Informed Prefetching and Caching

R. Hugo Patterson*, Garth A. Gibson,
Eka Ginting, Daniel Stodolsky, Jim Zelenka

11 May 1995

CMU-CS-95-134

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

The underutilization of disk parallelism and file cache buffers by traditional file systems induces I/O stall time that degrades the performance of modern microprocessor-based systems. In this paper, we present aggressive mechanisms that tailor file system resource management to the needs of I/O-intensive applications. In particular, we show how to use application-disclosed access patterns (hints) to expose and exploit I/O parallelism and to allocate dynamically file buffers among three competing demands: prefetching hinted blocks, caching hinted blocks for reuse, and caching recently used data for unhinted accesses. Our approach estimates the impact of alternative buffer allocations on application execution time and applies a cost-benefit analysis to allocate buffers where they will have the greatest impact. We implemented informed prefetching and caching in DEC's OSF/1 operating system and measured its performance on a 150 MHz Alpha equipped with 15 disks running a range of applications including text search, 3D scientific visualization, relational database queries, speech recognition, and computational chemistry. Informed prefetching reduces the execution time of the first four of these applications by 20% to 87%. Informed caching reduces the execution time of the fifth application by up to 30%.

*Department of Electrical and Computer Engineering, Carnegie Mellon University

http://www.cs.cmu.edu/Web/Groups/PDL

This research was partially supported by the National Science Foundation under grant number ECD-8907068, the Advanced Research Projects Agency under contract number DABT63-93-C-0054, and an IBM Graduate Fellowship. 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 NSF, ARPA, IBM, or the U.S. government.

Keywords: caching, prefetching, file systems, hints, I/O, resource management

1 Introduction

The management of disk accesses and main-memory file buffers is one of the basic functions of a file system. The goal in disk management is to start accesses early to avoid latency and maximize disk throughput. In file-buffer management, the goal is to exploit locality in the access stream to minimize the number of disk accesses.

In this paper, we show how to use hints from I/O-intensive applications to prefetch aggressively enough to eliminate I/O stall time while maximizing buffer availability for caching. Moreover, we show how to allocate cache buffers dynamically among competing hinting and non-hinting applications for the greatest performance benefit.

There are three principle reasons that we revisit file-system resource management:

- the increasing availability of storage parallelism and its frequent underutilization,
- the increasing occurrence of applications dependent on file-access performance, and
- the ability of such I/O-intensive applications to offer hints beneficial to their file-access performance.

Storage parallelism is increasingly available because of the growth in disk array products and striping device drivers. These hardware or software arrays promise the I/O throughput needed to balance ever-faster CPUs by distributing the data of a single file system over many disk arms [Salem86]. Embarrassingly parallel I/O workloads benefit immediately. Very large accesses benefit from parallel transfer and multiple concurrent accesses benefit from independent disk actuators. Unfortunately, many I/O workloads are not at all parallel but instead consist of serial streams of comparatively small, non-sequential accesses. Such workloads access one disk at a time while idling the other disks in an array. Disk seek and rotational latencies dominate service time. Disk arrays, by themselves, do not improve I/O performance for these workloads any more than a multiprocessor improves the performance of a single-threaded program. Prefetching strategies adaptive to storage parallelism are needed to "parallelize" these workloads.

Applications are processing larger objects and executing their CPU-bound components faster. Unless file-cache miss ratios decrease in proportion to processor performance, Amdahl's law tells us that overall performance will increasingly depend on I/O performance [Patterson88]. Unfortunately, simply growing the cache does not increase cache-hit ratios as much as one might expect. For example, the Sprite group's 1985 caching study led them to predict higher hit ratios for larger caches. But, in 1991, when larger caches were installed, hit ratios were not much changed — files had grown just as fast as the caches [Ousterhout85, Baker91].

The problem is especially acute for the growing class of I/O-intensive applications. Examples include: text search, 3D scientific visualization, relational database queries, speech recognition, and computational chemistry. For most of these, the amount of data processed is large relative to file cache sizes, locality is poor or limited, accesses are frequently non-sequential, and I/O stall time is a significant fraction of total execution time.

Yet, for all of these applications, access patterns are largely predictable. This predictability could be used for explicit prefetching via asynchronous I/O but doing so would cripple resource management. First, the depth to which an application should prefetch depends on the throughput of the application, which varies as other applications place demands on the system. Second, asynchronously fetched data may flush useful data from the file cache. Finally, asynchronously fetched file blocks end up indistinguishable from any other block in virtual memory, requiring the programmer to be explicitly aware of virtual image size to avoid losing far more to paging than is gained from concurrent I/O.

Instead, we recommend using the predictability of these applications to inform the file system of future demands on it. Specifically, we propose that applications disclose their future accesses in hints to

the file system. We show how to use this information to exploit storage parallelism, balance caching against prefetching, and distribute cache buffers among competing applications.

We allocate buffers in a three-step process. First, we independently estimate the impact on execution time of alternative buffer allocations and deallocations. We find that there is a prefetch horizon beyond which there is no benefit in prefetching until the application catches up, that the cost in flushing hinted data is the overhead of prefetching it back into the cache later, and that shrinking the LRU cache increases the cache-miss ratio and thus the average I/O stall time for unhinted accesses. In step two, we normalize these estimates to the global standard of time saved or lost per unit of time that a buffer would be allocated. Finally, we use these standardized estimates to compare the alternatives and reallocate buffers to maximize the net reduction in execution time. For example, we compare the benefit of prefetching a block to the cost of either flushing a hinted block from a buffer or shrinking the LRU cache by one buffer, and prefetch only if it benefit is greater than the cost of the least valuable buffer.

The rest of this paper more fully explains and justifies this approach. Section 2 reviews prior work and describes disclosure-based hints. Section 3 describes an implementation in DEC's OSF/1 file system of informed prefetching according to disclosure (without informed caching), describes the applications in our test suite, and reports that four out of five of our test applications benefit substantially. Section 4 introduces our cost-benefit model for informed caching. Section 5 describes the implementation of informed prefetching and caching in DEC's OSF/1 file system. Section 6 reports that the fifth application benefits substantially from informed caching. Section 7 discusses future directions.

2 Prior work

Hints are a well established, broadly applicable technique for improving system performance. Lampson reports their use in operating systems (Alto, Pilot), networking (Arpanet, Ethernet), and language implementation (Smalltalk) [Lampson83]. Broadly, these examples consult a possibly out-of-date cache as a hint to short-circuit some expensive computation or blocking event.

In the context of file systems, historical information is often used for both file caching and prefetching. The ubiquitous least-recently-used cache replacement algorithm relies on the history of recent accesses to choose a buffer for replacement. For prefetching, the most successful approach is sequential readahead [Feiertag71, McKusick84]. OSF/1 is an aggressive example, prefetching up to 64 blocks ahead when it detects long sequential runs. Others, notably Kotz, have looked at detecting more complex access patterns and prefetching non-sequentially within a file [Kotz91].

At the level of whole files or database objects, a number of researchers have looked at inferring future accesses based on past accesses [Korner90, Kotz91, Tait91, Palmer91, Curewitz93, Griffioen94]. The danger in speculative prefetching based on historical access patterns is that it risks hurting, rather than helping, performance [Smith85]. As a result of this danger, speculative prefetching is usually conservative, waiting until its theories are confirmed by some number of demand accesses.

An alternate class of hints are those that express one system component's advance knowledge of its impact on another. Perhaps the most familiar of these occurs in the form of policy advice from an application to the virtual-memory or file-cache modules. In these hints, the application recommends a resource management policy that has been statically or dynamically determined to improve performance for this application [Trivedi79, Sun88, Cao94].

In large integrated applications, more detailed knowledge may be available. The database community has long taken advantage of this for buffer management. The buffer manager can use the access plan for a query to help determine the number of buffers to allocate [Sacco85, Chou85, Cornell89, Ng91, Chen93]. Ng, Faloutsos and Sellis's work on marginal gains considered the question of how much benefit a query would derive from an additional buffer. Their work stimulated the development of our approach to

cache management. It also stimulated Chen and Roussopoulos in their work to supplement knowledge of the access plan with the history of past access patterns when the plan does not contain sufficient detail.

Relatively little work has been done on the combination of caching and prefetching. One example, though, is [Cao95] which considers the question in the context of complete knowledge of future accesses.

We advocate a different form of hints based on advance knowledge which we call disclosure [Patterson93]. An application discloses its future resource requirements when its hints describe its future requests in terms of the existing request interface. For example, a disclosing hint might indicate that a particular file is going to be read sequentially four times in succession while a corresponding advising hint might specify that the named file should be prefetched and cached with a caching policy whose name is "MRU." Advice exploits a programmer's knowledge of application and system implementations to recommend how resources should be managed. In contrast, disclosure is simply a programmer revealing knowledge of the application's behavior.

Disclosure has three principle advantages over advice. First, because it expresses information independent of the system implementation, it remains correct when the application's execution environment, system implementation or hardware platform changes. Thus, disclosure is a mechanism for portable I/O optimizations. Second, because disclosure provides the evidence for a policy decision, rather than the policy decision, it is more robust. Specifically, if the system cannot easily honor a particular piece of advice — there being too little free memory to cache a given file, for example — there is more information in disclosure that can be used to choose a partial measure. Third, because disclosure is expressed in terms of the interface that the application later uses to issue its accesses; that is, in terms of file names, file descriptors, and byte ranges, rather than inodes, cache buffers, or file blocks, it conforms to software engineering principles of modularity.

