Appears in *Proc. of the 15th ACM Symp. on Operating System Principles*, Copper Mountain Resort, CO, Dec. 3-6, 1995, pp. 79-95.

Informed Prefetching and Caching

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

*Department of Electrical and Computer Engineering
†School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
{rhp, garth, eginting, danner, jimz}@cs.cmu.edu
http://www.cs.cmu.edu/Web/Groups/PDL/

Abstract

In this paper, we present aggressive, proactive 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 dynamically allocate 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 cost-benefit analysis to allocate buffers where they will have the greatest impact. We have implemented informed prefetching and caching in Digital's OSF/1 operating system and measured its performance on a 150 MHz Alpha equipped with 15 disks running a range of applications. Informed prefetching reduces the execution time of text search, scientific visualization, relational database queries, speech recognition, and object linking by 20-83%. Informed caching reduces the execution time of computational physics by up to 42% and contributes to the performance improvement of the object linker and the database. Moreover, applied to multiprogrammed, I/O-intensive workloads, informed prefetching and caching increase overall throughput.

1 Introduction

Traditional disk and file buffer cache management is reactive; disk accesses are initiated and buffers allocated in response to application demands for file data. In this paper, we show that proactive disk and buffer management based on application-disclosed hints can dramatically improve performance. We show how to use these hints to prefetch aggressively, thus eliminating the I/O stalls incurred by accesses that would otherwise have missed in the cache, and how to keep hinted data in the cache in anticipation of reuse. At the core of our approach is a cost-benefit analysis which we use both to balance buffer usage for prefetching versus caching, and to integrate this proactive management with traditional LRU (least-recently-used) cache management for non-hinted accesses.

Three factors make proactive I/O management desirable and possible:

1. the underutilization of storage parallelism,
2. the growing importance of file-access performance, and
3. the ability of I/O-intensive applications to offer hints about their future I/O demands.

Storage parallelism is increasingly available in the form of disk arrays and striping device drivers. These hardware and 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]. Trivially 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 non-sequential accesses. In such workloads, the service time of most disk accesses is dominated by seek and rotational latencies. Moreover, these workloads access one disk at a time while idling the other disks in an array. Disk arrays, by themselves, do not improve I/O performance for these workloads any more than multiprocessors improve the performance of single-threaded programs. Prefetching strategies are needed to "parallelize" these workloads.

The second factor encouraging our proactive I/O management is that ever-faster CPUs are processing data more quickly and encouraging the use of ever-larger data objects. Unless file-cache miss ratios decrease in proportion to processor performance, Amdahl's law tells us that overall system performance will increasingly depend on I/O-subsystem performance [Patterson88]. Unfortunately, simply growing the cache does not decrease cache-miss 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, after larger caches had been installed, hit ratios were not much changed — files had grown just as fast as the caches [Ousterhout85, Baker91]. This suggests that new techniques are needed to boost I/O performance.

The problem is especially acute for read-intensive applications. Write performance is less critical because the writing application generally does not wait for the disk to be written. In this common case, write behind can exploit storage parallelism even

This work was supported in part by Advanced Research Projects Agency contract DABT63-93-C-0054, in part by National Science Foundation grant ECD-8907068, and in part by donations and scholarships from Data General, Symbios Logic, IBM, Digital, and Seagate. The United States government has certain rights in this material. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any of the funding agencies.

when the application's writes are serial and non-sequential [Rosenblum91, Solworth90]. Examples of read-intensive applications include: text search, 3D scientific visualization, relational database queries, speech recognition, object code linkers, and computational physics. In general, these programs process large amounts of data relative to file-cache sizes, exhibit poor access locality, perform frequent non-sequential accesses, and stall for I/O for a significant fraction of their total execution time.

Yet, all of these applications' access patterns are largely predictable. This predictability could be used directly by the application to initiate asynchronous I/O accesses. But this sort of explicit prefetching can cripple resource management. First, the depth to which an application needs to prefetch depends on the throughput of the application, which varies as other applications place demands on the system. Second, asynchronously fetched data may eject useful data from the file cache. Third, 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 parallel I/O. Finally, the specializations a programmer puts into overcoming these problems may not be appropriate when the program is ported to a different system.

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.

The rest of this paper explains and justifies proactive I/O management based on informed prefetching and caching. Sections 2 and 3 review related work and describe disclosure-based hints. Section 4 develops our cost-benefit model and Section 5 describes its implementation in Digital's OSF/1 v2.0A file system. Section 6 describes our experimental testbed. Benchmark applications and single-application performance experiments are presented in Section 7. Section 8 presents multiple application experimental results. Finally, Sections 9 and 10 provide directions for future research and conclusions.

2 Related 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 LRU cache replacement algorithm relies on the history of recent accesses to choose a buffer for replacement. For history-based prefetching, the most successful approach is sequential readahead [Feiertag71, McKusick84]. Digital's 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 [Sacco82, 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. In one notable example, however, Cao, Felton, Karlin and Li derive an aggressive prefetching policy with excellent competitive performance characteristics in the context of complete knowledge of future accesses [Cao95a]. These same authors go on to show how to integrate prefetching according to hints with application-supplied cache management advice [Cao95b]. In contrast, we use the same hints, described in the next section, for both caching and prefetching.

Much richer languages for expressing and exploiting disclosure include collective I/O calls [Kotz94] and operations on structured files [Grimshaw91] or dynamic sets [Steere95].

3 Hints that disclose

The proactive management strategy described in this paper depends on a reliable picture of future demands. We advocate a 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. Such hints stand in contrast to hints which give advice. For example, an 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. Disclosure is simply a programmer revealing knowledge of the application's behavior.

Disclosure has three 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. As such, disclosure is a mechanism for portable I/O optimizations. Second, because disclosure provides the evidence for a policy decision, rather than the policy decision itself, it is more robust. Specifi-

File Specifier		Pattern Specifier
file name	×	sequential whole file
file descriptor		list of <offset, length>

Figure 1. The disclosure hint interface. Disclosure hints describe future requests in the same terms as the existing interface. Thus, our file system hints have two components, a file specifier and pattern specifier. The file specifier describes the file either by name or file descriptor. The pattern specifier describes the access pattern within the file. Currently, we support two pattern specifiers: a file read sequentially from beginning to end, or read according to an ordered list of <offset, length> intervals. Thus, there are currently four different forms of hints.

cally, 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 (ioctl) system call. As shown in Figure 1, hints specify a file and an access pattern for the file. There may be multiple outstanding hints, and the order in which hints are given indicates the order of the subsequent accesses.

Given disclosing hints, proactive management can deliver three primary benefits:

1. informed prefetching can "parallelize" the I/O request stream and take advantage of disk arrays to eliminate I/O stalls;
2. informed caching can hold on to useful blocks and outperform LRU caching independent of prefetching; and
3. informed disk management can schedule future disk I/Os to reduce access latency, and batch multiple requests for increased access efficiency.

This paper demonstrates the first two of these benefits.

4 Cost-benefit analysis for I/O management

The I/O manager's goal is to deploy its limited resources to minimize I/O service time. At its disposal are disk arms and file cache buffers. But, because we are primarily concerned with the exploitation of storage parallelism, we assume an adequate supply of disk arms and focus on the allocation of cache buffers.

Bidding to acquire cache buffers are two consumers: demand accesses that miss in the cache, and prefetches of hinted blocks. Holding out are two buffer suppliers: the traditional LRU cache, and the cache of hinted blocks. The I/O manager must resolve this tension between buffer consumers and suppliers.

In this section, we develop a framework for cache management based on cost-benefit analysis. We show how to estimate the benefit (decrease in I/O service time) of giving a buffer to a consumer and the cost (increase in I/O service time) of taking a buffer from a supplier. Finally, we show how to use these estimates to decide whether a buffer should be reallocated from a supplier to consumer, and, if so, how to pick the buffer for reallocation.

As shown in Figure 2, each potential buffer consumer and supplier has an *estimator* that independently computes the value of its use of a buffer. The buffer allocator continually compares these

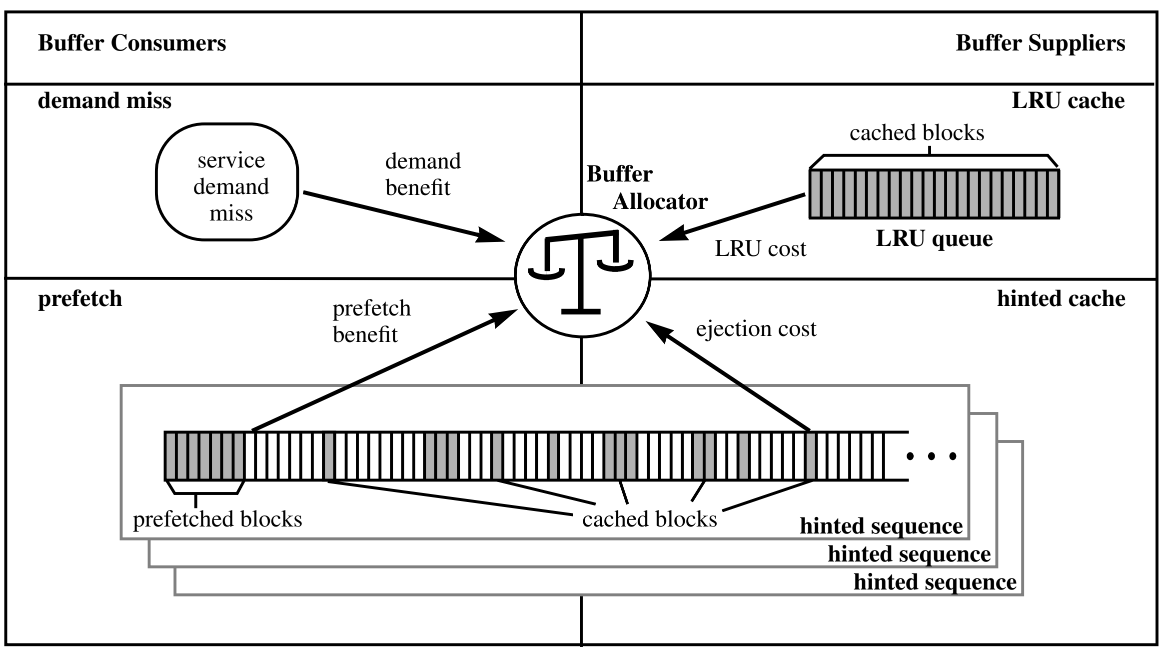


Figure 2. Informed cache manager schematic. Independent estimators express different strategies for reducing I/O service time. Demand misses need a buffer immediately to minimize the stall that has already started. Informed prefetching would like a buffer to initiate a read and avoid disk latency. To respond to these buffer requests, the buffer allocator compares their estimated benefit to the cost of freeing the globally least-valuable buffer. To identify this buffer, the allocator consults the two types of buffer suppliers. The LRU queue uses the traditional rule that the least recently used block is least valuable. In contrast, informed caching identifies as least valuable the block whose next hinted access is furthest in the future. The buffer allocator takes the least-valuable buffer to fulfill a buffer demand when the estimated benefit exceeds the estimated cost.

