneck, as shown in Table 2.

tial accesses, it might seem that TIP is not needed for this application. However, such single-file heuristics never get a chance to engage when, as is often the case, the files searched are small. In contrast, TIP knows about multiple files it can prefetch before they are even opened. In this way, TIP exposes concurrency across files that simple heuristics cannot.

To evaluate the performance of TIP, we instrumented a fast version of grep called *agrep*, developed by U. Manber at the University of Arizona. Before agrep reads any files, it loops through the list of files to validate its arguments. It was a simple matter to add a few lines in this loop to pass the *tipio_seq* hint to TIP.

We then used the modified agrep to search four sets of files. The first set consisted of 139 include files for the X11 window system. These files were stored in 187 file blocks and block fragments 8KB or less in size. The second set consisted of 158 mail messages and archived bulletin board posts including a couple of posts of Frequently Asked Questions. The third set was a directory full of source files. The last set was three chapters of Unix manual pages. We ran the four-test sequence thirty times both with and without TIP. When running with three and four disks, our test hardware occasionally suffered from an erroneous bus reset, adding about fifteen seconds to a run. Tests suffering from such glitches were eliminated from the sample leaving between twenty-five and thirty trials for each test. Table 1 summarizes the results of these experiments.

The first result is that grep does not, by itself, significantly benefit from the disk array. Looking across the 'no TIP' rows in Table 1 reveals that grep execution times without TIP are essentially flat. The parallel hardware of a disk array does not benefit applications that present a serial I/O workload.

The second result is that, with one disk, TIP achieves speedups of 1.2 to 1.4 by overlapping I/O with computation. As more disks are added, TIP is able to leverage its knowledge of multiple files to deliver speedups ranging from 1.4 to 1.9, nearly halving the execution time.

A surprising result is that, as Table 2 shows, grep becomes CPU-bound with just two disks. For this application, the CPU is unable to take advantage of additional disks. CPU utilization would be 100%, instead of 93%, except that, as discussed in Section 2, the current TIP implementation uses the standard, blocking FFS file name resolution routines, and, hence, does not prefetch metadata such as directory and inode blocks.

Table 2 further reveals that the UX server, not grep, is the largest consumer of CPU time. Partially, this reflects

Files searched from 2 disk array		Time (seconds)						CPU utiliza-tion
		grep	UX	MK	Striper	Idle	Total	
bboard, mail	no TIP	0.40	0.88	0.08	0.15	1.86	3.40	45%
	with TIP	0.56	1.25	0.09	0.09	0.15	2.17	93%

Table 2: CPU usage during a *grep* text search. This table reports CPU usage statistics collected during an example test reported in Table 1. Shown are the CPU times for the *grep* task, the UX server task, the Striper disk array task and the idle thread. The numbers don't quite add up both because of measurement truncation and because there are other tasks on the system not being measured. Without TIP, CPU utilization during a *grep* text search is low. With TIP, CPU utilization is nearly 100%. These numbers reveal the UX filesystem code as a major CPU bottleneck, in part because the current version of TIP duplicates name resolution work in UX. Fortunately, the TIP-induced reduction in idle time far outweighs this additional overhead.

the cost of multiple data copies and other overheads of the microkernel structure. The main cost, however, is the basic work required to open files and guide data through the file system. TIP compounds this problem by resolving file names twice: once when hint processing implicitly opens the file, and later when the application explicitly opens the file. We expect that by caching the result of TIP's name resolution, we can eliminate most of the TIP-induced system overhead.

In the larger picture, CPU performance is increasing rapidly. Already, workstations four times faster than the DECstation 5000/200 are widely available. Such workstations should dramatically reduce system overhead and exploit the concurrency of much larger arrays. This trend will only increase the benefits of TIP.

3.2: *XDataSlice*: non-sequential prefetching within a file

Traditional, sequential readahead can effectively reduce execution time for I/O-intensive applications that emphasize sequential access into large files. But, it does not benefit, and may penalize, applications that feature non-sequential file access. XDataSlice is one such application.

XDataSlice (XDS) is a data visualization package developed by the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign. XDS allows its user to select and view a false-color representation of an arbitrary planar slice from a 3-D scientific dataset. Such viewable datasets are generated by a variety of applications such as airplane and automobile airflow simulation, climate and pollution simulation, and magnetic resonance imaging equipment. Because engineers, scientists, and doctors periodically call for much greater scope and resolution for their datasets, it is not uncommon for these datasets to be very large. For example, a cube of 300^3 32-bit floating point data elements requires more than 100 megabytes of storage, and a cube of 800^3 elements requires almost 2 gigabytes. Unfortunately, too often, science must defer to computer specifications such as main memory size and accounting policies when determining dataset size. For example, NCSA's original XDS does not support dynamic access to large datasets; if your dataset does not fit in memory, you can't use XDS to view it.