In our implementations, disclosing hints are issued through an I/O-control system call. Hints list a series of byte ranges which will be accessed in the given order or indicate that the whole file will be accessed sequentially. The order hints are given indicates the order of the subsequent accesses. This is a minimal interface, but it meets the needs of the applications in our test suite. Much richer languages for expressing and exploiting disclosure include collective I/O calls [Kotz94], operations on structured files [Grimshaw91], or unordered sets [Ebling94].

Given disclosing hints, our informed prefetching and caching system delivers three primary benefits:

1. given a list of future accesses, informed prefetching can "parallelize" the I/O request stream according to available disk resources;
2. by prefetching more than one access per disk into the future, disk scheduling and batching can effectively increase each disk's throughput; and
3. given the list of future accesses, informed caching can outperform LRU caching independent of prefetching.

Our experimental results show all three benefits.

3 Informed prefetching

In this section we describe and evaluate a simple informed prefetching mechanism, TIP-1, implemented in the OSF/1 file system and applied to a suite of five I/O-intensive benchmarks. All are single-threaded and operate on data fetched from the file system as needed. All are I/O-bound in common usage. Four derive substantial benefit from prefetching alone. As we shall see later on, the fifth needs informed caching to obtain significant benefits.

3.1 Testbed implementation

Our testbed is a DEC 3000/500 workstation, containing a 150 MHz 21064 processor, 128 MB of memory and five KZTSA fast SCSI-2 adapters each hosting three HP2247 1GB disks. This machine runs DEC's OSF/1 version 2.0 monolithic kernel. OSF/1's file system contains a unified buffer-cache (UBC) module that trades memory between its file cache and virtual memory. To eliminate buffer cache size as a factor in our experiments, we fixed the cache at 12 MB (1536 8 KB buffers).

The system's 15 drives are bound into a disk array by a striping pseudo-device with a stripe unit of 64 KB. This device driver maps and forwards accesses to the appropriate per-disk device driver. Demand accesses are forwarded immediately, while prefetch reads are forwarded whenever there are fewer than five outstanding requests at the drive. Forwarding five prefetch requests ensures I/O subsystem resilience to transient CPU saturation while limiting priority inversion of prefetch over demand requests. The striper SCAN sorts queued prefetch requests.

TIP-1, the informed prefetching system, is integrated with the unified buffer cache. It uses a simple mechanism to manage resources: it allows a maximum of 512 file-cache buffers (about 1/3 of the cache) to hold hinted but still unread data. Whenever the number of such buffers is below the limit, TIP-1 prefetches according to the next available hint. Hinted blocks already in the cache are promoted to the tail of OSF/1's least-recently-used (LRU) list. When an application accesses a hinted block for the first time, TIP-1 reduces the count of unread buffers and resumes prefetching. Unread buffers may also age out of the cache, triggering further prefetching.

TIP-1 has been running since mid 1993 in the 4.3 BSD FFS in Mach 2.5. Soon thereafter, it was ported to the UX server in a Mach 3.0 system on a DECstation 5000/200. Equipped with four disks and a user-level striper, this system was able to, at its best, reduce the elapsed time of a seek-intensive data visualization by 70% [Patterson94]. During the summer of 1994 we ported TIP-1 to the Alpha testbed to exploit its greater CPU and disk performance.

The default OSF/1 file system implements two I/O optimizations that give it excellent sequential read performance. First, it coalesces up to eight contiguous blocks into a cluster which it requests from the disk in a single access. Second, it has an aggressive readahead mechanism that detects sequential accesses to a file. The longer the run of sequential accesses, the further ahead it prefetches up to a maximum of eight clusters of eight blocks each. For large sequential accesses, such as "cat 1GB_file > /dev/null," OSF/1 achieves 18.2 MB/s from 15 disks through our striper.

Because we are primarily interested in reducing the total execution time of our I/O-intensive application suite, elapsed time is our basic metric. However, our systems make no attempt to reduce application CPU demands, so we tend to be interested in an application's I/O stall time. If there is no I/O stall time left to recover, our mechanisms can show no more benefit. Our goal, then, is to exploit unused disk parallelism to convert applications from being I/O-bound to being CPU-bound. To that end, each application is run without competition on arrays of from 1 to 15 disks. For comparison, we also show the results of running the same applications without issuing hints. All reported measurements are the average of three independent tests. Each test was run on a system with a cold cache. Before each sequence of three runs, the file system was formatted (block size = fragment size = 8192, inter-block rotational delay = 0, maximum blocks per file per cylinder group = 10000, bytes per inodes = 32K, all other parameters default), and the run's data was copied into the file system.

3.2 Agrep

Agrep is a variant of grep written by Wu and Manber at the University of Arizona [Wu92]. It is a full-text pattern matching program that allows errors. Invoked in its simplest form, its opens the files specified on its command line one at a time, in argument order, and reads each sequentially.

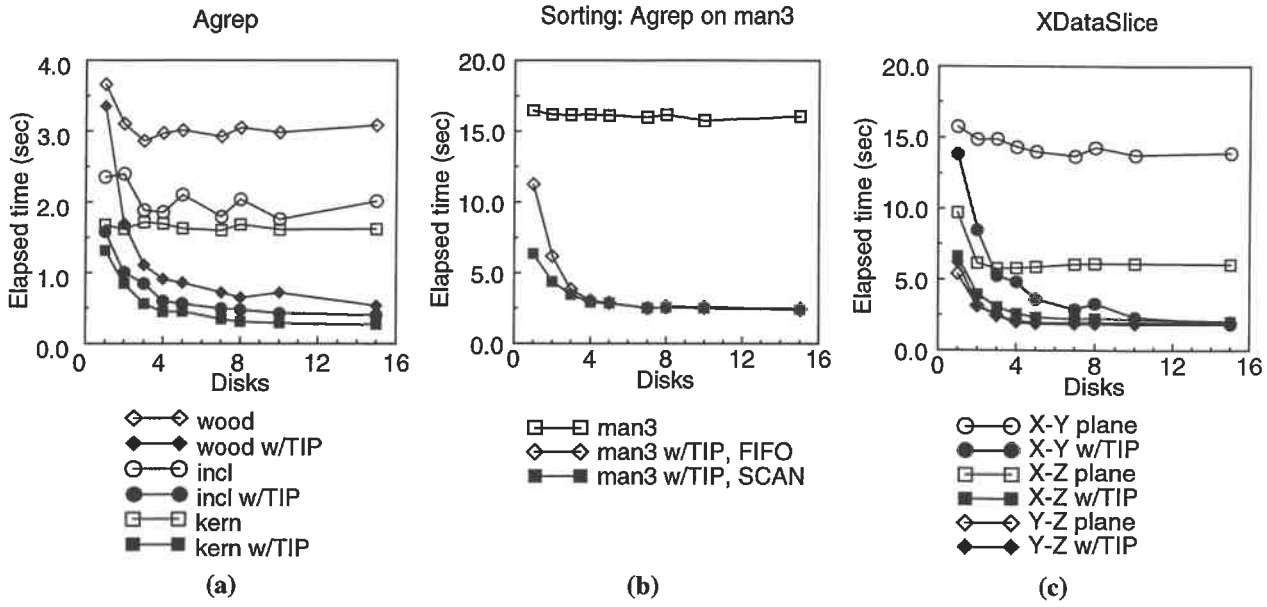

Figure 1. Elapsed time of search and visualization using informed prefetching. Figures (a) and (b) report the elapsed time for searches through files in four different directories. Without TIP, agrep is unable to exploit the parallelism of the disk array. Figure (b) shows the benefit of increasing the throughput of disk accesses by sorting the prefetch requests. The effect is especially pronounced when there is only one disk and disk bandwidth is at a premium. Figure (c) shows the elapsed time for rendering each of three orthogonal planes through a 512 MB dataset. The Y-Z plane is stored sequentially on disk and OSF/1 readahead performs well. In contrast, a seek is required to read every block in the X-Y plane. Without sequentiality, OSF/1 readahead is ineffective, but, informed by hints, TIP is able to prefetch in parallel and mask the latency of the many seeks.

Since the arguments to agrep completely determine the files that agrep will access, we can issue hints as soon as the application is invoked. Agrep simply loops through the argument list and informs the file system of the names and order of the files that will be sequentially read. When searching data collections such as software header files or mail messages, hints from agrep frequently specify hundreds of files too small to benefit from history-based readahead.

Our text-search benchmark comprises four different executions of agrep searching for a simple string that will not be found through all the files of a directory:

- 224 newsgroup messages on the topic of woodworking consuming 358 disk blocks (*wood*),
- 1352 unformatted OSF/1 manual pages consuming 1556 disk blocks (*man3*),
- 193 /usr/include header files consuming 260 disk blocks (*incl*), and
- 107 OSF/1 kernel source files consuming 183 disk blocks (*kern*).

Figures 1(a) and (b) report the elapsed time of these four searches. Note that without TIP-1, agrep is largely unable to exploit the disk array's parallelism because there is little parallelism in its I/O workload. The files are searched serially and most of them are small, so even OSF/1's readahead does not achieve parallel transfer. However, agrep's disclosure of future accesses exposes I/O concurrency. Informed prefetching leverages these hints to exploit disk parallelism and reduces the elapsed time of agrep searches by 80% to 85%. On our testbed, arrays of as few as four disks are sufficient to achieve most of this benefit. As the gap between CPU and I/O performance grows, however, we expect informed prefetching to take advantage of larger arrays for even greater benefits.

This application also demonstrates the second of the three benefits of disclosing hints: increased disk throughput through better disk scheduling. Figure 1(b) shows the effect of striper scheduling most dramatically with one disk. In this figure, the "SCAN" curve corresponds to the SCAN sorting of prefetches in the striper plus sorting of up to five prefetches at the disk as described in Section 3.1 and as used to obtain the results in Figure 1(a). The "FIFO" curve shows the elapsed time of the search when the striper does not reorder prefetches before forwarding them to the disk. The figure shows that sorting the first five accesses and overlapping I/O and computation reduce elapsed time by 31%. Sorting all prefetches reduces elapsed time by 63%.