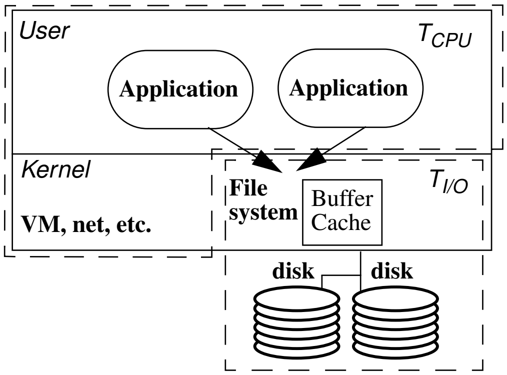


Figure 3. Components of system execution. In our simplified system model, application execution time, T, has two components, computation and I/O. The computational component, T_{CPU}, consists of user-level application execution plus time spent in kernel subsystems other than the file system. The I/O component, $T_{I/O}$, consists of time spent in the file system, which includes time for reading blocks, allocating blocks for disk I/Os, servicing disk interrupts, and waiting for a physical disk I/O to complete.

estimates and reallocates buffers when doing so would reduce I/O service time.

When comparing the different estimates, the buffer allocator must consider more than the absolute change in I/O service time; it must consider how much of the limited buffer resource is involved. Thus, we define the unit of buffer usage as the occupation of one buffer for one inter-access period and call it one *buffer-access*. Then we define the *common currency* for the expression of all value estimates as the *magnitude of the change in I/O service time per buffer-access*. With this common currency, the buffer allocator can meaningfully compare the independent value estimates and allocate buffers where they will have the greatest impact on I/O service time.

In the following sections, we define our system model and then develop each estimator's strategy for valuing buffers.

4.1 System model

We assume a modern operating system with a file buffer cache running on a uniprocessor with sufficient memory to make available a substantial number of cache buffers. With respect to our workload, consistent with our emphasis on read-intensive applications, we assume that all application I/O accesses request a single file block that can be read in a single disk access. Further, we assume that system parameters such as disk access latency, T_{disk}, are constants. Lastly, as mentioned above, we assume enough disk parallelism for there never to be any congestion (that is, there is no disk queueing). As we shall see, distressing as these assumptions may seem, the policies derived from this simple system model behave well in a real system, even one with a single congested disk.

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

$$T = N_{I/O}(T_{CPU} + T_{I/O}), \qquad (1)$$

where $N_{I/O}$ is the number of I/O accesses, T_{CPU} is the inter-access application CPU time, and $T_{I/O}$ is the time it takes to service an I/O access. Figure 3 diagrams our system model.

In our model, the I/O service time, $T_{I/O}$, includes some system CPU time. In particular, an access that hits in the cache experiences time T_{hit} to read the block from the cache. In the case of a cache miss, the block needs to be fetched from 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 disk operation completes. The total time to service an I/O access that misses in the cache, T_{miss}, is the sum of these times:

$$T_{miss} = T_{hit} + T_{driver} + T_{disk} \quad . \qquad (2)$$

In the terms of this model, allocating a buffer for prefetching can mask some disk latency. Deallocating an LRU cache buffer makes it more likely that an unhinted access misses in the cache and must pay a delay of T_{miss} instead of T_{hit}. Ejecting a hinted block from the cache means an extra disk read will be needed to prefetch it back later. In the next sections, we quantify these effects.

4.2 The benefit of allocating a buffer to a consumer

The two consumers of buffers are demand accesses that miss in the cache and prefetches of hinted blocks. Since any delay in servicing a demand miss adds to I/O service time, we treat requests from demand misses as undeniable and assign them infinite value. Computing the benefit of prefetching, explained below, is a bit harder.

Prefetching a block according to a hint can mask some of the latency of a disk read, T_{disk}. Thus, in general, an application accessing such a prefetched block will stall for less than the full T_{disk}. Suppose we are currently using x buffers to prefetch x accesses into the future. Then, stall time is a function of x, $T_{stall}(x)$, and the service time for a hinted read, also a function of x, is

$$T_{pf}(x) = T_{hit} + T_{driver} + T_{stall}(x) \ . \qquad (3)$$

The benefit of using an additional buffer to prefetch one access deeper is the change in the service time,

$$\Delta T_{pf}(x) = T_{pf}(x+1) - T_{pf}(x) \qquad (4)$$

$$= T_{stall}(x+1) - T_{stall}(x) \ . \qquad (5)$$

Evaluating this expression requires an estimate of $T_{stall}(x)$.

A key observation is that the application's data consumption rate is finite. Typically, the application reads a block from the cache in time T_{hit}, does some computation, T_{CPU}, and pays an overhead, T_{driver}, for future accesses currently being prefetched. Thus, even if all intervening accesses hit in the cache, the soonest we might expect a block x accesses into the future to be requested is $x(T_{CPU} + T_{hit} + T_{driver})$. Under our assumption of no disk congestion, a prefetch of this xth future block would complete in T_{disk} time. Thus, the stall time when requesting this block is at most

$$T_{stall}(x) \le T_{disk} - x(T_{CPU} + T_{hit} + T_{driver}) \ . \qquad (6)$$

Figure 4 shows this worst case stall time as a function of x.

This stall-time expression allows us to define the distance, in terms of future accesses, at which informed prefetching yields a zero stall time. We call this distance the *prefetch horizon*,

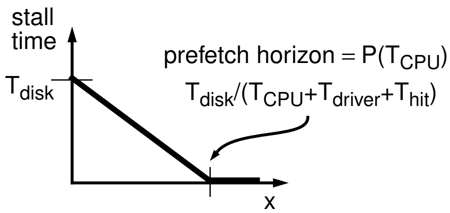


Figure 4. Worst case stall time and the prefetch horizon. Data consumption is limited by the time an application spends acquiring and consuming each block. This graph shows the worst case application stall time for a single prefetch x accesses in advance, assuming adequate I/O bandwidth, and therefore no disk queues. There is no benefit from prefetching further ahead than the prefetch horizon.

$P(T_{CPU})$, recognizing that it is a function of a specific application's inter-access CPU time.

$$P(T_{CPU}) = \frac{T_{disk}}{(T_{CPU} + T_{hit} + T_{driver})} . \tag{7}$$

Because there is no benefit from prefetching more deeply than the prefetch horizon, we can easily bound the impact of informed prefetching on effective cache size; prefetching a stream of hints will not lead informed prefetching to acquire more than $P(T_{CPU})$ buffers.

Equation (6) is an upper bound on the stall time experienced by the xth future access assuming that the intervening accesses are cache hits and do not stall. Unfortunately, it overestimates stall time in practice. In steady state, multiple prefetches are in progress and a stall for one access masks latency for another so that, on average, only one in x accesses experiences the stall in Equation (6). Figure 5 diagrams this effect. Thus, the average stall per access as a function of the prefetch depth, $P(T_{CPU}) > x > 0$, is

$$T_{stall}(x) = \frac{T_{disk} - x(T_{CPU} + T_{hit} + T_{driver})}{x} . \tag{8}$$

At x = 0, there is no prefetching, and $T_{stall}(0) = T_{disk}$. Similarly, for $x \geq P(T_{CPU})$, $T_{stall}(x) = 0$. Figure 6 shows that this estimate, though based on a simple model, is a good predictor of the actual stall time experienced by a synthetic application running on a real system.

We can now plug Equation (8) into Equation (5) and obtain an expression for the impact on I/O service time of acquiring one additional cache buffer to increase the prefetching depth,

$$\Delta T_{pf}(x) = \begin{cases} x = 0 & -(T_{CPU} + T_{hit} + T_{driver}) \\ x < P(T_{CPU}) & \dfrac{-T_{disk}}{x(x+1)} \\ x \geq P(T_{CPU}) & 0 \end{cases} . \tag{9}$$

Every access that this additional buffer is used for prefetching benefits from this reduction in the average I/O service time. Thus, Equation (9) is the change in I/O service time per buffer-access, and the magnitude of this change is the value of allocating a buffer for prefetching in terms of the common currency.

Having estimated the benefit of giving a buffer to a demand miss or prefetch consumer, we now consider the cost of freeing a buffer that could be used to obtain these benefits. We estimate the cost first of taking a buffer from the LRU queue and then of ejecting a hinted block to take the buffer it occupies.