One of our goals in studying data visualization is to reverse this technology-drives-science dynamic by making graceful and limited the performance degradation and code complexity that results from increasing dataset size. Informed prefetching combines a simple, uniform interface with transparent exploitation of storage concurrency and a global throttle on memory and storage resource commitments. Thus, programs written to dynamically load data and disclose their accesses to an underlying informed prefetching system can transparently benefit from increased storage concurrency even if their access patterns are single-threaded and non-sequential. To demonstrate this, we first extended XDS to load slices of data dynamically from disk. These extensions have been delivered back to NCSA and are available in their contributed-code tree.

We next added disclosing hints for TIP to exploit. Our selection of XDS as an example application for informed prefetching was fortuitous because XDS has an internally layered structure. In adding hints to XDS, we show how to use layered disclosure to pass optimization information through layers of software without violating the integrity of module interfaces.

We now describe the structure of XDS, how we added dynamic loading and disclosing hints to it, and its performance on a 300^3 element dataset striped over multiple disks.

3.2.1: *XDataSlice* organization: XDS reads data from files stored in a self-describing format called the Hierarchical Data Format (HDF). NCSA has developed a library of routines to simplify access to HDF files and to enforce the HDF standard format. XDS binds this HDF library between itself and the file system. The HDF library is itself composed of two layers: low-level storage management in the *H* layer and scientific dataset object management in the *DFSD* layer.

A single HDF file may contain many data objects such as raster images, raw scientific data, or the format of numerical data. But, to the low-level H layer of the HDF library, all are just arrays of bytes with a name. The high-level DFSD layer of the library refers to these elemental data objects by name and may request the H layer to deliver logical byte ranges from within individual objects. It is up to the H layer to allocate file space and keep track of the location and size of all the data objects.

The DFSD layer groups a number of elemental data objects together to form a scientific data set. These objects include one holding the raw scientific data and others holding dataset metadata such as the dimensions of the data, the data type, and units and labels for the axes. Applications built on top of the DFSD layer refer to the scientific data set as if it were one complex data object with many typed data fields.

The XDataSlice code, operating outside of the HDF library, uses the DFSD interface to determine dataset size so it can allocate adequate memory. Then, in the original code, it uses this interface to read the entire dataset into memory. To render a slice of the dataset, XDS loops through all the pixels in the slice mapping each to a data element stored in memory. False color is applied based on a data element's value and the resulting bitmap is displayed in an X window.

We extended this basic package to load data dynamically from large datasets. Standard 3-D HDF data objects are written to disk in row-major order. This has the disadvantage of requiring that the entire data object be read to render a slice that cuts across all rows. To make loading arbitrary slices efficient, we reorganized the object into submatrices as shown in Figure 2 and updated the DFSD layer to export a blocked view of the scientific data object. We then modified XDS to first determine which blocks are needed and load them into memory before rendering the requested slice in the usual way. All of these changes add useful functionality and are independent of TIP.

3.2.2: Extending HDF to issue and deliver hints to TIP: For this new version of XDataSlice to take advantage of TIP, it must disclose its expected accesses. Since the primary benefit of TIP is exposing I/O concurrency, the source of hints should be at a level aware of a large volume of work before it is actually requested. There are a number of possibilities, but a simple and natural choice is to issue hints within the DFSD layer of the library since

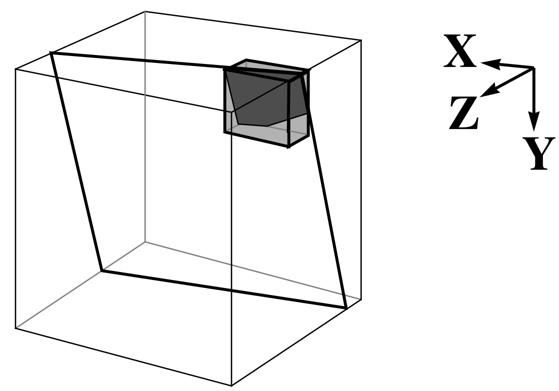

Figure 2: Blocked dataset storage layout. To facilitate the retrieval of arbitrary slices of data, the dataset is partitioned into submatrices each stored in its own file system block. The shaded cube above shows one such block and its share of a slice through the dataset. The blocks themselves are stored in row-major order, Z-axis first. Thus, sequential disk access favors slices in the Y-Z plane. To compensate, the blocks are asymmetrical, so that rendering slices in the X-Y plane requires fewer total blocks.