The benefit of sorting in the striper decreases with larger arrays. To some extent, this is because the fixed number of prefetches is spread over more disks, so the individual disk queues are not as deep and there is less opportunity for scheduling optimization. But, the primary reason is that, as more disks are added to the array, there is more raw disk bandwidth available. As the performance bottleneck shifts from the disks to the CPU, optimizing disk performance becomes less important.

3.3 XDataSlice

XDataSlice (XDS) is an interactive scientific visualization tool developed at the National Center for Supercomputer Applications at the University of Illinois [NCSA89]. Among other features, XDS lets scientists view arbitrary planar slices through their 3-dimensional data with a false color mapping. The datasets may originate from a broad range of applications such as airflow simulations, pollution modelling, or magnetic resonance imaging and tend to be very large.

It is often assumed that because disks are so slow, good performance is only possible when data is in main memory. Thus, many applications, including XDS, require that the entire dataset reside in memory. Because memory is still expensive, the amount available often constrains scientists who would like to work with higher resolution and therefore larger datasets. Informed prefetching invalidates the slow-disk assumption and makes practical out-of-core computing, even for interactive applications. To demonstrate this, we added an out-of-core capability to XDS.

To render a slice through an in-core dataset, XDS iteratively determines which data point maps to the next pixel, reads the datum from memory, applies false coloring, and writes the pixel in the output pixel array. In contrast, to render a slice from an out-of-core dataset, XDS splits this loop in two. Both to manage its internal cache and to generate hints, XDS first maps all of the pixels to data-point coordinates and stores the mappings in an array. Having determined which data blocks will be needed to render the current slice, XDS flushes unneeded blocks from its cache, gives hints to TIP and reads the needed blocks from disk. In the second half of the split loop, XDS reads the cached pixel mappings, reads the corresponding data from the cached blocks, and applies the false coloring.

In general, generating hints may require a little extra work to precompute the file accesses. Agrep does a quick loop through its argument list. XDS splits a loop to precompute which blocks will be needed. The cost of this extra work is often negligible, as in the case of agrep. Often, as in XDS, the net cost of the work may be minimized by caching the result of the precomputation for future use. In some cases, as we shall see for Postgres, restructuring the code to precompute the needed blocks itself improves performance even when no hints are given.

Our test dataset consists of 512^3 32-bit floating point values requiring 512 MB of disk storage. The dataset is organized into 8 KB blocks of 16x16x8 data points each. The blocks are stored on the disk in Z-major order. Our test renders one random slice in each of the three orthogonal planes: X-Y, X-Z, and Y-Z. Figure 1(c) reports elapsed time when rendering these slices both with and without informed prefetching.

Without TIP, OSF/1 readahead is effective for the Y-Z plane which is stored sequentially. But, as soon as the requested slice deviates from this plane, sequentiality is lost and performance plummets. Even

in the X-Z plane in which whole rows are stored sequentially, OSF/1 is only able to take advantage of two disks in the array. In the X-Y plane, every block requires a seek, and performance is miserable, requiring fifteen seconds to render a slice. This is unacceptable for an interactive application.

With informed prefetching, however, all slice orientations are rendered in the same elapsed time, about two seconds. TIP prefetches in parallel to mask the cost of a seek for every block in the X-Y plane. The sum of the elapsed time of slices in all three orientations is reduced by up to a factor of 6. With informed prefetching, XDS performance is acceptable even when working with huge out-of-core datasets.

Unlike agrep, disk batching and scheduling have little effect in XDS because the components of each slice are requested in ascending order of offset within the file, which is a good order to fetch from the disk.

3.4 Postgres

Postgres, version 4.2, [Stonebraker86, Stonebraker90] is an extensible, object-oriented relational database system from UC Berkeley. In our test, Postgres executes a join of two relations. The outer relation contains 20,000 unindexed tuples (3.2 MB) while the inner relation has 200,000 tuples (32 MB) and is indexed (5 MB). On average, one in five tuples in the outer relation finds a match in the inner relation, so the output is about 4000 tuples written sequentially.

To perform the join, Postgres reads the outer relation sequentially. For each outer tuple, Postgres checks the inner relation's index for a matching inner tuple and, if there is one, reads the tuple from the inner relation. From the perspective of storage, the inner relation and its index are accessed randomly. Most of the inner relation reads incur the full latency of a disk access because they are random, which defeats readahead, and they have very poor locality, which defeats caching. To disclose these inner-relation accesses, we employ a loop-splitting technique similar to that used in XDS. In the precomputation phase, Postgres reads the outer relation (disclosing the sequential access), looks up each outer-relation tuple address in the index (unhinted), and stores the addresses in an array. Postgres discloses these precomputed block address to TIP. In the second pass, Postgres rereads the outer relation but skips the index lookup and directly reads the outer-relation tuple whose address is stored in the array.

Figure 2(a) shows the elapsed time required for the join under three conditions: standard Postgres, Postgres with the precomputation loop but without giving hints, and Postgres with both the precomputation loop and giving hints. Simply splitting the loop reduces elapsed time by about 20%. When the loop is split, the buffer cache does a much better job of caching the index since there are no inner-relation data blocks polluting the cache. Even though Postgres reads the outer relation twice, there are about 900 fewer total disk I/Os in the precomputation run. In this case, precomputing disk accesses actually saves time. Delivering the hints to TIP reduces elapsed time by an additional 45% for a total reduction over standard Postgres of up to 55%.

3.5 Sphinx

Sphinx [Lee90] is a high-quality, speaker-independent, continuous speech-recognition system. In our experiments, Sphinx is recognizing an 18-second recording commonly used in Sphinx regression testing.

Sphinx represents acoustics with Hidden Markov Models and uses a Viterbi beam search to prune unpromising word combinations from these models. To achieve higher accuracy, Sphinx uses a language model to effect a second level of pruning. The language model is a table of the conditional probability of word pairs and word triples. At the end of each 10 ms acoustical frame, the second-level pruning is presented with the words likely to have ended in that frame. For each of these potential words, the probability of it being recognized is conditioned by the probability of it occurring in a triple with the two most recently recognized words, or occurring in a pair with the most recently recognized word when there is no entry in

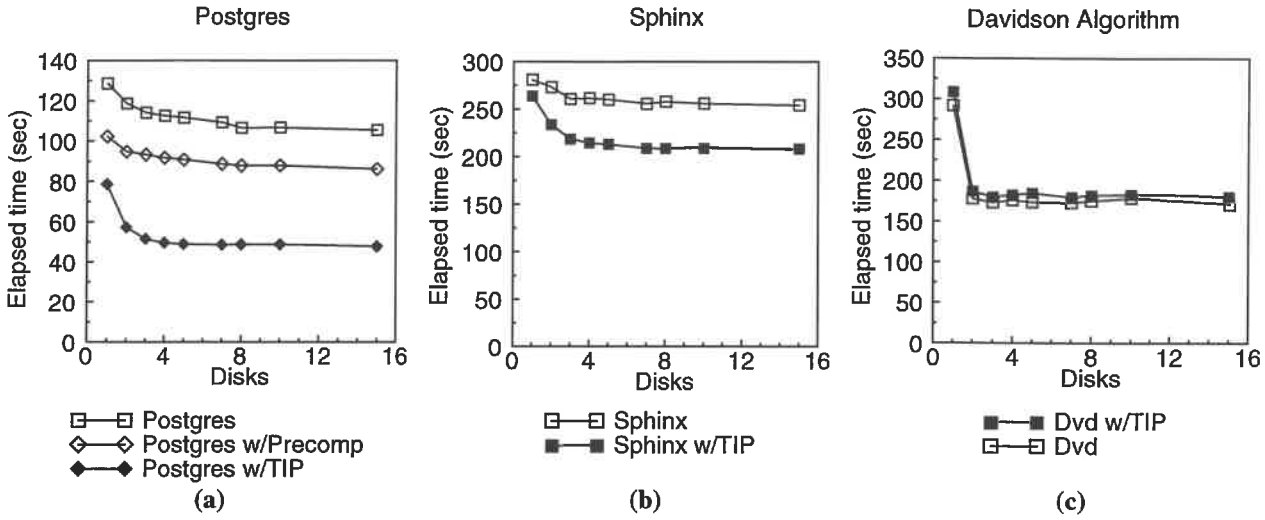

Figure 2. Elapsed time of Postgres, Sphinx, and Davidson using informed prefetching. Figure (a) shows the elapsed time for join with the standard Postgres relational database, Postgres restructured to precompute offsets in the inner relation, and the restructured Postgres when it gives hints. The restructuring improves accesses locality and therefore cache performance and runs faster than standard Postgres. Delivering hints further improves performance. Figure (b) shows the benefits of informed prefetching for the Sphinx speech-recognition program. Figure (c) shows the performance of the Davidson algorithm applied to a computational-chemistry problem. The algorithm repeatedly reads a large file sequentially. OSF/1's aggressive readahead algorithm performs slightly better than informed prefetching for this access pattern because it incurs a little less overhead.

the language model for the current triple. To further improve accuracy, Sphinx makes three similar passes through the search data structure, each time restricting the language model based on the results of the previous pass.

Sphinx, like XDS, came to us as an in-core only system. Since it was commonly used with a dictionary containing 60,000 words, the language model was several hundred megabytes in size. With the addition of its internal caches and search data structures, virtual-memory paging occurs even on a machine with 512 MB of memory. We modified Sphinx to fetch from disk the language model's word pairs and word triples as needed. This enables Sphinx to run on our 128 MB test machine 90% as fast as on a 512 MB machine.