4.3 The cost of shrinking the LRU cache

Over time, the portion of demand accesses that hit in the cache is given by the cache-hit ratio, $H(n)$, a function of the num-

access number	Time (1 time-step = T_{CPU} + T_{hit} + T_{driver})																				
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	I	⊖	⊖	⊖	⊖	⊗															
2	I	-	-	-	-	C	X														
3	I	-	-	-	-	C		X													
4						I	-	-	⊖	⊖	⊗										
5							I	-	-	-	-	⊗									
6								I	-	-	-	-	⊗								
7											I	-	-	⊖	⊖	⊗					
8												I	-	-	-	-	⊗				
9													I	-	-	-	-	⊗			
10																I	-	-	⊖	⊖	⊗

I : initiate prefetch - : prefetch in progress C : block arrives in cache X : consume block O : stall

Figure 5. Average stall time when using a fixed number of buffers for parallel prefetching. This figure illustrates informed prefetching as a pipeline. In this example, three prefetch buffers are used, prefetches proceed in parallel, T_{CPU} is fixed, and $P(T_{CPU}) = 5$. At time T=0, the application gives hints for all its accesses and then requests the first block. Prefetches for the first three accesses are initiated immediately. The first access stalls until the prefetch completes at T=5, at which point the data is consumed and the buffer is reused to initiate the fourth prefetch. Accesses two and three proceed without stalls because the latency of prefetches for those accesses is overlapped with the latency of the first prefetch. But, the fourth access stalls for $T_{stall} = T_{disk} - 3(T_{CPU}+T_{hit}+T_{driver})$. The next two accesses don't stall, but the seventh does. The application settles into a pattern of stalling every third access. In general, when x buffers are used for prefetching, a stall occurs once every x accesses.

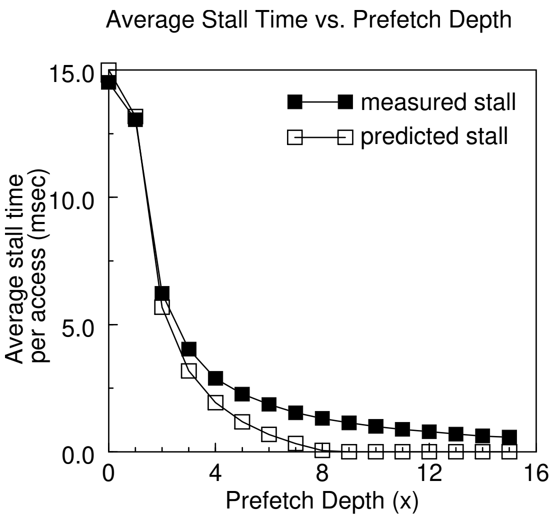


Figure 6. Predicted and measured per-access stall time. To verify the utility of Equation (8), we measured the stall time of a synthetic microbenchmark as we varied prefetch depth. The benchmark does 2000 reads of random, unique 8K blocks from a 500 MB file striped over 15 disks. It has 1 millisecond of computation between reads, so T_{CPU} = 1ms, and for the system described in Section 6, T_{hit}+T_{driver} = 823µs and T_{disk} = 15ms. Overall, Equation (8) has a maximum error of about 2 milliseconds, making it is a good predictor of actual stall time. The equation underestimates stall time because the underlying model neglects two factors, disk contention and variation in T_{disk}. Deeper prefetching increases the chance that two or more accesses contend for the same disk and add unmodelled stalls. Variability in T_{disk} has a more subtle effect. Longer than average disk accesses may be balanced in number and duration by shorter than average accesses, but the former always add stall time to the measurement, while the latter only reduce stall time if their access time is not fully overlapped. With deeper prefetching most accesses are well overlapped, so shorter accesses do not reduce measured stall time. Effectively, variability in T_{disk} makes a constant T_{disk} appear longer.

ber of buffers in the cache, *n*. Given *H(n)*, the average time to service a demand I/O request, denoted $T_{LRU}(n)$, is

$$T_{LRU}(n) = H(n)T_{hit} + (1 - H(n))T_{miss} \quad . \tag{10}$$

Taking the least-recently-used buffer from a cache employing an LRU replacement policy results in an increase in the average I/O service time of

$$\Delta T_{LRU}(n) = T_{LRU}(n-1) - T_{LRU}(n)$$

$$= (H(n) - H(n-1))(T_{miss} - T_{hit}) \quad . \tag{11}$$

Since *H(n)* varies as the I/O workload changes, our LRU cache estimator dynamically estimates *H(n)* and the value of this expression as explained in Section 5.1.

Every access that the LRU cache is deprived of this buffer will, on average, suffer this additional I/O service time. Thus, Equation (11) is in terms of the common currency, magnitude of change in I/O service time per buffer-access.

4.4 The cost of ejecting a hinted block

Though there is no benefit from prefetching beyond the prefetch horizon, caching any block for reuse can avoid the cost of prefetching it back later. Thus, ejecting a block increases the service time for the eventual access of that block from a cache hit, T_{hit}, to the read of a prefetched block, T_{pf}. If the block is prefetched back *x* accesses in advance, then the increase in I/O service time caused by the ejection and subsequent prefetch is

$$\Delta T_{eject}(x) = T_{pf}(x) - T_{hit} \tag{12}$$

$$= T_{driver} + T_{stall}(x) \; . \tag{13}$$

Though the stall time, $T_{stall}(x)$, is zero when *x* is greater than the prefetch horizon, T_{driver} represents the constant CPU overhead of ejecting a block no matter how far into the future the block will be accessed.

The cost of ejecting a block, $\Delta T_{eject}(x)$, does not affect every access; it only affects the next access to the ejected block. Thus, to express this cost in terms of the common currency, we must average this change in I/O service over the accesses that a buffer is freed. If the hint indicates the block will be read in *y* accesses, and the prefetch happens *x* accesses in advance, then ejection frees one buffer for a total of *y*–*x* buffer-accesses. Conceptually, if the block is ejected and its buffer lent where it accrues an savings in average I/O service time, then it will have *y*–*x* accesses to accrue a total savings that exceeds the cost of ejecting the block.

Averaging over *y*–*x* accesses, the increase in service time per buffer-access is

$$\Delta T_{eject}(x, y) = \frac{T_{driver} + T_{stall}(x)}{y - x} \; , \tag{14}$$

where $T_{stall}(x)$ is given by Equation (8). As we shall see in Section 5.3, our implementation simplifies this estimate further to eliminate the dependence on the variable *x*.

4.5 Putting it all together: global min-max valuation

Figure 7 summarizes the absolute value of Equations (9), (11), and (14) which the various estimators use to determine the local value of a buffer. Before comparing these values, the buffer allocator must normalize these local estimates by the relative rates of accesses to each estimator. Thus, the LRU cache estimate is multiplied by the rate of unhinted demand accesses, r_d, while the estimates for each hint sequence are multiplied by the rate of accesses to that sequence, r_h.

The buffer allocator uses these normalized estimates to decide when to take a buffer from a supplier and use it to service a request for a buffer. For example, deallocating a buffer from the LRU cache and using it to prefetch a block would cause a net reduction in aggregate I/O service time if $r_d|\Delta T_{LRU}(n)| < r_h|\Delta T_{pf}(x)|$. For the greatest reduction, though, the globally least-valuable buffer should be allocated. Our algorithm for identifying this buffer is as follows.

Each supply estimator determines the costs of losing any of its buffers. If multiple estimators claim the same buffer, which happens, for example, when a hint refers to a block already in the LRU queue, then each estimators independently values the buffer. The global value of a buffer is the maximum of the normalized values provided by each of the independent supply estimators. The global value is not the sum because it only takes one disk I/O to fetch a block no matter how many times the block is accessed thereafter.

Buffer Consumers	**Buffer Suppliers**
demand miss ∞	**LRU cache** $(H(n) - H(n-1))(T_{miss} - T_{hit})$
prefetch $x = 0$: $T_{CPU} + T_{hit} + T_{driver}$ $x < P(T_{CPU})$: $\frac{T_{disk}}{x(x+1)}$ $x \geq P(T_{CPU})$: 0	**hinted cache** $\frac{T_{driver} + T_{stall}(x)}{y - x}$

Figure 7. Local value estimates. Shown above are the locally estimated magnitudes of the change in I/O service time per buffer-access for the buffer consumers and suppliers of Figure 2. Since demand misses must be satisfied immediately, they are treated as having infinite value. The remaining three formulas are the absolute values of Equations (11), (14), and (9), for the LRU cache, hinted cache, and prefetch estimates, respectively.

The globally least-valuable buffer is the one whose maximum valuation is minimal over all buffers. Hence, our replacement policy employs a global min-max valuation of buffers. While the overhead of this estimation scheme might seem high, in practice, as we shall see in Section 5, the value of only a small number of buffers needs to be determined to find the globally least-valuable.

4.6 An example: emulating MRU replacement

As an aid to understanding how informed caching 'discovers' good caching policy, we show how it exhibits MRU (most-recently-used) behavior for a repeated access sequence. Figure 8 illustrates an 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-ratio function, $H(n)$, indicates that the least-recently-used blocks in the LRU queue don't get many hits, then these blocks 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-ratio functions, the fewer the buffers in the LRU cache, the more valuable they are. Eventually, the cost of taking another LRU buffer exceeds the cost of ejecting the most-recently-consumed hinted block. At the next prefetch, this MRU block is ejected 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 ejecting moves through the remaining blocks of the first iteration.

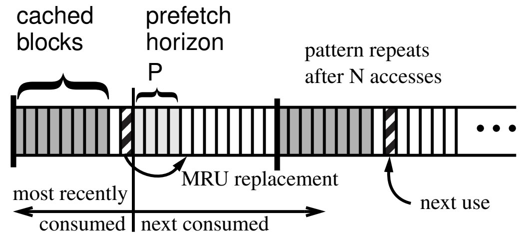


Figure 8. MRU behavior of the informed cache manager on repeated access sequences. The number of blocks allocated to caching for a repeated access pattern grows until the caching benefit is not sufficient to hold an additional buffer for the N accesses before it is reused. At that point, the least-valuable 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.

Because the prefetch horizon limits prefetching, there are never more than the prefetch horizon, $P(T_{CPU})$, 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 manager 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. This behavior is a result of valuing hinted, cached blocks and ejecting the block whose next access is furthest in the future when a buffer is needed. These techniques will improve cache performance for arbitrary access sequences where blocks are reused with no particular pattern. All that is needed is a hint that discloses the access sequence.