XDS hands this layer a list of the needed blocks. This list is an excellent hint for TIP.

Unfortunately, the DFSD layer cannot directly pass the list of blocks on to TIP. Even after the DFSD layer translates block coordinates into logical offsets within the scientific data object, it does not know the offsets within the enclosing file. It relies on the lower H layer for addressing and accessing files. It is possible for the DFSD layer to "peek beneath the covers" of the H layer to find the offsets of the objects about which it wishes to issue hints, but this exposure of the H layer internal data structures would violate the design's modularity and expose the implementation to unforeseen bugs when the H layer is independently modified at some later time.

A much better solution is to formally incorporate a path for the disclosure of optimization information into the interface to the H layer of the library [Kiczales92]. We have done this by adding a *Hhint()* routine. It accepts hints from higher layers of the library in the language used by the rest of H layer: offsets and lengths within data objects. Hhint maps data object offsets to file offsets and passes these file offsets to TIP using the *tipio_seg* hint. Such disclosure is consistent with the module interfaces already in place; the DFSD layer issues hints about data objects and the Hhint routine translates these data object hints into files access hints which it discloses directly to TIP. The modularity of the HDF library is not a barrier to hints that disclose.

Slice rendered		Time (seconds)			
		1 disk	2 disks	3 disks	4 disks
Y-Z (722 blocks)	no TIP	5.21 (0.06)	5.25 (0.05)	5.17 (0.08)	5.18 (0.73)
	with TIP	5.12 (0.07)	4.27 (0.05)	4.32 (0.06)	4.36 (0.05)
	Speedup	1.02	1.23	1.20	1.19
X-Z (722 blocks)	no TIP	5.86 (0.05)	6.07 (0.09)	6.17 (0.04)	6.36 (0.08)
	with TIP	5.84 (0.06)	4.36 (0.05)	4.43 (0.07)	4.43 (0.04)
	Speedup	1.00	1.39	1.39	1.43
X-Y (361 blocks)	no TIP	8.16 (0.10)	8.40 (0.19)	8.16 (0.11)	8.23 (0.15)
	with TIP	7.86 (0.04)	3.49 (0.07)	2.56 (0.06)	2.23 (0.03)
	Speedup	1.04	2.41	3.19	3.69

Table 3: Time to visualize a slice of a 3-D scientific dataset. This table shows the response time of XDataSlice when it renders a slice of a 3-D dataset along three planes: Y-Z, X-Z, and X-Y. All values are computed as the average of 10 executions with sample standard deviations shown in parentheses. Each plane was tested with and without TIP when the dataset was striped over 1, 2, 3, and 4 disks. Speedup is the ratio of the slice's response time without TIP and with TIP. These results show that XDataSlice cannot exploit the disk array without TIP and that with only one disk, XDataSlice is so I/O-bound that TIP is unable to overlap much computation. With as little as two disks, however, TIP provides speedups of 1.2 to 2.4, saturating the CPU for the Y-Z and X-Z planes. The X-Y plane continues to benefit from increased disk parallelism, saturating the CPU at four disks with a speedup of 3.7.

3.2.3: XDataSlice performance: To evaluate the performance of a TIP-enhanced XDataSlice we measured the time to render slices of a 3-D scientific dataset containing 300^3 32-bit floating point numbers. The file containing this dataset is a little over 100 megabytes in size. Because the Decstation 5000/200 on which we ran our tests contained only 32 megabytes of RAM, this test depends on the ability of XDS to load slices dynamically from disk. Though XDS can render slices in any orientation, for convenience we rendered slices normal to one of the three axes. For each orientation, we measured the time required to render each of 10 random slices. The experimental setup, described in Section 2, is the same one used for the multi-file search experiments.

Table 3 shows speedups in the time to render a slice that range from 1.0 (no speedup), in the case of X-Z planes with the dataset contained on one disk, to 3.7 (meaning that TIP-enhanced response time is 27% non-TIP response time) in the case of X-Y planes with the dataset striped over four disks.