Additionally, we modified Sphinx to disclose the word pairs and word triples that will be needed to evaluate each of the potential words offered at the end of each frame. Because the language model is sparsely populated, at the end of each frame there are about 100 byte ranges that must be consulted, of which all but a few are in Sphinx's internal cache. However, there is a high variance on the number of pairs and triples consulted and fetched, so storage parallelism is often employed.

Figure 2(b) shows the elapsed time of Sphinx recognizing the 18-second regression test with and without hints. As with the previous applications, Sphinx derives only a small benefit from the disk array. With informed prefetching, the elapsed time of this Sphinx test is reduced by up to 20%.

3.6 MCHF Davidson algorithm

The Multi-Configuration Hartree-Fock, MCHF, is a suite of computational-chemistry programs used for atomic-structures calculations which we obtained from Vanderbilt University. The Davidson algorithm [Stathopoulos94] is an element of the suite that computes, by successive refinement, the extreme

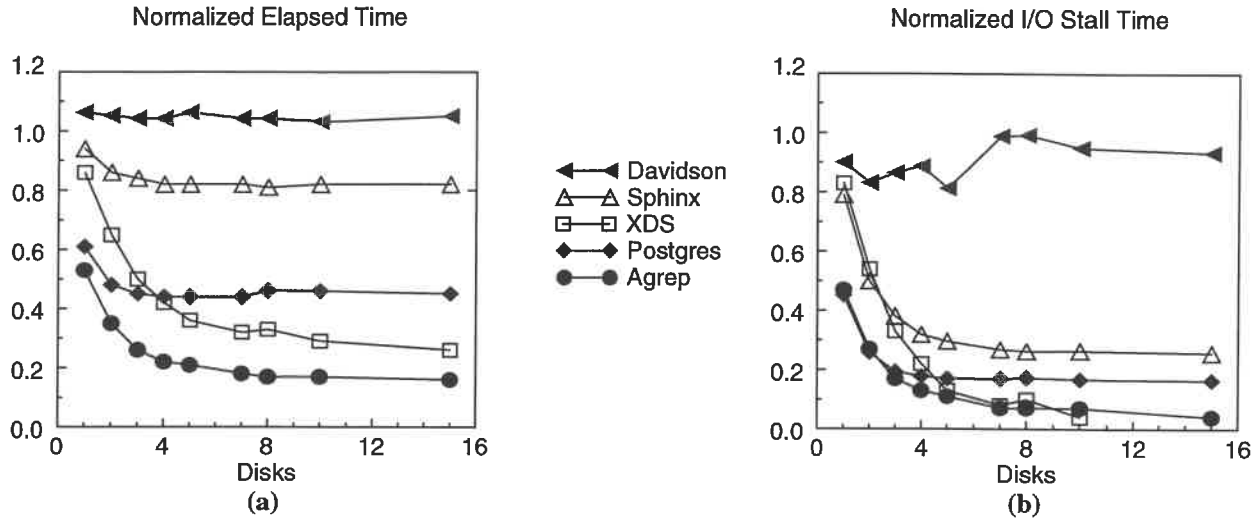

Figure 3. Normalized elapsed and I/O stall times of the five test applications. Figure (a) shows elapsed time with hints as a fraction of elapsed time without hints for each of the five applications. Informed prefetching does not improve on OSF/1 readahead for the large sequential accesses in Davidson but does show significant benefits for the other applications. The goal of informed prefetching is to reduce I/O latency. Figure (b) shows I/O stall time with hints as a fraction of stall time without hints. With the exception of Davidson, informed prefetching uses the parallelism of disk arrays to eliminate 75% to 95% of I/O stall time. All of these I/O-intensive applications become CPU-bound when informed prefetching and disk arrays combine to improve I/O performance.

eigenvalue-eigenvector pairs of a large, sparse, real, symmetric matrix stored on disk. In our test, which uses the lowest 4800 energy states of a sodium atom as a basis, the size of this large matrix is 17 MB.

At its core, the Davidson algorithm improves its estimate of the extreme eigenpairs by computing the extreme eigenpairs of a much smaller, derived matrix. Each iteration of this core computes a new derived matrix by a matrix-vector multiplication involving the large, on-disk matrix. Thus, the algorithm repeatedly accesses the same large file sequentially. Annotating this code to give hints was straight forward. At the start of each iteration, the Davidson algorithm discloses the whole-file, sequential read anticipated in the next iteration.

Figure 2(c) reports the elapsed time of the entire computation with and without informed prefetching. Because these hints disclose only sequential access in one large file, OSF/1's aggressive readahead matches the performance of informed prefetching and in fact performs slightly better because it incurs less overhead. With or without hints, this Davidson test benefits significantly from the extra bandwidth of a second disk but then becomes CPU-bound.

Neither OSF/1 nor informed prefetching alone uses the 12 MB of cache buffers well. Because the 17 MB matrix does not fit in the cache, the least-recently-used (LRU) replacement algorithm flushes all of the blocks before any of them are reused. As we will see in the following sections, however, informed caching is able to use the hint of repeated access to modify cache behavior and significantly reduce the elapsed time of the Davidson algorithm.

3.7 Summary of informed prefetching experiments

Figure 3(a) shows, for each of the five test applications, the elapsed time when they give hints as a fraction of the elapsed time when they do not. For agrep and XDS, which have multiple parts, we show the sum of the times for the individual parts. From this figure we see that

- Davidson does not benefit from informed prefetching,
- Sphinx elapsed time is reduced by up to 20%,
- Postgres's join elapsed time is reduced by up to 55%,
- XDataSlice's elapsed time for the sum of three slices is reduced by up to 75%, and
- Agrep's elapsed time for the sum of four searches is reduced by up to 84%.

Informed prefetching reduces I/O latency to achieve these results. Figure 3(b) illustrates this by showing the time each application spends stalled on I/O with informed prefetching as a fraction of the stall time without. With the exception of Davidson, which is well served by OSF/1 readahead, informed prefetching eliminates between 75% and 95% of I/O stall time. Further, informed prefetching leverages the parallelism of disk arrays to eliminate much more stall time than is possible by simply overlapping computation with the activity of a single disk. Still, even a high-performance workstation such as our Alpha testbed needs only a modest array of as few as four disks to eliminate most of the I/O stall time of these I/O intensive applications. As processor performance improves, however, we expect informed prefetching to take advantage of greater disk parallelism and the benefits of informed prefetching will increase.

In this section, we have demonstrated that informed prefetching can deliver two of the three benefits we ascribe to the disclosure of application knowledge of future I/O accesses. TIP-1 and our striper expose I/O parallelism and exploit it with parallel prefetching from disk arrays and with improved disk scheduling. However, as the performance of the Davidson algorithm shows, TIP-1 is unable to exploit disclosure to deliver the third benefit, informed caching. Thus, we now turn our attention to informed caching.

4 Informed caching by cost-benefit analysis

4.1 Approach

The goal of the informed cache manager is to allocate cache buffers to minimize application elapsed time. Its basic approach is to estimate the impact on execution time of alternative buffer allocations and then choose the best allocation. As shown in Figure 4, the manager has to choose from among three broad uses for each buffer: caching recently used data in the traditional LRU queue; prefetching data according to hints (or some other predictor of future accesses); and caching data that a predictor indicates will be reused in the future. In practice, it would be extremely difficult to estimate the performance of allocations at a global level both because of the huge number of possible allocations and because of the complex interactions among the alternative uses of cache buffers.

To avoid these complexities, we have developed a framework for global allocation that depends on local estimates of the impact on execution time and on a basis for comparing these local estimates. Within this framework, allocation decisions are made on a buffer-by-buffer basis as reallocation opportunities arise. Thus, our informed cache manager repeatedly faces two decisions:

- Which block should be replaced when a buffer is needed for prefetching or to service a demand request?
- Should a cache buffer be used to prefetch data now?

The manager uses a three-step process to make these decisions. First, it independently estimates the impact on execution time of taking an LRU cache buffer, or prefetching a block, or flushing a hinted block. Then, it converts these estimates to the common basis for comparison: the amount of time saved or lost per buffer per access. Finally, it allocates buffers to obtain the biggest bang per buffer.

In describing our framework for informed caching, we start by describing our model of system performance from which we derive our estimates of the impact of using a cache buffer in different ways.

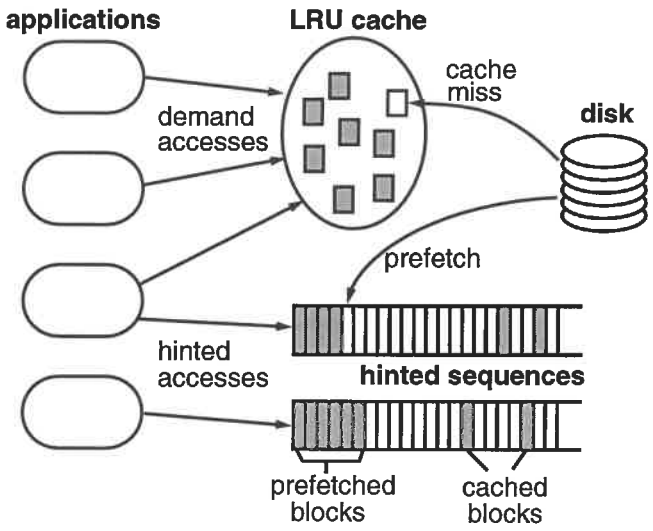

Figure 4. Three uses of cache buffers. Cache buffers may hold recently used data for unhinted demand accesses in the LRU queue. Or, they may hold data that hints indicate will be reaccessed soon. Finally, they are used to prefetch data according to hints. The job of the cache manager is to allocate buffers for these different uses to maximize cache hits while initiating prefetches early enough to minimize the time spent waiting for data.