5 Implementation of informed caching and prefetching

Our implementation of informed prefetching and caching, which we call TIP, replaces the unified buffer cache (UBC) in version 2.0A of Digital's OSF/1 operating system. To service unhinted demand accesses, TIP creates an LRU estimator to manage the LRU queue and estimate the value of its buffers. In addition, TIP creates an estimator for every process that issues hints to manage its hint sequence and associated blocks.

To find the globally least-valuable buffer, it is sufficient that each estimator be able to identify its least-valuable buffer and declare its estimated value. From the LRU estimator's perspective, the least-recently-used buffer is least valuable. For a hint estimator, because all disk accesses are assumed to take the same amount of time, the least-valuable buffer contains the block whose next access is furthest in the future. TIP takes these declared estimates, normalizes them by the relative access rates, and ranks the estimators by these normalized declared values.

When there is a demand for a buffer, TIP compares the normalized benefit of servicing the demand to the normalized declared cost of the lowest-ranked estimator. If there are multiple consumers with outstanding requests, TIP considers the requests in order of their expected normalized benefit. If the benefit exceeds the cost, TIP asks the lowest-ranked estimator to give up its least-valuable buffer. After doing so, the estimator stops *tracking* this buffer. As far as it is concerned, the buffer is gone. It identifies a new least-valuable buffer from among the buffers it is still tracking and declares its value. TIP then reranks the estimators if necessary.

Before the block is actually ejected, TIP checks to see if any other estimator would value the buffer more than the cost of the lowest-ranked estimator. If so, that estimator starts tracking the buffer, including it when identifying its least-valuable buffer. The request for a buffer is then reconsidered from the start. At some later time, when this new estimator picks this almost-ejected buffer for replacement, the first estimator will get a chance to revalue the buffer and resume tracking it. A data structure keeps track of which estimators value a buffer at all to make this search for another estimator fast.

Once TIP is sure that no estimator values the buffer more than the current global minimal amount, the block is ejected and the buffer reallocated.

Since only tracked blocks are ever picked for replacement, all blocks must be tracked by at least one estimator. 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 TIP uses a special orphan estimator to track the block until it can be replaced.

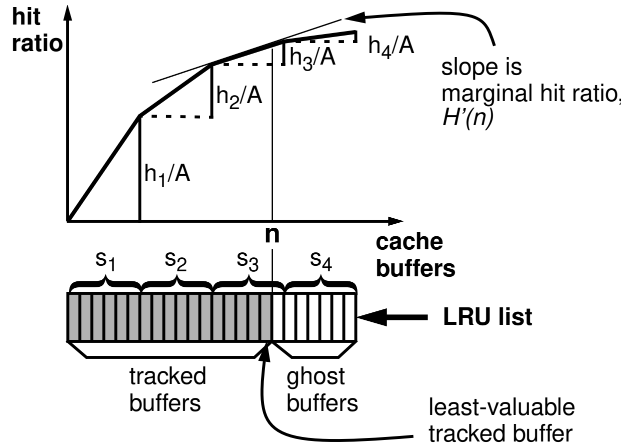


Figure 9. Piecewise estimation of *H(n)*. The LRU list is broken into segments, s_1, s_2, s_3, … Each buffer is tagged to indicate which segment it is in. The tag is updated when a buffer passes from one segment to the next. When there is a cache hit in segment *i*, the segment hit count, h_i, is incremented. That segment's contribution to the hit ratio is then h_i/A, where *A* is the total number of accesses to the LRU cache.

5.1 Implementing LRU estimation

LRU block replacement is a stack algorithm, which means that the ordering of blocks in the LRU queue is independent of the size of the cache. 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 ratio as a function of the number of buffers, *n*, in the cache for any cache size less than *N*, $0 < n < N$. Specifically, *H(n)* is estimated by the sum of the number of hits with stack depths less then or equal to *n* divided by the total number of accesses to the LRU cache, *A*.

In TIP, the number of buffers in the LRU stack varies dynamically. Thus, to determine *H(n)* for caches larger than the current size, TIP uses ghost buffers. Ghost buffers are dataless buffer headers which serve as placeholders to record when an access would have been a hit had there been more buffers in the cache [Ebling94]. The length of the LRU queue, including ghosts, is limited to the total number of buffers in the cache.

To reduce overhead costs and estimate variation, hit counts are recorded not by individual stack depths, but by disjoint intervals of stack depths, called segments. Shown in Figure 9, this allows a piecewise estimation of *H(n)*.

The cost of losing an LRU buffer given in Equation (11) requires an estimate of $\Delta H(n)=H(n)-H(n-1)$. Direct evaluation with a piecewise estimate of *H(n)* yields a function that is zero everywhere, except at segment boundaries. Instead, we estimate $\Delta H(n)$ with the marginal hit ratio, $H'(n)$, the slope of *H(n)*. Given our piecewise estimate of *H(n)*, we can estimate $\Delta H(n)$,

$$\Delta H(n) \approx 1 \cdot H'(n) \approx \frac{h_i}{A|s_i|} \quad , \tag{15}$$

where *n* falls within segment s_i, *A* is the total number of accesses to the LRU, and $|s_i|$ represents the number of buffers in segment s_i. In our implementation, $|s_i| = 100$.

A final complexity arises because, in general, *H(n)* may not be similar to the smooth function suggested by Figure 9. There is often a large jump in the hit ratio when the entire working set of an application fits into the buffer cache. TIP's LRU estimator uses a simple mechanism to avoid being stuck in a local minima that ignores the benefit of a much larger cache: $\Delta H(n)$ is modified to be $max_{i \geq n}\{H'(i)\}$; that is, the value of the marginal hit ratio is rounded up to the value of any larger marginal hit ratio occurring deeper in the LRU stack. Thus, if the LRU cache is currently small, but a larger cache would achieve a much higher hit ratio, this mechanism encourages the cache to grow.

This gives us the following expression for the cost of losing an LRU buffer:

$$\Delta T_{LRU}(n) \approx max_{i \geq n}\{H'(i)\}(T_{miss} - T_{hit}) \quad . \tag{16}$$

5.2 Implementing informed prefetching estimations

Section 4 presents two expressions, Equation (7) for determining the prefetch horizon, and Equation (9) for estimating the benefit of prefetching. To reduce estimation overhead and increase tolerance to both variation in application inter-access computation, T_{CPU}, and the need to prefetch other blocks, TIP assumes $T_{CPU} = 0$ and discounts the overhead of prefetching other blocks, T_{driver}, to arrive at a static, system-wide upper-bound on the prefetch horizon, $\hat{P}$,

$$\hat{P} = \frac{T_{disk}}{T_{hit}} . \tag{17}$$

To simplify the prefetcher's estimate of the value of acquiring a buffer, we recognize that it will obtain at least a few buffers and use the following variant of Equation (9)

$$\Delta T_{pf}(x) \approx \begin{cases} x = 0 & -T_{disk} \\ x < \hat{P} & \dfrac{-T_{disk}}{x(x+1)} \\ x \geq \hat{P} & 0 \end{cases} \quad . \tag{18}$$

Buffer Consumers	**Buffer Suppliers**
demand miss ∞	**LRU cache** $max_{i \geq n}\{H'(i)\}(T_{miss} - T_{hit})$
prefetch $x = 0 \quad T_{disk}$ $x < \hat{P} \quad \frac{T_{disk}}{x(x+1)}$ $x \geq \hat{P} \quad 0$	**hinted cache** $y = 1 \quad T_{driver} + T_{disk}$ $1 < y \leq \hat{P} \quad T_{driver} + \frac{T_{disk}}{y-1}$ $y > \hat{P} \quad \frac{T_{driver}}{y - \hat{P}}$

Figure 10. Local value estimates in the implementation. Shown above are the local estimates of the value per buffer-access for the buffer consumers and suppliers of Figure 2. These estimates are easy-to-compute approximations of the exact estimates of Figure 7.

5.3 Implementing informed caching estimations

Equation (14) in Section 4 expresses the cost of ejecting a hinted block in terms of *y*, the number of accesses till the hinted read, and *x*, how far in advance the block will be prefetched back. To eliminate the overhead of determining the value of *x* dynamically, we simplify this expression by assuming that the prefetch will occur at the (upper bound) prefetch horizon, $\hat{P}$. If the block is already within the prefetch horizon, $y < \hat{P}$, we assume that the prefetch will occur at the next access. Then, in accordance with the assumptions of Section 5.2 used to compute $\hat{P}$, we set $T_{CPU} = 0$, neglect T_{driver}, and take $T_{stall}(y) \approx (T_{disk} - yT_{hit})/y$, for $1 < y < \hat{P}$. Plugging into Equation (14), we get, for $1 < y < \hat{P}$

$$\Delta T_{eject}(y) = T_{driver} + \frac{T_{disk}}{y-1} - T_{hit}. \qquad (19)$$

Unfortunately, using this equation could lead to prefetching a block back shortly after ejecting it. To avoid this thrashing, there should be hysteresis in the valuations; that is, we need $|\Delta T_{eject}(y)| > |\Delta T_{pf}(y-1)| = T_{disk}/y(y-1)$. Comparing this expression to Equation (19), we see that the inequality does not hold for all possible values of T_{driver}, T_{disk}, and T_{hit}. To guarantee robustness for all values of these parameters greater than zero, we choose to add T_{hit} to $\Delta T_{eject}(y)$ for $1 < y < \hat{P}$. Thus, we have,

$$\Delta T_{eject}(y) \approx \begin{cases} y = 1 & T_{driver} + T_{disk} \\ 1 < y \leq \hat{P} & T_{driver} + \dfrac{T_{disk}}{y-1} \\ y > \hat{P} & \dfrac{T_{driver}}{y - \hat{P}} \end{cases} . \qquad (20)$$

Figure 7 summarizes the equations used to estimate buffer values in our implementation.