With only one disk, rendering any slice is so I/O-bound that overlapping computation with I/O has no significant effect. Striping the data set over just two disks yields speedups of 1.2 to 2.4. These speedups result from overlapping concurrent disk accesses. In the Y-Z and, to a lesser extent, X-Z plane most of the blocks accessed are sequential on disk, there is little positioning time to overlap, so the TIP-increased parallelism has raised the total storage bandwidth to its maximum. The CPU saturates moving the data through the system to the application. Increasing the number of disks over which data is striped does not further improve the rendering time of these slices. To reduce their rendering time below 4.3 seconds we would need to upgrade the DECstation 5000/200's with a faster processor, upgrade its memory system for faster copies, or significantly reduce the operating system overhead.

Speedups are more pronounced in the X-Y plane because its access pattern on disk is much more non-sequential, causing its disk accesses to return data at a lower bandwidth (per disk). This allows the increased storage concurrency of three and four disks to be effective because the bandwidth maximum of the operating system structure is not as quickly reached. However, with the data striped over four disks, accessing X-Y planes saturates the CPU; additional disks would not be effective. Note that the benefits of TIP are greatest when the disk bottleneck is most severe and the access latencies resulting from long seeks are largest.

The faster response time of the X-Y plane with TIP at saturation results from the asymmetry of data submatrices assigned to each filesystem block. The asymmetrical dimensions of 16x16x8, roughly balance the disk read latency for the three planes when TIP is not engaged. Because the dataset dimensions in elements are 300x300x300, the dataset dimensions in blocks are 19x19x38. This means that slices in the Y-Z and X-Z planes must read 19x38=722 blocks while slices in the X-

Y plane read only 19x19=361 blocks. With TIP-generated disk concurrency, the positioning time delays which lead to this asymmetrical blocking are completely overlapped, the operating system is driven to bandwidth saturation, and the smaller number of blocks per X-Y plane halves the response time relative to other planes.

Notice also that for the X-Y plane, TIP-derived speedups are superlinear for two and three disks. This occurs because the cost of data copying and computation not overlapped in the non-TIP case is large enough that overlapping it with "N" disks leads to speedups approaching "N+1".

4: 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 [Sun88].

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 informed prefetching.

Many researchers have looked into prefetching based on access patterns inferred from the stream of user I/O requests [Curewitz93, Kotz91, Tait91, Palmer91, Korner90]. Kotz looked at intelligent prefetching for MIMD multiprocessors with scientific workloads. He extended the applicability of readahead to non-sequential, but regular, accesses within one file by predicting future accesses based on previously observed access patterns. He achieved significant performance improvements for stride access patterns in large scientific datasets.

A drawback to speculative prefetching based on prior observations is that it risks hurting, rather than helping, performance [Smith85]. When a prediction is wrong, prefetching the unneeded data consumes valuable resources which could have been used for accessing needed data or storing recently used and soon-to-be-reused data. Kotz observed this phenomena in some of his tests, and its impact would be more severe if his techniques were applied to less regular accesses.

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. However, Korner uses traces of file system activity to predict future access patterns.

Researchers have also 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 cache 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.

5: Conclusions

The primary benefit of informed prefetching, its ability to increase the I/O concurrency of single-threaded applications, is perhaps best illustrated by an analogy to parallel processing.

Parallel processors provide the hardware infrastructure to relieve the bottleneck of a serial processor. But, existing, serial programs do not automatically benefit from parallel hardware. Converting old serial algorithms and inventing new parallel algorithms to take advantage of parallel processors is hard. But, without this software effort, parallel processors are of limited use.

Similarly, disk arrays provide the I/O hardware infrastructure to relieve the bottleneck of slow disk drives. But, applications do not automatically benefit from arrays. In this paper, we show that informed prefetching exposes I/O concurrency so that existing and future applications can transparently utilize the parallelism of disk arrays.

We have presented our experience with two applications running on TIP, our prototype informed prefetching filesystem. TIP achieves speedups of 1.4 to 1.9 for text searches through large numbers of small files and speedups of 1.2 to 3.7 when visualizing slices from a large scientific dataset. In all cases, speedups were limited only by the throughput of the CPU, not by the latency of disk accesses. Our results suggest that the combination of TIP with disk arrays will be able to satisfy the I/O needs of ever-faster processors.

We have also shown that such substantial benefits do not depend on tightly knit and tuned applications and systems. Instead, they are available to structured, modular programs that disclose optimization information in a manner consistent with their established module interfaces.