4.2 System model

The execution time, T, for an application is given by,

$$T = N_{I/O}\left(\overline{T_{CPU}} + \overline{T_{I/O}}\right) \tag{1}$$

where $N_{I/O}$ is the number of I/O requests, $\overline{T_{CPU}}$ is the average application CPU time between requests, and $\overline{T_{I/O}}$ is the average time it takes to service an I/O request.

The time to service a request depends on whether or not the requested data is in the cache. If it is, T_{hit} is the time it takes to read a block from the cache. If the block is not in the cache, it needs to be fetched from the disk before it may be delivered to the application. In addition to the latency of the fetch, T_{disk}, these requests suffer the computational overhead, T_{driver}, of allocating a buffer, queuing the request at the drive, and servicing the interrupt when the I/O completes. The time to service an I/O request that misses in the cache, T_{miss}, is the sum of these times,

$$T_{miss} = T_{hit} + T_{driver} + T_{disk} \tag{2}$$

Prefetching has the benefit of reducing execution time by masking some or all of the disk latency of a miss. But, as shown in the next section, stealing a buffer from the LRU cache makes it more likely that an unhinted request misses in the cache and must pay a delay of T_{miss} instead of T_{hit}. Holding a hinted block in the cache saves the costs involved in prefetching that block back into the cache later. But, holding on to the hinted buffer deprives the LRU cache of a buffer which it could use to increase the number of hits.

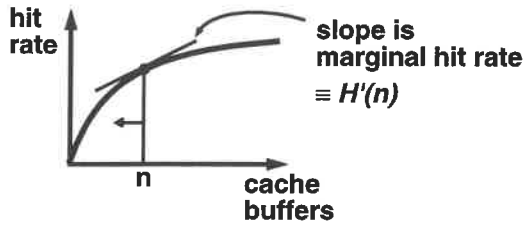

Figure 5. Impact of stealing a buffer from the LRU cache. When a buffer is taken from the LRU cache, the hit rate drops in proportion to the marginal hit rate. This increases, on average, the number of cache misses and therefore the average time required to service an I/O request. This increase is the cost of reallocating a buffer for prefetching.

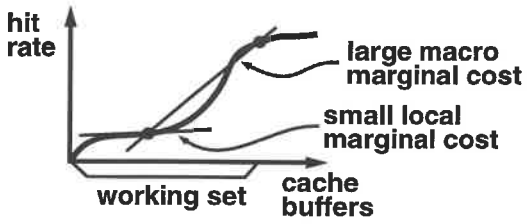

Figure 6. Avoiding local minima in LRU cost estimate. Because *H'(n)* is a point estimate of the cost of losing a buffer, it is possible to lose sight of large-scale effects such as occur when the working set of an application fits in the cache. To avoid this, one should use a broader estimate of the cost by looking past points of inflection in *H(n)* and use the long-term average marginal cost instead.

4.3 The cost of deallocating an LRU buffer

Over time, the portion of requests that hit in the cache is given by the cache-hit ratio. The cache-hit ratio, *H(n)*, is a function of *n*, the number of buffers in the cache and the I/O workload. Given *H(n)*, $\overline{T_{I/O}}$ for demand accesses is

$$\overline{T_{I/O}} = H(n)\,T_{hit} + (1 - H(n))\,T_{miss}. \tag{3}$$

Figure 5 shows that the cost of taking a buffer from the LRU list for prefetching is a decrease in the cache-hit ratio. This is always true because LRU replacement is a stack algorithm. The decrease in the cache-hit ratio results in an increase in the average access time:

$$\Delta\overline{T_{I/O}} \approx H'(n)\,(T_{hit} - T_{miss}), \tag{4}$$

where *H'(n)* is the derivative with respect to *n*. Our informed cache manager directly and continually estimates *H'(n)* as described in Section 5.2 and computes the expected impact of losing an LRU buffer.

One drawback of this approach is that it is a point estimate. In general, *H(n)* may not be a smooth function as suggested by Figure 5. Instead, the function may be more like that in Figure 6. There is often a large jump in the hit rate when the entire working set of an application fits in the buffer cache. In cases such as these, it is better to use the average marginal cost over the region that includes the jump in the hit rate.

4.4 The benefit of prefetching

The benefit of prefetching a block is the avoidance of some or all of the disk component of the cost of a miss. In contrast to LRU buffers, whose usefulness is averaged over many accesses, the benefit of prefetching is a one-time reduction in the cost of the first access to a block. Note that repeated accesses to a block do not make it any more desirable to prefetch that block. Prefetched or not, after the first access, the block will be available in the cache. Note also that prefetching does not avoid the CPU overhead of a disk access, T_{driver}. Thus, T_{disk} is the upper bound on the benefit of prefetching a block.

For most blocks, the benefit of starting a prefetch now is less than T_{disk}. This is because the informed cache manager will have the opportunity to reconsider its prefetch decision soon (such as at the

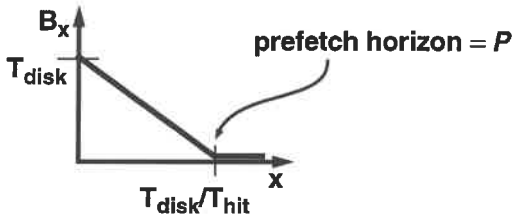

Figure 7. The prefetch horizon. Data consumption is limited, at a minimum, by T_{hit}, the time it takes to read data from the cache. Assuming adequate I/O bandwidth, and therefore no disk queues, there is no benefit from prefetching further ahead than the prefetch horizon, $P = T_{disk}/T_{hit}$. In general, T_{CPU} and T_{driver} further limit the data consumption rate.

next file-access event). If the prefetch can be delayed and still complete before it is needed, we consider there to be no benefit from starting the prefetch now.

The key observation is that the application's data consumption rate is bounded. At a minimum, T_{hit} limits data consumption. In general, consumption is further limited by T_{CPU}, the application's computation between I/O requests. In steady state, when data is streaming in while the application processes the data as fast as it can, T_{driver} further reduces the data consumption rate. Based on this, we could estimate the time until an application consumes data x requests in the future as $x(\overline{T_{CPU}} + T_{hit} + T_{driver})$ and the benefit of prefetching the block that will be needed then as,

$$B_x = T_{disk} - x\left(\overline{T_{CPU}} + T_{hit} + T_{driver}\right). \tag{5}$$

However, to avoid consistently failing to have completed prefetches during bursts of data consumption, we define the benefit of prefetching assuming that such a burst is about to occur. In particular, we assume T_{CPU} and T_{driver} are about to go to zero. In this case, as shown in Figure 7, the benefit of prefetching a block that will not be used for x requests in the future, B_x, is at most

$$B_x = T_{disk} - xT_{hit}. \tag{6}$$

An important consequence of this relation is that, assuming adequate I/O bandwidth, there is no benefit from prefetching further than

$$P = T_{disk}/T_{hit} \tag{7}$$

requests into the future. We call P the *prefetch horizon*. Prefetching any further ahead than the prefetch horizon wastes buffers that could be used to cache data.

4.5 Comparing LRU cost to prefetching benefit

We now have estimates of the cost of taking a buffer from the LRU cache, $\overline{\Delta T_{I/O}}$, and the benefit of prefetching for an access x requests in the future, B_x. Unfortunately, these estimates are not directly comparable. The cost to the LRU is an average cost per access, while the benefit of prefetching accrues in a lump after x requests.

We solve this problem by defining a basis for comparing these estimates. The shared resource we are allocating is cache buffers. The smallest denomination of this resource is the occupation of one buffer for one I/O-access period. Buffers are spent to reduce application execution time. Thus, the basis for comparison, the common currency in the cache manager, is the change in application execution time per access that a buffer is allocated.

Conversion of the two independent estimates to the common basis is straightforward. The LRU cache's $H'(n)$ is effectively the number of hits per access for the least-recently-used buffer. Thus, the cost of losing an LRU cache buffer is already computed in terms of time per access per buffer, so no conversion is necessary. To convert B_x, we average the benefit of prefetching over the x requests that the buffer will be occupied. Conceptually, the benefit is greater than the cost if tying up a buffer for x accesses is unlikely to cause an additional miss in the LRU cache. But, the mechanism of a common basis for comparison relieves the informed cache manager of the need to understand this combined effect. Instead, cost and benefit estimates are made independently and then compared in terms of the common basis.

One last factor we must take into account is the difference in the rates of unhinted demand accesses, r_d, and hinted accesses, r_h. To compensate for this difference, we normalize the estimates by these rates. We can now say that a buffer should be reallocated from the LRU cache for prefetching when

$$r_h \frac{B_x}{x} > r_d H'(n) \, (T_{miss} - T_{hit}) \quad . \tag{8}$$

4.6 The cost of flushing a hinted block

Though there is no benefit from prefetching beyond the prefetch horizon, the same is not true for caching. In this section, we compute the cost of flushing from the cache a block that a hint indicates will be reused.

If we flush a cached block for which there is a hint y accesses in the future and later prefetch it back into the cache, we add to execution time the CPU cost of accessing the disk, T_{driver}. On the other hand, flushing the block frees its buffer for use elsewhere. If we assume that the block will be prefetched back into the cache at the prefetch horizon, P, then flushing the block frees the buffer for $y - P$ accesses. In terms of the common basis, the cost of flushing the block is $T_{driver}/(y - P)$.