5.4 Exploiting OSF/1 clustering for prefetches

OSF/1 derives significant performance benefits from clustering the transfer of up to eight contiguous blocks into one disk access. One might ask of the informed prefetcher: when should buffers be allocated to prefetch secondary blocks as part of a cluster?

If the decision to prefetch a block has already been made, then the cost, T_{driver}, of performing a disk read will be paid. Any blocks that could piggyback on this read avoid most of the disk related CPU costs. If there are hinted blocks that can cluster with the required block, and they are not prefetched now in such a cluster, their later prefetch will incur the full overhead of performing a disk access and possibly the cost of any unmasked disk latency. These are exactly the costs considered when deciding whether to eject a hinted block. Thus, the decision to include an additional hinted contiguous block in a cluster is the same as the decision not to eject this additional hinted block once the prefetch is complete. If the informed cache would decide not to eject the block if it were in cache, then a buffer is allocated and the additional block is included in the pending cluster read.

6 Experimental testbed

Our testbed is a Digital 3000/500 workstation (SPECint92=84.4; SPECfp92=127.7), containing a 150 MHz Alpha (21064) processor, 128 MB of memory and five KZTSA fast SCSI-2 adapters each hosting three HP2247 1GB disks. This machine runs version 2.0A of Digital's OSF/1 monolithic kernel. OSF/1's file system contains a unified buffer cache (UBC) module that dynamically trades memory between its file cache and virtual memory. To eliminate buffer cache size as a factor in our experiments, we fixed the cache size 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 two outstanding requests at the drive. We forward two prefetch requests to reduce disk idle time between requests, and we don't forward more than two to limit priority inversion of prefetch over demand requests. The striper sorts queued prefetch requests according to C-SCAN.

System parameters for the TIP estimators were: T_{disk} = 15 milliseconds, T_{hit} = 243 microseconds, and T_{driver} = 580 microseconds. T_{hit} was measured by repeatedly reading a cached, hinted file, and dividing the elapsed time by the number of blocks read. T_{driver} was derived by measuring the non-idle time of a trivial application that hinted, then read, 2000 unique, non-sequential blocks of a 500MB file with the assumption that non-idle time equals 2000*(T_{hit}+T_{driver}). T_{disk} was estimated from direct measurements on a variety of applications.

In addition to the clustering fetches described in Section 5.4, the default OSF/1 file system implements 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.

We report results from two modified OSF/1 systems, TIP-1 and TIP-2, in addition to the default OSF/1 system. TIP-1, our first

Kernel	CPU Time	Elapsed Time
OSF/1, TIP-1	3,463	4,236
TIP-2	3,546	4,357

Table 1. Kernel build times. This table shows the total (non-hinting) build time for an OSF/1 2.0 kernel on an OSF/1 or TIP-1 kernel and on a TIP-2 kernel. All times are in seconds, and all kernels had the buffer cache size fixed at 12MB. TIP-2 is about 2.5% slower than OSF/1.

prototype, does informed prefetching but does not exploit hints for caching. It is integrated with the unified buffer cache in OSF/1, requiring only a few small hooks in the standard code. It uses a simple mechanism to manage resources: it uses up to $\hat{P}=62$ cache buffers 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. If the hinted block is already in the cache, the block is promoted to the tail of OSF/1's LRU list and counted as an unread buffer. When an application accesses a hinted block for the first time, TIP-1 reduces the count of unread buffers and resumes prefetching. Hinted but unread blocks may age out of the cache, triggering further prefetching, though this does not occur with any of our test applications.

TIP-1 has been running since mid 1993 in the 4.3 BSD FFS of 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 reduce the elapsed time of a seek-intensive data visualization tool (XDataSlice) by up to 70% [Patterson94]. During the summer of 1994 we ported TIP-1 to the current Alpha testbed to exploit its greater CPU and disk performance.

During 1994, we designed and began implementation of a second test system, TIP-2, which exploits hints for both informed prefetching and informed caching. It completely replaces the unified buffer manager in OSF/1 as described in Sections 3, 4, and 5.

To estimate the overhead of our TIP-2 system, we timed the complete build of an OSF/1 kernel. Table 1 summarizes the results. TIP-2 adds about 2.4% CPU overhead and 2.8% elapsed time for the build. CPU overhead for TIP-2 is dependent on I/O intensity. Therefore, overheads for our suite of I/O-intensive benchmarks, tend to be higher than this. They are: Davidson, 7%; XDataSlice, 13%; Sphinx, 1.9%; Agrep, 13%; Gnuld, 10%; and Postgres, 1.8% and 3.5% respectively for the low-match and high-match joins. The current system is tuned only for fidelity in the estimation of $H(n)$, and not for low overhead.

Our goal with informed prefetching is to exploit unused disk parallelism and convert our benchmark applications from being I/O-bound to being CPU-bound. Informed caching tries to further reduce the number of I/Os. The key performance metrics are elapsed time, I/O stall time, and CPU busy time. To obtain accurate measures of elapsed time, we used the Alpha processor cycle counter. To measure idle time, we kept a running counter of the number of processor cycles spent in the idle loop, taking care to exclude time spent servicing interrupts that occurred during the idle loop.

7 Single-application performance

In this section, we evaluate the performance of our informed prefetching and caching systems with a suite of six I/O-intensive benchmarks. All are single-threaded, synchronous, and I/O-bound in common usage. Five derive substantial benefit from prefetching alone. Three benefit from informed caching, especially when there is insufficient disk bandwidth available.

We report the results of each application run without competition on arrays of 1 to 10 disks (performance with 15 disks is essentially the same as with 10 disks). We report execution and I/O stall time for each application when not giving hints and when giving hints to the TIP-1 and TIP-2 systems. Each test was run on a system with a cold cache. Before each sequence of five 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. The standard deviation for both the elapsed time and stall time was less than 3% of the mean for all of these measurements.

7.1 MCHF Davidson algorithm

The Multi-Configuration Hartree-Fock, MCHF, is a suite of computational-physics programs which we obtained from Vanderbilt University where they are used for atomic-physics calculations. The Davidson algorithm [Stathopoulos94] is an element of the suite that computes, by successive refinement, the extreme eigenvalue-eigenvector pairs of a large, sparse, real, symmetric matrix stored on disk. In our test, the size of this matrix is 16.3 MB.

The Davidson algorithm iteratively improves its estimate of the extreme eigenpairs by computing the extreme eigenpairs of a much smaller, derived matrix. Each iteration 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 straightforward. At the start of each iteration, the Davidson algorithm discloses the whole-file, sequential read anticipated in the next iteration.

Figure 11(a) reports the elapsed time of the entire computation on OSF/1 (TIP-1 without hints is just OSF/1), when not giving hints to TIP-2, and when giving hints to TIP-1 and TIP-2. As with most of the figures in this section, data is striped over 1 to 10 disks, and the cache size is 12 MB. With or without hints, Davidson benefits significantly from the extra bandwidth of a second disk but then becomes CPU-bound. Because the hints disclose only sequential access in one large file, OSF/1's aggressive readahead matches the performance of TIP-1's informed prefetching and, in fact, performs slightly better because it incurs less overhead.

Neither OSF/1 nor informed prefetching in TIP-1 uses the 12 MB of cache buffers well. Because the 16.3 MB matrix does not fit in the cache, the LRU replacement algorithm ejects all of the blocks before any of them are reused. The informed cache manager in TIP-2, however, effectively reuses cache buffers, reducing the number of blocks fetched from 125,340 to 53,200. On one disk, this reduces elapsed time by over 30%. When disk bandwidth is inadequate, 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, as can be seen by comparing TIP-2 no

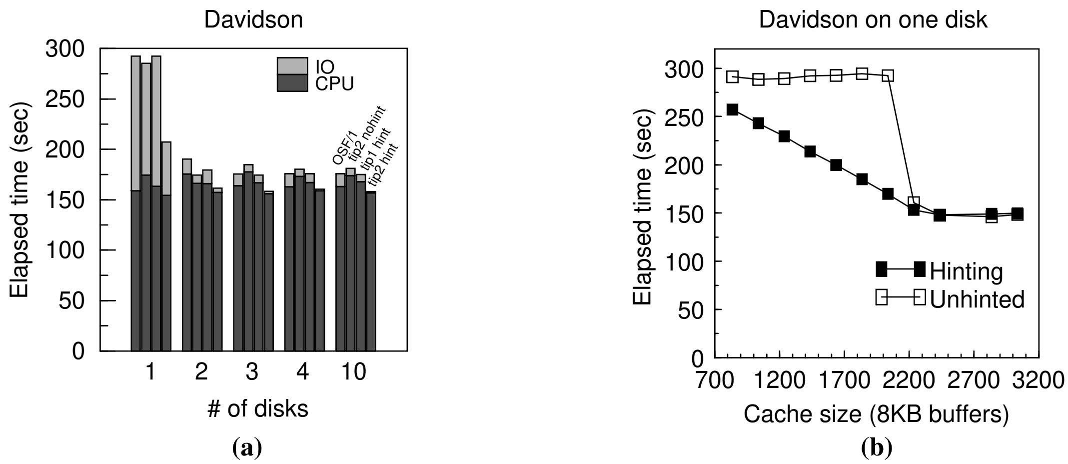


Figure 11. Benefit of informed caching for repeated accesses. Figure (a) shows the performance of the Davidson algorithm applied to a computational-physics problem. The algorithm repeatedly reads a large file sequentially. OSF/1's aggressive readahead algorithm performs about the same as TIP-1 with hints for this access pattern. Informed caching in TIP-2 reduces elapsed time by more than 30% on one disk by avoiding 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 in TIP-2 discovers an MRU-like policy which uses additional buffers to increase cache hits and reduce execution time. TIP-2 takes advantage of a 16 MB cache to reduce execution time by 42%. In contrast, LRU caching derives no benefit from additional buffers until there are enough of them to cache the entire dataset, which is 16.3 MB (2089 8K blocks).

hint and hint CPU times. Figure 11(b) shows Davidson's elapsed time with one disk on TIP-2 with and without hints as a function of cache size. Without hints, extra buffers are of no use until the entire dataset fits in the cache. In contrast, TIP-2's min-max global valuation of blocks yields the smooth exploitation of additional cache buffers that is expected from an MRU replacement policy. 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. TIP-2 effectively balances the allocation of cache buffers between prefetching and caching.