Just as the efficient utilization of parallel processors makes possible the computation of ever larger and more important problems, so, we hope, will the exposure and exploitation of I/O concurrency through informed prefetching make possible work with ever larger and more important datasets.

Much work remains to be done in this area. Most immediately, a member of our group is adding asynchronous name resolution to the TIP prototype so that we can prefetch metadata in addition to user data. Of the three benefits of informed prefetching, this paper addresses only one, exposure of I/O concurrency. We have begun research into the other two, informed cache management and increased I/O efficiency. The former emphasizes global resource allocation given partial foreknowledge and the latter emphasizes disk scheduling policies given deep queues of low-priority accesses.

More broadly, for TIP to be generally useful, a wide range of applications need to generate hints. In some cases, this is best done manually or in libraries. In the long run, we believe that compiler-created runtime code is the best method of generating hints. Finally, mechanisms are needed to ensure that performance gracefully degrades when hints are imprecise or incorrect. These mechanisms should allow inferences from application traces to be given as hints when better information is unavailable.

6: Acknowledgment

We are deeply grateful to Daniel Stodolsky for his extensive help with Mach 3.0 and for the striping driver he built for us. We also appreciate the many helpful discussions we have had with M. Satyanarayanan. We would like to thank them both as well as Lily Mummert for reading drafts of this paper. Jaiwen Su deserves our thanks for taking on the task of adding asynchronous name resolution to TIP.

7: References

[Accetta86] Accetta, M.J., et al, "Mach: A New Kernel Foundation for Unix Development," *Proc. of the Summer 1986 USENIX Conference*, 1986, pp. 93-113.

[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.

[Cao94] Cao, P., Felten, E.W., Li, K., "User Level File Caching Policies," *Proc.of the Summer 1994 USENIX Conference*, Boston, MA, 1994, pp. 171-182.

[Chen90] Chen, P. M., Gibson, G. A., Katz, R. H., Patterson, D. A., "An Evaluation of Redundant Arrays of Disks Using an Amdahl 5890," *Proc. of the 1990 ACM Conf. on Measurement and Modeling of Computer Systems* (SIGMETRICS), Boulder CO, May 1990.

[Chen93] Chen, J.B., Bershad, B.N., "The Impact of Operating System Structure on Memory System Performance", *Proc. of the 14th Symp. on Operating System Principles*, 1993, pp. 120-133.

[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.

[Curewitz93] Curewitz, K.M., Krishnan, P., Vitter, J.S., "Practical Prefetching via Data Compression," *Proc. of the 1993 ACM Conf. on Management of Data (SIGMOD)*, Washington, DC, May, 1993, pp. 257-66.

[Druschel93] Druschel, P., Peterson, L.L., "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility," *Proc. of the 14th Symp. on Operating System Principles*, 1993, pp. 189-202.

[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.

[Golub90] Golub, D., Dean, R., Forin, A., Rashid, R., "Unix as an Application Program," *Proc.of the Summer 1990 USENIX Conference*, 1990, pp. 87-95.

[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.

[Kiczales92] Kiczales, G., "Towards a New Model of Abstraction in the Engineering of Software," *Proc. of the IMSA '92 Workshop on Reflection and Meta-level Architectures*, 1992.

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

[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.

[Leffler89] Leffler, S.J., McKusick, M.K., Karels, M.J., Quarterman, J.S., "The Design and Implementation of the 4.3BSD Unix Operating System," Addison-Wesley, New York, 1989.

[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.

[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.

[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.

[Patterson93] Patterson, R.H., Gibson, G.A., Satyanarayanan, M., "A Status Report on Research in Transparent Informed Prefetching," *Operating Systems Review*, V 27 (2), April, 1993, pp. 21-35.

[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.

[Sandberg85] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyon, B., "Design and Implementation of the Sun Network File System," *Proc. of the Summer 1985 USENIX Conference*, Portland, OR, June 1985, pp. 119-30.

[Satya85] 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.

[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.

[Steere94] Steere, D., Satyanarayanan, M., "A Case for Dynamic Sets," in preparation.

[Stodolsky93] Stodolsky, D, Gibson, G., "Parity Logging Disk Arrays," *ACM Trans. on Computer Systems*, V 12 (3), August 1994.

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

[Sun88] Sun Microsystems, Inc., *Sun OS Reference Manual*, Part Number 800-1751-10, Revision A, May 9, 1988.

[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.