This estimate is faulty if the flush candidate is within the prefetch horizon, $y \le P$. Here it may be impossible to prefetch the block back before it is needed. In this case, the cost of flushing is the access cost, T_{driver}, plus the non-overlapped disk access time that will have to be paid if the block is flushed, B_y. Furthermore, because y is within the prefetch horizon, the prefetch of the block should be imminent. Thus, it is not safe to assume that flushing such a block would free a buffer for any more than one access.

Summarizing, the change in execution time expected as a result of flushing a block and prefetching it back later in terms of the common basis (that is, averaged over the number of accesses the buffer is freed) is

$$\Delta\overline{T_{flush}} = \begin{pmatrix} y > P & \dfrac{T_{driver}}{y - P} \\ y \le P & \dfrac{T_{driver} + B_y}{1} \end{pmatrix} . \tag{9}$$

Since this expression is in terms of the common basis, it allows comparison to the value of the LRU cache's least valuable block. It also allows comparison to the benefit of a prefetch when a buffer is needed for that purpose. In this latter case, note that the cost of flushing a hinted block is always greater than the benefit of prefetching that same block as given by equation (6). Thus, there is hysteresis in these estimates that reduces the likelihood of thrashing. A block that is flushed is not likely to be prefetched back soon and vice versa.

4.7 Putting it all together: global min-max

We now have estimates of the value or cost of each action that the informed cache manager might choose:

- the expected cost of stealing the LRU cache's least valuable block,
- the expected benefit of prefetching any hinted but uncached block, and
- the expected cost of flushing any hinted cache block.

We use these to resolve the informed cache manager's questions:

1. Which block should be replaced when a buffer is needed for prefetching or to service a demand request?
   - The globally least valuable block in the cache.
2. Should a cache buffer be used to prefetch data now?
   - Prefetch if the expected benefit is greater than the expected cost of flushing or stealing the least valuable block.

To identify the globally least valuable block in the cache, the informed cache manager maintains a separate estimator for the LRU cache and for each independent stream of hints. Each estimator determines the costs of flushing or stealing the blocks for which it has information. It is possible for a single block to be tracked by two or more estimators. For example, a hint could refer to a block that already resides in the LRU queue. In this case, each of the estimators independently estimates the cost of flushing that block.

For the LRU cache, the value of a block depends on its order in the LRU queue. If a block has position i in this queue, then its value is estimated as $H'(i)\ (T_{miss} - T_{hit})$.

Hint estimators rank their blocks according to the cost of flushing them as determined by equation (9). When the cost of access, T_{disk}, is the same for all blocks tracked by an estimator, the least valuable block is simply the one whose next reference is furthest in the future.

The values determined by each estimator are normalized according to the rate of accesses to the estimator as shown in equation (8). If several estimators have determined values for a single block, the global value of the block is the maximum of the normalized values of the independent estimators. For example, a block may be very far down in the LRU queue, but a hint says that it will be accessed soon, so assign to the block the maximum of its valuations.

The globally least valuable block is the block whose maximum valuation is minimal over all blocks. Hence, our replacement policy employs a global min-max valuation of blocks. The overhead of all this estimation seems prohibitive. In practice, as will be described in Section 5.1, the value of only a small number of blocks needs to be determined to find the globally least valuable block.

Note that the informed cache manager does not choose from a restricted set of blocks. An LRU buffer may be used to prefetch a block. Later, a cached hinted block may be flushed to service a demand miss. In this sense, buffers do not belong to individual estimators. Note also that buffer allocation is done one buffer at a time. That is, there is no high-level decision that this estimator should have N blocks.

4.8 Informed Caching Example: MRU

As an aid to understanding how informed caching 'discovers' good caching policy, we show how it exhibits most-recently-used (MRU) behavior for repeated access sequences. Figure 8 illustrates our example.

At the start of the first iteration through a sequence that repeats every N accesses, the cache manager prefetches up to the prefetch horizon. After the first block is consumed, it becomes a candidate for replacement either for further prefetching or to service demand misses. However, if the hit-rate function, $H(n)$, indicates that the least recently used blocks in the LRU queue don't get many hits, then these blocks

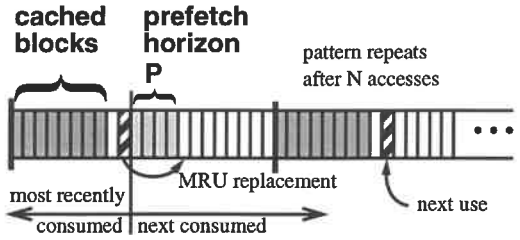

Figure 8. MRU behavior of the informed cache manager on repeated access sequences. The number of blocks allocated to caching for a repeated access pattern grows until the caching benefit is not sufficient to hold an additional buffer for the N accesses before it is reused. At that point, the least valuable buffer is the one just consumed because its next access is furthest in the future. This block is recycled to prefetch the next block within the prefetch horizon. A wave of prefetching, consumption, and recycling moves through the accesses until it joins up with the blocks still cached from the last iteration through the data.

will be less valuable than the hinted block just consumed. Prefetching continues, replacing blocks from the LRU list and leaving the hinted blocks in the cache after consumption.

As this process continues, more and more blocks are devoted to caching for the repeated sequence and the number of LRU buffers shrinks. For most common hit-rate functions, the fewer the buffers in the LRU cache, the more valuable they are. Eventually, the value of the LRU buffer exceeds the benefit of retaining the most recently consumed hinted block until its reuse N accesses in the future. At the next prefetch, this MRU block is flushed because, among the cached blocks with outstanding hints, its next use is furthest in the future.

At this point, a wave of prefetching, consumption, and flushing moves through the remaining blocks of the first interaction. Because the prefetch horizon limits prefetching, there are never more than P buffers in this wave. Even if a disk array delivers blocks faster than the application consumes them, there is no risk that the cache manager will use the cached blocks to prefetch further into the future. Thus, the MRU behavior of the cache manager is assured. Further, the cache manger strikes a balance in the number of buffers used for prefetching, caching hinted blocks, and LRU caching.

The informed cache manager discovers MRU caching without being specifically coded to implement this policy. Thus, we can expect similar performance for arbitrary access sequences where some of the blocks may be reused. An appropriate number of blocks will cache hinted data and when flushing must occur, the block whose next access is farthest in the future will be flushed.

4.9 Generalizing to non-uniform device performance

It is commonly suggested that blocks should be prefetched in the order they will be accessed [Cao95]. This is correct if all accessed devices take the same time to complete a fetch. However, in the general case, some data may be on a local disk and some data may be on the far side of a wide-area network. For the remote blocks, $T_{network} + T_{server} + T_{disk}$ should be substituted for T_{disk} when determining the benefit of prefetching and the prefetch horizon. This will cause the benefit of prefetching later remote blocks to exceed that of prefetching earlier local blocks. The informed-caching framework described in this section can adapt to these differences and make the correct trade-off. There is a similar effect for the order of flushing cached blocks with outstanding hints.

5 Implementation of informed caching and prefetching

5.1 Cache structure

The TIP cache manager replaces the unified buffer cache in OSF/1. It is organized around the concept of an estimator which determines the value of data blocks for servicing a stream of related accesses. From start up, the LRU estimator handles blocks for the stream of unhinted demand accesses. Every process that issues hints has an estimator for its stream of hinted access. Unhinted accesses from hinting processes belong to the global stream of unhinted demand accesses.

Blocks are replaced according to the global min-max valuation of blocks described in Section 4.7. At first blush, it appears that to find the globally least valuable block, each estimator needs to consider the value of every block in the cache. This would be extremely expensive. In practice, it is sufficient that each estimator be able to identify its least valuable block and estimate its value. From the LRU estimator's perspective, the least recently used block is the least valuable. For a hint estimator, assuming the same access time to all the blocks, the least valuable block is the one that will be accessed farthest in the future.

Each estimator converts its value estimate to the common basis of time saved or lost per access and declares this converted estimate. A global arbitrator normalizes the estimates by the rate of requests going to each estimator and ranks the estimators by the normalized estimate. When the arbitrator needs to find the globally least valuable block, it asks the least valued estimator to pick its least valued block. Thence forth, that estimator treats the block as if it is no longer in the cache, and declares to the arbitrator the value of the least valuable of its remaining blocks. Thus, the estimators determine the least valuable of only a subset of all the blocks in the cache. If a block is in the subset, the estimator is said to be *tracking* the value of the block. An estimator may maintain an interest in other blocks, such as the one it just picked, but it guarantees that all of the blocks that it tracks are at least as valuable as the globally declared value of its least valuable block.

In most cases, there is only one estimator interested in any particular block. When that estimator picks a candidate block for replacement, it is the end of the story. However, any other interested estimators are queried to see if they want to track the block and save it from replacement. An estimator only agrees to track the block if, in its estimation, the block is more valuable than the lowest globally declared value. If an estimator saves the first candidate from replacement, the arbitrator repeats the process, asking the least valued estimator to pick a new candidate block for replacement.

Estimators only pick for replacement blocks they are tracking. From the point of view of an estimator, an untracked block is not in the cache. Thus, all blocks must be tracked by at least one estimator, or they would remain in the cache forever. If no estimator considers a block valuable enough to track, then it is replaced. If the block cannot be replaced immediately, for example because it contains dirty data, then a special orphan estimator tracks the block till it is ready to be replaced.

5.2 Estimating the marginal LRU hit rate

Hint estimators directly apply equations (6) and (9) in Section 4 to estimate the values of the blocks they are tracking. The LRU estimator does the same. The only missing piece is how the hit-rate function $H(n)$ is determined for use in equation (4).

LRU block replacement is a stack algorithm and thus the ordering of blocks in the LRU queue is independent of the size of the cache. Thus, by observing where, in a queue of N buffers, cache hits occur, it is possible to make a history-based estimate of $H(n)$, the cache-hit rate as a function of the number of buffers, n, in the cache for any cache size less than N, $0<n<N$.