7.2 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 images and therefore larger datasets. Informed prefetching invalidates the slow-disk assumption and makes out-of-core computing practical, even for interactive applications. To demonstrate this, we added an out-of-core capability to XDS.

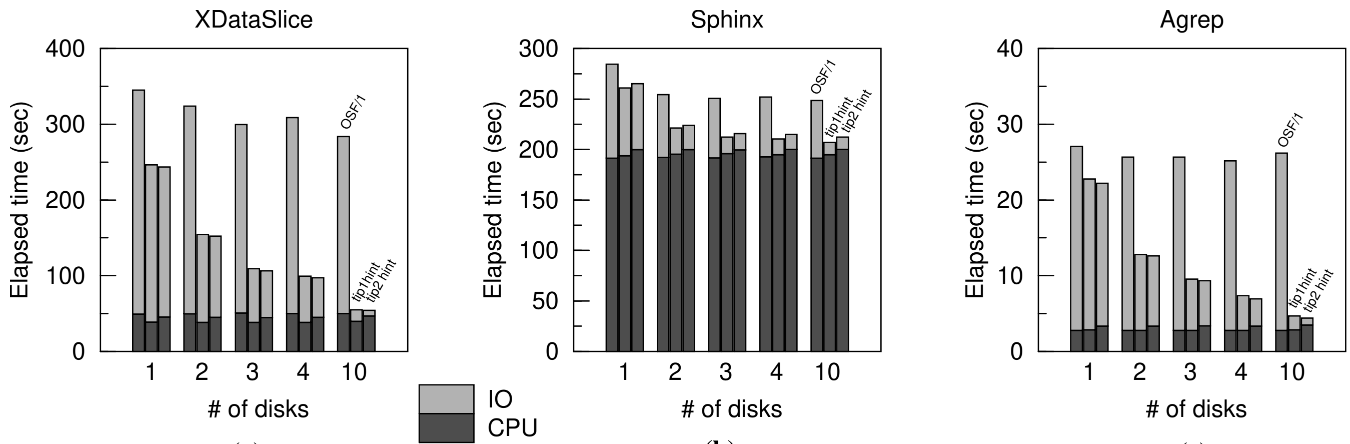
To render a slice through an in-core dataset, XDS iteratively determines which data point maps to the next pixel, reads the datum from memory, applies false coloring, and writes the pixel in the output pixel array. To render a slice from an out-of-core dataset, XDS splits this loop in two. Both to manage 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 ejects 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 [Patterson94].

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 and is stored on the disk in Z-major order. Our test renders 25 random slices through the dataset. Figure 12(a) reports the average elapsed time per slice on OSF/1, TIP-1 and TIP-2.

While OSF/1 readahead is effective for the sequential access pattern of Davidson, it is detrimental for XDS. XDS frequently reads a short sequential run, which triggers an equal amount of readahead by OSF/1. Only slices closely aligned with the Z-axis read long runs of sequential blocks for which the readahead is effective. Consequently, for this set of 25 slices, the nonhinting version of XDS reads 1.86 times as much data from disk as the application actually consumes. This combination of false readahead and lack of I/O parallelism causes XDS to take about 12 seconds to render an arbitrary slice without hints, leading to unacceptable interactive performance.

In contrast, informed prefetching both avoids false readahead and exploits the concurrency of a disk array. TIP-1 eliminates 70% of the I/O stall time on four disks, and 92% on 10 disks. On 10 disks, TIP-1 reduces the time to render a random slice by a factor of 6 to about 2 seconds, resulting in a much more tolerable interactive latency.

TIP-1 and TIP-2 perform similarly. However, because TIP-2 can use hints to coalesce into one disk read blocks that are contiguous on disk but widely separated in the access sequence, TIP-2 reduces the number of distinct disk reads from 18,700 to 15,000.



(a) (b) (c)

Figure 12. Elapsed time of visualization, speech recognition and search. Figure (a) shows the elapsed time for rendering 25 random slices through a 512 MB dataset. Without TIP, OSF/1 makes poor use of the disk array. But, informed by hints, TIP is able to prefetch in parallel and mask the latency of the many seeks. There is very little data reuse, so the informed caching does not decrease elapsed time relative to the simple prefetching in TIP-1. Figure (b) shows the benefits of informed prefetching for the Sphinx speech-recognition program. Sphinx is almost CPU-bound, so the improvements are less dramatic. As for XDataSlice, there is little data reuse so informed caching provides no benefit over TIP-1, and, in fact, incurs some additional overhead. Figure (c) reports the elapsed time for searches through files in three different directories and shows the benefit of prefetching across files. Again, informed caching provides no improvement over informed prefetching.

This improved I/O efficiency contributes to the slight performance advantage of TIP-2 over TIP-1.

7.3 Sphinx

Sphinx [Lee90] is a high-quality, speaker-independent, continuous-voice, 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 pruner 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 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.

We additionally 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 12(b) shows the elapsed time of Sphinx recognizing the 18-second recording. Sphinx starts with one sequential read of the 200MB language model which benefits from the array without hints. But, with informed prefetching, it takes advantage of the array even for the many small accesses and thereby reduces execution time by as much as 17%.

Sphinx's internal cache and large datasets lead to little locality in its file system accesses. Thus, the informed caching in TIP-2 does not improve upon the performance of simple informed prefetching in TIP-1.

7.4 Agrep

Agrep, a variant of grep, was 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, it opens the files specified on its command line one at a time, in argument order, and reads each sequentially.

Since the arguments to Agrep completely determine the files it will access, Agrep can issue hints for all accesses upon invocation. Agrep simply loops through the argument list and informs the file system of the files it will 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. In such cases, informed prefetching has the advantage of being able to prefetch across files and not just within a single file.

In our benchmark, Agrep searches 1349 kernel source files occupying 2922 disk blocks for a simple string that does not occur in any of the files.

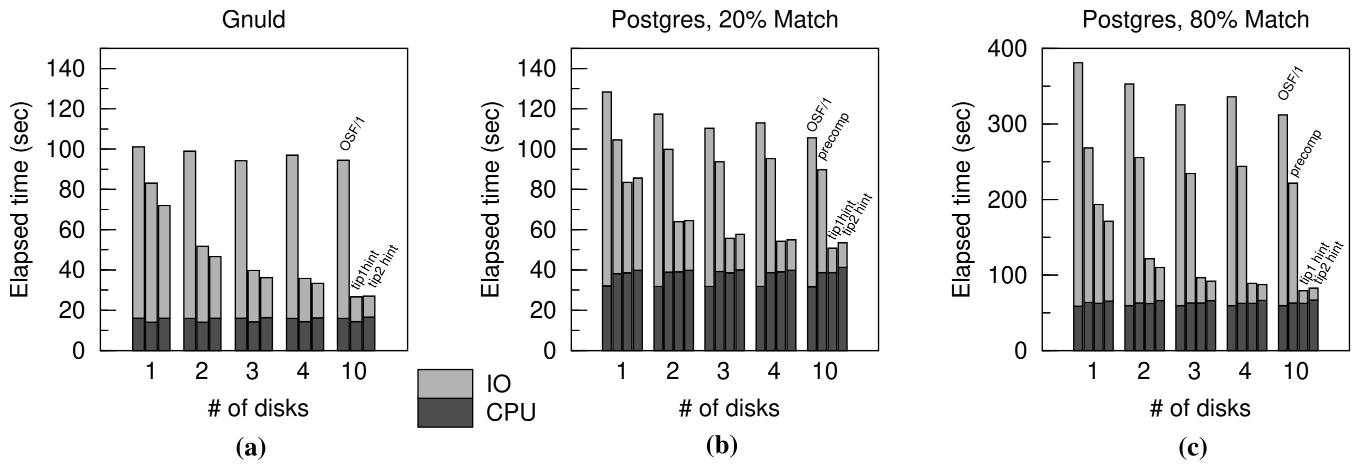


Figure 13. Elapsed time of Gnuld and Postgres. Figure (a) shows the elapsed time for Gnuld to link an OSF/1 TIP-1 kernel. Figures (b) and (c) show the elapsed time for two different joins in the standard Postgres relational database, a restructured Postgres that precomputes offsets for the inner relation, and in the restructured Postgres when it gives hints. The restructuring improves access locality and therefore cache performance, allowing it to run faster than standard Postgres. Delivering hints then dramatically reduces I/O stall time.

Figure 12(c) reports the elapsed time for this search. As was the case for XDataSlice and Sphinx, there is little parallelism in Agrep's I/O workload. The files are searched serially and most are small, so even OSF/1's readahead does not achieve parallel transfer. However, Agrep's disclosure of future accesses exposes potential I/O concurrency. On our testbed, arrays of as few as four disks reduce execution time by 73% and 10 disks reduce execution time by 83%.

7.5 Gnuld

Gnuld version 2.5.2 is the Free Software Foundation's object code linker which supports ECOFF, the default object file format under OSF/1. Gnuld performs many passes over input object files to produce the output linked executable. In the first pass, Gnuld reads each file's primary header, a secondary header, and its symbol and string tables. Hints for the primary header reads are easily given by replicating the loop that opens input files. The read of the secondary header, whose location is data dependent, is not hinted. Its contents provide the location and size of the symbol and string tables for that file. A loop splitting technique similar to that in XDataSlice is used to hint the symbol and string table reads.

After verifying that it has all the data needed to produce a fully linked executable, Gnuld makes a pass over the object files to read and process debugging symbol information. This involves up to nine small, non-sequential reads from each file. Fortunately, the previously read symbol tables determine the addresses of these accesses, so Gnuld loops through these tables to generate hints for its second pass.