In the TIP informed cache manager, the number of buffers in the LRU queue may vary dynamically. Thus, it is just as important to know how the cache would perform with more buffers as with less.

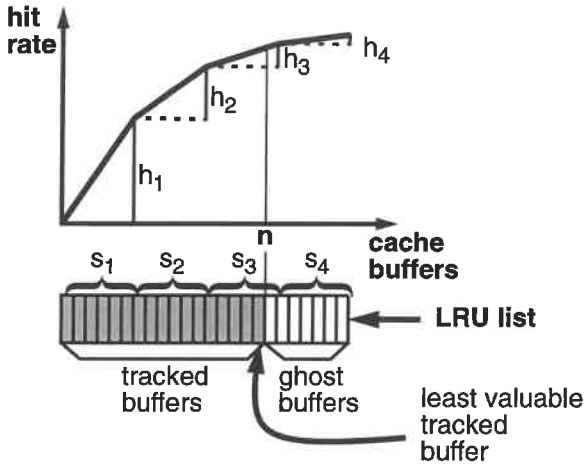

Figure 9. Piecewise estimation of the marginal hit rate for the LRU list. The list is broken into segments, s_1, s_2, s_3, ... Each buffer is tagged to indicate which segment it is in. The tag is updated when a buffer passes from one segment to the next. When there is a cache hit in segment i, the segment hint count, h_i, is incremented. The segment hit count divided by the number of buffers in the segment and normalized by the total number of accesses to the LRU cache is the piecewise approximation of the marginal hit rate for buffers in that segment.

The solution, borrowed from Maria Ebling [Ebling94], is to use ghost buffers, or buffer headers for which no data is allocated, to record when an access would have been a hit had there been more buffers in the cache. The LRU estimator tracks only the most recently used blocks. When it picks the head of the list for replacement, the buffer turns into a ghost but remains in the LRU list. Future requests for that block will find the ghost and record the fact that if the cache had been larger, the request would have been a hit. The length of the LRU queue, including ghosts, is limited to the total number of buffers in the cache.

Maintaining hit ratios separately for every size cache is expensive. Also, the hit ratio for a particular size, n, is probably less useful than a local average which has lower variance. For both of these reasons, TIP actually makes a piecewise estimate of the function $H(n)$ as shown in Figure 9. The LRU queue is segmented into contiguous disjoint regions and the hits to each segment are recorded. The marginal hit rate for cache sizes within that segment is then the number of hits in the segment divided by the size of the segment normalized by the total number of accesses.

5.3 The prefetcher

Each prefetching estimator passes the prefetcher an ordered list of blocks to prefetch and the current progress through the hints that the hinting application has made. Each block is marked with its position in the hint list so the prefetcher can tell how far into the future a given block will be used. The prefetcher computes the benefit of prefetching the most desired non-resident page and normalizes and sorts the estimators as above for the arbitrator. The prefetcher then compares the cost of losing the currently least valuable block in the cache to the benefit of prefetching. If the benefit exceeds the cost, the prefetcher acquires the buffer and initiates a prefetch. It then estimates the benefit of prefetching the next block and resorts the estimators. As new hints arrive or hinted data is consumed by the application, estimators may ask the prefetcher to update its estimate of the benefit of prefetching blocks for that estimator.

5.4 Clustering prefetches

OSF/1 derives significant performance benefits from clustering the transfer of up to eight blocks into one disk access. When should a buffer be allocated to prefetch a block as part of a cluster?

If the decision to prefetch a block has already been made, then the cost, T_{driver}, of performing an I/O will be paid. Any blocks that could piggyback on this I/O avoid most if not all of these costs. If these blocks are not clustered in now, prefetching them later will incur the full overhead of performing an I/O and possibly the cost of any unmasked disk latency. These are exactly the costs considered when deciding whether to flush a hinted block. Thus, the decision to include a block in a cluster is the same as the decision not to flush a hinted block.

When the prefetcher has decided to prefetch a block, it tries to build a cluster. It checks for contiguous hinted blocks not currently in the cache. It queries the appropriate estimator to see if it wants to track the block. If it does, then caching this block is more valuable than caching the globally least valuable block. The buffer is reallocated and the block is included in the cluster.

6 Informed caching experiments

Our implementation of informed caching in OSF/1 runs on the same testbed hardware as used in the evaluations of Section 3. We report its effect on the Davidson application which benefits neither from informed prefetching nor from a simple LRU cache manager. We look at this application over a range of array and cache sizes running without competition and over a range of cache sizes while competing for buffers with a more I/O-bound, non-disclosing application.

6.1 Discovering MRU

Recall from Section 3.6 that Davidson's out-of-core file is 17 MB in size and is read sequentially multiple times. OSF/1's aggressive readahead performs as well as informed prefetching for this simple access pattern. However, the static LRU replacement policy used by both OSF/1 and TIP-1 makes poor use of the 12 MB of cache. All blocks are flushed before they can be reused.

With informed caching, however, the replacement policy is dynamic and adaptive. Figure 10(a) shows the elapsed time of the Davidson benchmark when its data is striped over from 1 to 10 disks and the size of the informed cache is 12 MB. Enabled by disclosure, informed caching reduces the elapsed time of this benchmark by over 30% on one disk. When disk bandwidth is limited, improved caching avoids disk latency. On more disks, prefetching masks disk latency, but informed caching still reduces execution time more than 15% by avoiding the CPU overhead of extra disk accesses.

Figure 10(b) shows Davidson's elapsed time with and without hints as a function of cache size. Without hints, extra buffers are of no use until the entire 17 MB dataset fits in the cache. In contrast, the informed cache manager's min-max global valuation of blocks yields the smooth exploitation of additional cache buffers that you would expect from an MRU replacement policy. The notion of the prefetch horizon limits the use of buffers for prefetching, even when there is more than enough disk bandwidth to flush the cache with prefetched blocks. The informed cache manager effectively balances the allocation of cache buffers between prefetching and caching.

6.2 Balancing contention for cache buffers

Informed caching's cost-benefit model also balances the allocation of buffers among competing applications. To show this effect, we ran a highly I/O-bound, repetitive, non-hinting application ("cat 10MB_file ... 10MB_file > /dev/null") concurrently with a hinting run of Davidson. The non-hinting application makes poor use of cache buffers until the cache is big enough to hold its 10MB file. When the

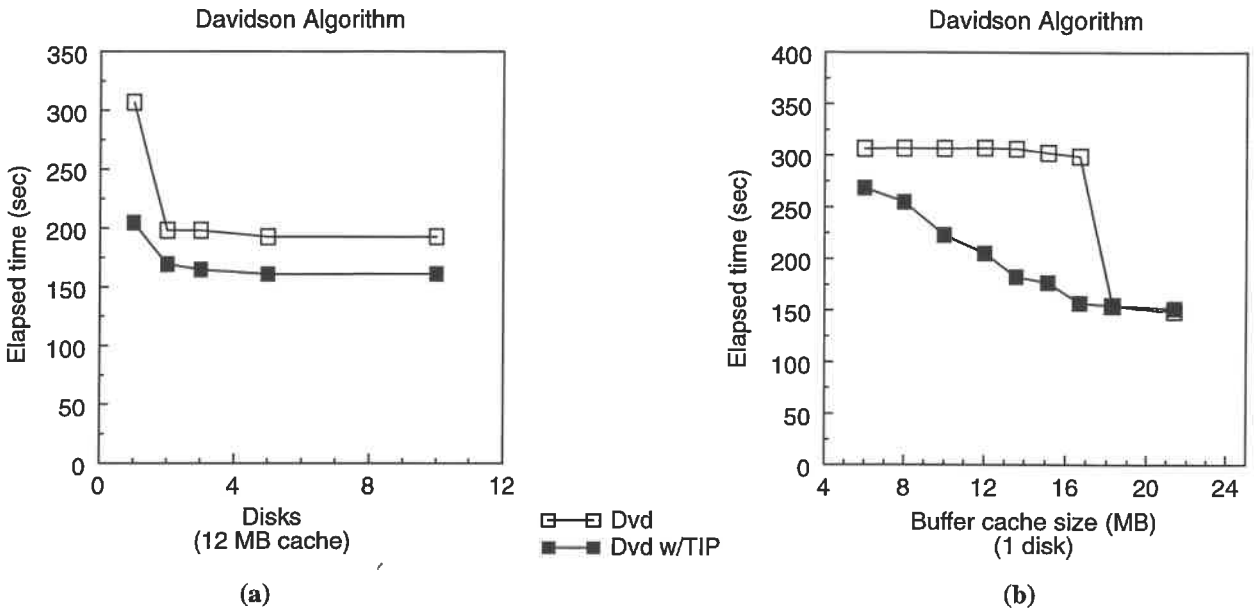

Figure 10. Benefit of informed caching for repeated accesses. Figure (a) shows that informed caching reduces elapsed time for Davidson's repeated accesses by more than 30% on one disk, where improved caching avoids disk latency. On more disks, prefetching masks disk latency, but informed caching still reduces execution time more than 15% by avoiding the overhead of going to disk. Figure (b) shows that informed caching discovers an MRU-like policy which uses additional buffers to increase cache hits and reduce execution time. In contrast, LRU caching derives no benefit from additional buffers until there are enough of them to cache the entire dataset.

file does fit in the cache, the application runs through its data at a very high rate because it does a negligible amount of computation per block.

Because elapsed time is difficult to interpret with two applications of different elapsed times, we selected the non-hinting application to complete first and we report the cache-miss ratio that the Davidson application has experienced up to that point. Figure 11 shows the results for cache sizes of 8 MB to 19 MB. For caches smaller than 12 MB, the informed cache manager finds no locality in the non-hinting application's use of the LRU cache. The marginal hit rate, $H(n)$, is effectively zero. The Davidson application's estimator does find value in caching its blocks. Thus, buffers migrate from the LRU queue to Davidson's estimator for MRU-style caching. As the cache increases in size from 8 MB to 12 MB, the additional buffers reduce Davidson's miss ratio. However, when the cache size reaches 12 MB, the non-hinting application can make good use of the LRU cache. Very few buffers migrate to Davidson's estimator and Davidson's miss ratio jumps to nearly 100%. Nevertheless, Davidson's execution time drops because the non-hinting application completes much more quickly and gets out of the way so Davidson can use all of the buffers to complete its execution. The non-hinting application's need for buffers is fixed, so as the cache grows beyond 12 MB, the additional buffers go to Davidson and the miss ratio drops. These results show that the informed cache manager effectively balances the use of cache buffers by hinting and non-hinting applications.

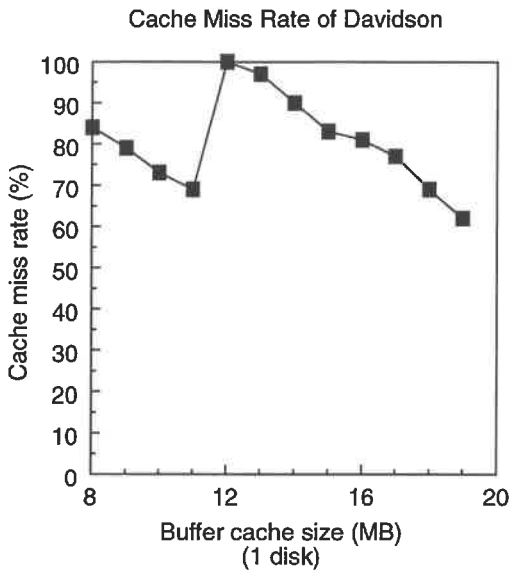

Figure 11. Competition for buffers between hinting and non-hinting applications. A background non-hinting task makes poor use of LRU cache buffers until, at 12MB, all its data fits in the cache. Thus, for caches smaller than 12MB, the informed cache manager shrinks the LRU cache and uses the buffers for MRU-style caching for the hinting Davidson application and the miss ratio drops. At 12 MB, the background task makes the best use of the buffers, so it gets the buffers, and Davidson's miss ratio jumps to nearly 100%. But, as the cache grows beyond 12 MB, Davidson gets the additional buffers and the miss ratio again declines.

7 Conclusion and future work

Traditional shallow readahead and LRU file caching no longer provide satisfactory resource management for the growing number of I/O-bound applications. Disk parallelism and cache buffers are squandered in the face of serial I/O workloads and large working sets. We advocate the disclosure of application knowledge of future accesses to enable informed prefetching and informed caching. Together, these resource managers can expose workload parallelism to exploit storage parallelism, apply disk scheduling to increase disk throughput, and adapt caching policies to the dynamic needs of running applications. The key to achieving these goals is to strike a balance between the desire to prefetch and the desire to cache.

We present a framework for informed caching based on a cost-benefit model of the value of a buffer. We show how to make independent local estimates of the value of caching a block in the LRU queue, prefetching a block, and caching a block for hinted reuse. We define a basis for comparing these estimates: the time gained or lost per buffer per I/O-access interval. And, we develop a global min-max algorithm to arbitrate among these estimates and maximize the global usefulness of every buffer.

Our results are taken from experiments with a suite of I/O-intensive applications executing on a DEC 3000/500 with an array of 15 disks. Our applications include text search, data visualization, database join, speech recognition, and computational chemistry. With the exception of computational chemistry, none of these applications exploits the parallelism of a disk array. Informed prefetching with at least four disks reduces the elapsed time of the remaining four applications by 20% to 85%. For the computational chemistry, which repeatedly reads a large file sequentially, OSF/1's aggressive readahead does as well as informed prefetching. However, informed caching's adaptive policy values this application's recently used blocks lower than older blocks and so 'discovers' an MRU-like policy that improves performance by up to

30%. Finally, our experimental results show that our resource-management framework effectively balances resources between prefetching and caching and between applications that disclose and those that do not.

Together, informed caching and informed prefetching provide a powerful resource management scheme that adapts to available storage bandwidth, adapts to an application's use of buffers, and optimizes disk throughput with scheduling and batching.

Although the results reported in this paper are taken from a running system, there remain many interesting related questions. In the area of hint generation, richer hint languages might significantly improve the ability of programmers to disclose future accesses. Even easier on the programmer would be the automatic generation of high quality hints and we have begun work on this problem.

There are a number of issues in the area of resource management. Currently, our hints are very accurate; appropriate strategies for dealing with imprecise, but still useful, hints are needed. Our cost-benefit model should easily adapt to non-uniform device bandwidths by replacing the average disk-access time with the appropriate value. But, we have not yet implemented this feature.

Perhaps the most exciting future work lies in exploiting the extensibility of our resource management framework. Because value estimates are made independently with local information, and then compared using a common currency, it should be possible to add new types of estimators. For example, a virtual-memory estimator could track VM pages, thereby integrating VM and buffer-cache management.

8 Acknowledgment

We wish to thank a number of people who contributed to this work including: Charlotte Fischer and the Atomic Structure Calculation Group in Department of Computer Science at Vanderbilt University for help with the Davidson algorithm, Ravi Mosur and the Sphinx group, here at CMU, Jiawen Su, who did the initial port of TIP to OSF/1 from Mach, David Golub for his debugging and coding contributions, Chris Demetriou, who wrote the striping driver, Alex Wetmore, who ported our version of XDataSlice to the Alpha, M. Satyanarayanan for early contribution of ideas and everyone in the PDL for their support and tolerance during the rush to complete this work.

9 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.

[Cao94] Cao, P., Felten, E.W., Li, K., "Implementation and Performance of Application-Controlled File Caching," *Proc. of the First USENIX Symp. on Operating Systems Design and Implementation*, Monterey, CA, November, 1994, pp.165-178.

[Cao95] Cao, P., Felten, E.W., Karlin, A., Li, K., "A Study of Integrated Prefetching and Caching Strategies," to appear in SIGMETRICS '95.

[Chen93] Chen, CM. M., Roussopoulos, N., "Adaptive Database Buffer Allocation Using Query Feedback," *Proc. of the 19th VLDB Conference, Dublin, Ireland, 1993, pp. 342-353.*

[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.

[Ebling94] Ebling, M.R., Mummert, L.B., Steere, D.C., "Overcoming the Network Bottleneck in Mobile Computing," *Proc. of the Workshop on Mobile Computing Systems and Applications*, December, 1994.

[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.

[Griffioen94] Griffioen, J., Appleton, R., "Reducing File System Latency using a Predictive Approach," *Proc. of the 1994 Summer USENIX Conference*, Boston, MA, 1994.

[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.

[Huang93] Huang, X.; Alleva, F.; Hsiao-Wuen Hon; Mei-Yuh Hwang; Kai-Fu Lee; Rosenfeld, R.; "The SPHINX-II speech recognition system: an overview," *Computer Speech and Language*, (UK); vol.7, no.2; April 1993; pp. 137-48.

[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.

[Kotz94] Kotz, D., "Disk-directed I/O for MIMD Multiprocessors," *Proc. of the First USENIX Symp. on Operating Systems Design and Implementation*, Monterey, CA, November, 1994, pp. 61-74.

[Lampson83] Lampson, B.W., "Hints for Computer System Design," *Proc. of the 9th Symp. on Operating System Principles*, Bretton Woods, N.H., 1983, pp. 33-48.

[Lee90] Lee, K.-F.; Hon, H.-W.; Reddy, R."An overview of the SPHINX speech recognition system," *IEEE Transactions on Acoustics, Speech and Signal Processing*, (USA); vol.38, no.1; Jan. 1990; pp. 35-45

[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.

[NCSA89] National Center for Supercomputing Applications. "XDataSlice for the X Window System," 1989.

[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.

[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., Satyanarayanan, M., "A Status Report on Research in Transparent Informed Prefetching," *ACM Operating Systems Review*, V 27(2), April, 1993, pp. 21-34.

[Patterson94] Patterson, R.H., Gibson, G., "Exposing I/O Concurrency with Informed Prefetching," *Proc. of the Third Int. Conf. on Parallel and Distributed Information Systems*, Austin, TX, Sept. 28-30, 1994, pp. 7-16.

[Ravishankar93] Ravishankar, M. K. "Parallel Implementation of Fast Beam Search for Speaker-Independent Continuous Speech Recognition," Computer Science & Automation, Indian Institute of Science, Bangalore, India, 1993.

[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.

[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.

[Stathopoulos94] Stathopoulos, A., Fischer, C. F., "A Davidson program for finding a few selected extreme eigenpairs of a large, sparse, real, symmetric matrix," *Computer Physics Communications,* vol. 79, 1994, pp. 268-290.

[Stonebraker86] Stonebraker, M., Rowe, L, "The Design of Postgres," *Proceedings of ACM SIGMOD '86 International Conference on Management of Data*, Washington, DC, USA, 28-30 May 1986.

[Stonebraker90] Stonebraker, M., Rowe, L.A., Hirohama, M., "The implementation of POSTGRES," *IEEE Transactions on Knowledge and Data Engineering*, vol.2, no.1, March 1990, pp. 125-42

[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.

[Wu92] Wu, S. and Manber, U. "AGREP-a fast approximate pattern-matching tool," *Proc. of the Winter 1992 USENIX Conference*, San Francisco, CA, Jan. 1992, pp. 20-24.