During its second pass, Gnuld constructs up to five shuffle lists which specify where in the executable file object-file debugging information should be copied. When the second pass completes, Gnuld finalizes the link order of the input files, and thus the organization of non-debugging ECOFF segments in the executable file. Gnuld uses this order information and the shuffle lists to give hints for the final passes.

Our test links the 562 object files of our TIP-1 kernel. These objects file comprise approximately 64 MB, and produce an 8.8MB kernel. Figure 13(a) presents the elapsed and I/O stall time for this test.

Like XDataSlice, Gnuld without hints incurs a substantial amount of false readahead, causing it to read 125 MB from disk. In contrast, Gnuld reads only 95 MB with hints on TIP-1. The informed caching of TIP-2 further reduces the read volume to 85 MB. With hints, Gnuld eliminates 77% of its stall time with 4 disks and 87% with 10 disks. The remaining stall time is mostly due to the remaining unhinted accesses that Gnuld performs.

7.6 Postgres

Postgres version 4.2 [Stonebraker86, Stonebraker90] is an extensible, object-oriented relational database system from the University of California at 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). We run two cases. In the first, 20% of the outer relation tuples find a match in the inner relation. In the second, 80% find a match. One output tuple is written sequentially for every tuple match.

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 that tuple from the inner relation. From the perspective of storage, accesses to the inner relation and its index are random, defeating sequential readahead, and have poor locality, defeating caching. Thus, most of these inner-relation accesses incur the full latency of a disk read.

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 its sequential access), looks up each outer-relation tuple address in the index (unhinted), and stores the addresses in an array. Postgres then discloses these precomputed block addresses to TIP. In the second pass, Postgres rereads the outer relation but skips the index lookup and instead directly reads the inner-relation tuple whose address is stored in the array.

Figures 13(b) and 13(c) show the elapsed time required for the two joins under three conditions: standard Postgres, Postgres with the precomputation loop but without giving hints, and Postgres giving hints with the precomputation loop. 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 it is not polluted by the inner-relation data blocks. Even though Postgres reads the outer relation twice, there are about 900 and 6,100 fewer total disk I/Os in the precomputation-based runs of the first and second cases, respectively.

Invoking informed prefetching by issuing hints from the precomputation runs in TIP-1 allows concurrency for reads of inner-relation blocks and reduces elapsed time by up to 45% and 64% for the two cases, respectively. Compared to standard Postgres, precomputation and informed prefetching in TIP-1 reduce execution time by up to 55% and 75%.

Enabling informed caching with hints in TIP-2 in general has little effect on elapsed time because most I/O accesses are random reads from the inner relation. However, on one disk, in the 80% match case, TIP-2 gets an 11% reduction in elapsed time. While part of this benefit arises from informed caching, a large fraction arises from TIP-2's exploitation of clustering described in Section 5.4. The availability of hints allows contiguous blocks to be read in one disk I/O even though accesses to the two blocks may be widely separated in time. Informed clustering allows Postgres on TIP-2 to perform only 4,700 disk reads in the 20% match case and 8,600 disk reads in the 80% match case as compared to 6,700 and 12,300 on TIP-1, respectively. Clustering disk I/Os makes better use of disk bandwidth, so the benefit of informed clustering, like informed caching, is greatest when disk bandwidth is scarce (one disk).

8 Multiple-application performance

Multiprogramming I/O-intensive applications does not generally lead to equitable or efficient use of resources because these programs flush each other's working set and disturb each other's disk head locality. However, it is inevitable that I/O-intensive programs will be multiprogrammed. In the rest of this section, we present the implications of informed prefetching and caching on multiprogrammed I/O-intensive applications.

When multiple applications are running concurrently, the informed prefetching and caching system should exhibit three basic properties. First and foremost, hints should increase overall throughput. Second, an application that gives hints should improve its own performance. Third, in the interest of fairness, non-hinting applications should not suffer unduly when a competing application gives hints. Our cost-benefit model attempts to reduce the sum of the I/O overhead and stall time for all executing applications, and thus, we expect our resource management algorithms to also benefit multiprogrammed workloads.

To explore how well our system meets these performance expectations, we report three pairs of application executions: Gnuld/Agrep, Sphinx/Davidson, and XDS/Postgres. Here, Postgres performs the join with 80% matches and, precomputes its data accesses even when it does not give hints. For each pair of applications, we ran all four hinting and non-hinting combinations on TIP-2 starting the two applications simultaneously with a cold cache. Figures 14 through 16 show selected results.

Figure 14 shows the impact of hints on throughput for the three pairs of applications. We report the time until both applications complete, broken down by total CPU time and simultaneous stall time. In all cases, the maximum elapsed time decreases when one application gives hints, and decreases further still when both applications give hints. Simultaneous I/O stall time is virtually eliminated for two out of the three pairs when both applications give hints and the parallelism of 10 disks is available.

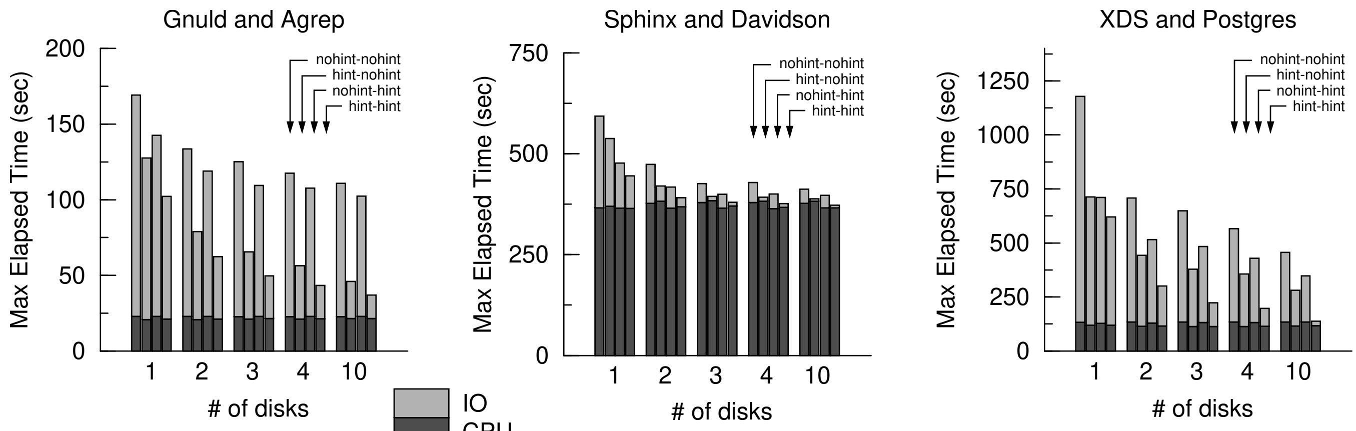
Figure 15 and Figure 16 show each named application's individual elapsed time after being initiated in parallel with another application (whose name is in parentheses). While vertical columns of graphs in Figures 14, 15, and 16 correspond to the same test runs, the middle two bars in any quartet of Figure 16 are swapped relative to the middle two bars in the corresponding quartets of Figures 14 and 15. So, for example, in Figure 15(a), 'hint-nohint' means Gnuld hints while Agrep does not, whereas in Figure 16(a) 'hint-nohint' means Agrep hints while Gnuld does not.

To see the impact of giving hints on an individual application's execution time when a second non-hinting application is run concurrently, compare bars one and two in Figures 15 and 16 Comparing bars three and four reveals the impact when the second application is giving hints. In most cases, giving hints substantially improves an application's execution time. A notable exception is Davidson when run with Sphinx as shown in Figure 16(b). When Davidson gives hints, informed caching reduces its I/O requirements so Sphinx's I/Os are serviced more quickly. Consequently, Sphinx demands more CPU time at the expense of Davidson and Davidson slows down. Recall, from Figure 14(b) that overall throughput increases when Davidson gives hints.

To see the impact on a non-hinting application of another application giving hints, compare the first and third bars in Figures 15 and 16. Comparing the second to fourth bars shows the impact on a hinting application. In two of six applications, a non-hinting application's execution time is increased by another application's hints. For example, in Figure 16(b), when Sphinx gives hints, it increases the execution time of a non-hinting Davidson. This is because, by giving hints, Sphinx stalls less often for I/O, so it competes more aggressively for the CPU at the expense of Davidson.

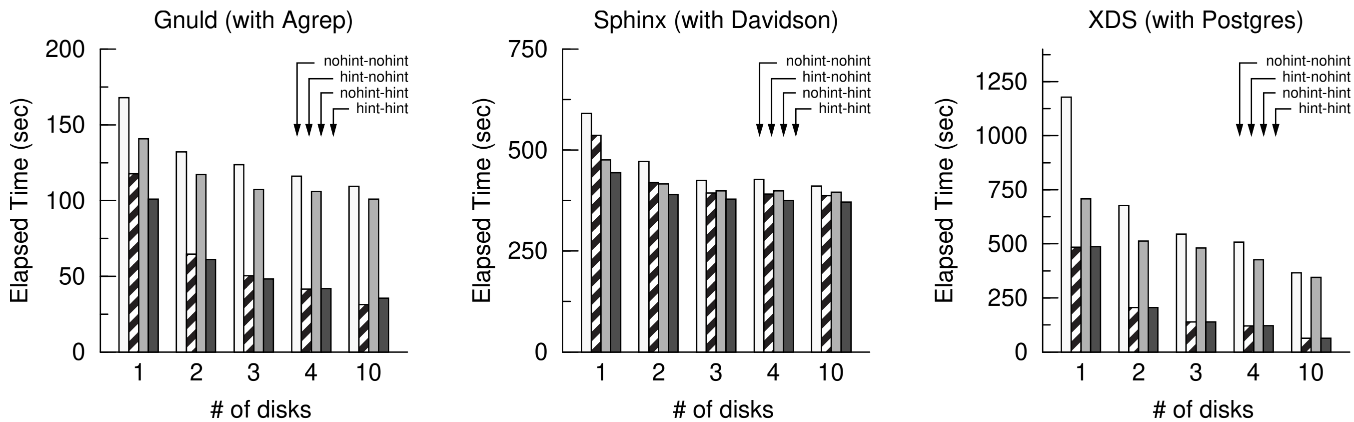
A more dramatic example is a non-hinting Agrep running with Gnuld shown in Figure 16(a). Here, CPU utilization is low even when the two applications run together; disk bandwidth determines performance. When neither application gives hints, they both usually have only one outstanding disk access at a time. From a single disk, about 40% of the accesses and 35% of the data transferred are attributable to Agrep over the course of its run. When Gnuld gives hints, prefetches queue up at the drive. Even though there is a limit of two prefetches queued in front of a demand request, Agrep's I/Os are more likely to be third in line instead of second. Agrep's share of disk accesses drops to about 24% and of data transferred to about 22%. Since Agrep is disk-bound and getting a smaller fraction of disk utilization, it takes longer to run.

In other cases, however, an application's hints benefit the other running application. For example, if either Postgres or XDS gives hints, the non-hinting other's elapsed time is substantially reduced. Multiprogramming this pair of applications causes both to run longer than the sum of their stand-alone elapsed times because interleaving their accesses dramatically reduces disk locality. So, when either gives hints, its I/Os are processed more efficiently. This allows it to finish more quickly, getting out of the way of the other, whose disk accesses are then more efficient. This does not happen for Agrep when Gnuld runs because even when



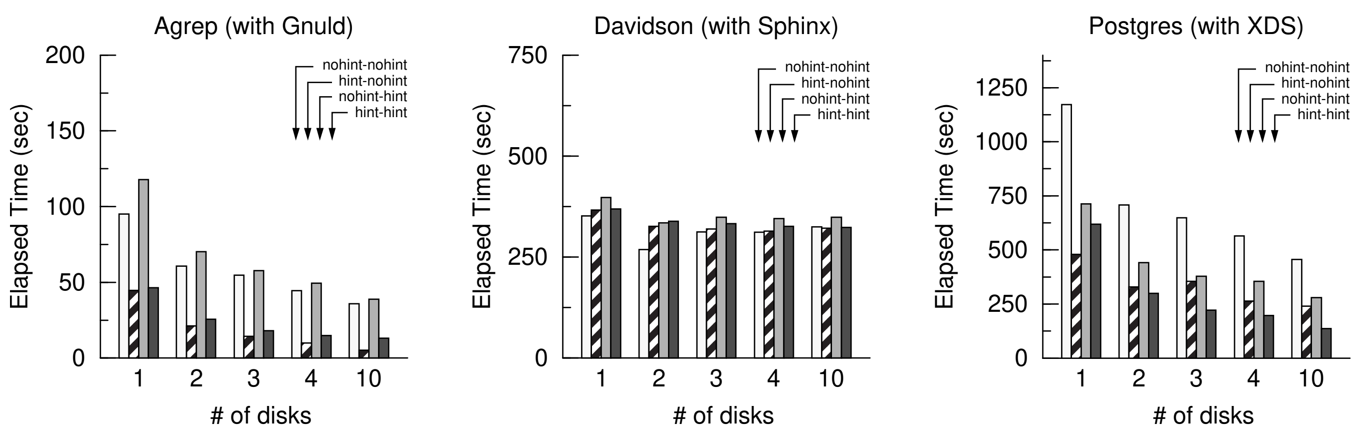
(a) (b) (c)

Figure 14. Elapsed time for both applications to complete. Three pairs of multiprogrammed workloads, (a) Gnuld and Agrep, (b) Sphinx and Davidson, and (c) XDataSlice and Postgres (80% of outer tuples match), are run on TIP-2 in parallel and the elapsed time of the last to complete is reported along with the total CPU busy time. For each number of disks, four bars are shown. These represent the four hint/nohint cases. For example, the second bar from the left in any quartet of (a) is Gnuld hinting and Agrep not hinting.



(a) (b) (c)

Figure 15. Elapsed time for one of a pair of applications. These figures report data taken from the same runs on TIP-2 as reported in Figure 14. However, the elapsed time shown represents only the named application's execution. The hint/nohint combinations are identical to Figure 14. Compare bars one and two or three and four to see the impact of giving hints when the other application is respectively non-hinting or hinting. Compare bars one and three or two and four to see the impact of the second application giving hints.



(a) (b) (c)

Figure 16. Elapsed time for the other of a pair of applications. These figures report data from the same set of runs as reported in Figures 14 and 15. However, the inner two bars are swapped relative to the inner two bars of the other figures. For example, the second bar from the left in any quartet of (a) is Gnuld not hinting and Agrep hinting. Compare bars one and two or three and four to see the impact of giving hints when the other application is respectively non-hinting or hinting. Compare bars one and three or two and four to see the impact of the second application giving hints.

Gnuld gives hints, it runs longer than Agrep and so never gets out of the way.

9 Future work

Together, informed caching and informed prefetching provide a powerful resource management scheme that takes advantage of available storage concurrency and adapts to an application's use of buffers.

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.

When all accessed devices have the same average access time, as in our experiments, blocks should be prefetched in the order they will be accessed [Cao95]. However, in the general case, some data is on a local disk while other data may be on the far side of a network. For the remote blocks, $T_{network} + T_{server} + T_{disk}$ could 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. This has far-reaching implications for informed device scheduling, the third and unaddressed point of leverage for hints based on disclosure.

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.

10 Conclusions

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 proactive resource managers can expose workload parallelism to exploit storage parallelism, 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 six I/O-intensive applications executing on a Digital 3000/500 with an array of 10 disks. Our applications include text search, data visualization, database join, speech recognition, object linking, and computational physics. With the exception of computational physics, none of these applications, without hints, exploits the parallelism of a disk array well. Informed prefetching with at least four disks reduces the elapsed time of the other five applications by 20% to 85%. For the computational physics application, 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 42%. Finally, our experimental multiprogramming results show that introducing hints always increases throughput.

Instructions for obtaining access to the code in our TIP prototype can be found in our Internet World Wide Web pages: http://www.cs.cmu.edu/afs/cs/Web/Groups/PDL.

11 Acknowledgments

We wish to thank a number of people who contributed to this work including: Charlotte Fischer and the Atomic Structure Calculation Group in the Department of Computer Science at Vanderbilt University for help with the Davidson algorithm; Ravi Mosur and the Sphinx group 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; LeAnn Neal for help with words and graphics; M. Satyanarayanan for his early contributions to our ideas; and the rest of the members of the Parallel Data Library for their support during this work.

12 References

[Baker91] Baker, M.G., Hartman, J.H., Kupfer, M.D., Shirriff, K.W., Ousterhout, J.K., "Measurements of a Distributed File System," *Proc. of the 13th Symp. on Operating System Principles*, Pacific Grove, CA, Oct. 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, Nov., 1994, pp.165-178.

[Cao95a] Cao, P., Felten, E.W., Karlin, A., Li, K., "A Study of Integrated Prefetching and Caching Strategies," *Proc. of the Joint Int. Conf. on Measurement & Modeling of Computer Systems (SIGMETRICS)*, Ottawa, Canada, May, 1995, pp. 188-197.

[Cao95b] Cao, P., Felten, E.W., Karlin, A., Li, K., "Implementation and Performance of Integrated Application-Controlled Caching, Prefetching and Disk Scheduling," *Computer Science Technical Report No. TR-CS-95-493,* Princeton University, 1995.

[Chen93] Chen, C-M.M., Roussopoulos, N., "Adaptive Database Buffer Allocation Using Query Feedback," *Proc. of the 19th Int. Conf. on Very Large Data Bases,* 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*, Dec. 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 Technical Report No. TR-91-14*, University of Virginia, 1991.

[Korner90] Korner, K., “Intelligent Caching for Remote File Service, *Proc. of the 10th 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 1st USENIX Symp. on Operating Systems Design and Implementation*, Monterey, CA, Nov. 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), V 38 (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), Aug. 1984, pp. 181-197.

[NCSA89] National Center for Supercomputing Applications. “XDataSlice for the X Window System,” http://www.ncsa.uiuc.edu/, Univ. of Illinois at Urbana-Champaign, 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., 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, Dec. 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, Jun. 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), Apr. 1993, pp. 21-34.

[Patterson94] Patterson, R.H., Gibson, G., “Exposing I/O Concurrency with Informed Prefetching,” *Proc. of the 3rd Int. Conf. on Parallel and Distributed Information Systems*, Austin, TX, Sept. 28-30, 1994, pp. 7-16.

[Rosenblum91] Rosenblum, M., Ousterhout, J.K., “The Design and Implementation of a Log-Structured File System,” *Proc. of the 13th Symp. on Operating System Principles*, Pacific Grove, CA, Oct. 1991, pp. 1-15.

[Sacco82] Sacco, G.M., Schkolnick, M., “A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model,” *Proc. of the 8th Int. Conf. on Very Large Data Bases*, Sep. 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), Aug. 1985, pp. 161-203.

[Solworth90] Solworth, J.A., Orji, C.U., “Write-Only Disk Caches,” *Proc. of the 1990 ACM Int. Conf. on Management of Data (SIGMOD)*, pp. 123-132.

[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.

[Steere95]Steere, D., Satyanarayanan, M., “Using Dynamic Sets to Overcome High I/O Latencies during Search,” *Proc. of the 5th Workshop on Hot Topics in Operating Systems*, Orcas Island, WA, May 4-5, 1995, pp. 136-140.

[Stonebraker86] Stonebraker, M., Rowe, L, “The Design of Postgres,” *Proc. of 1986 ACM Int. Conf. on Management of Data (SIGMOD)*, Washington, DC, USA, 28-30 May 1986.

[Stonebraker90] Stonebraker, M., Rowe, L.A., Hirohama, M., “The implementation of POSTGRES,” *IEEE Trans. on Knowledge and Data Engineering*, V 2 (1), Mar. 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 1992 Winter USENIX Conference*, San Francisco, CA, Jan. 1992, pp. 20-24.