Appears in *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems* (Sigmetrics '97), Seattle, Washington, June 15–18, 1997.

Informed Multi-Process Prefetching and Caching

Andrew Tomkins, R. Hugo Patterson and Garth Gibson
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891
{andrewt,rhp,garth}@cs.cmu.edu*
http://www.cs.cmu.edu/Groups/PDL

Abstract

Informed prefetching and caching based on application disclosure of future I/O accesses (hints) can dramatically reduce the execution time of I/O-intensive applications. A recent study showed that, in the context of a single hinting application, prefetching and caching algorithms should adapt to the dynamic load on the disks to obtain the best performance. In this paper, we show how to incorporate adaptivity to disk load into the TIP2 system, which uses *cost-benefit analysis* to allocate global resources among multiple processes. We compare the resulting system, which we call TIPTOE (TIP with Temporal Overload Estimators) to Cao et al's LRU-SP allocation scheme, also modified to include adaptive prefetching. Using disk-accurate trace-driven simulation we show that, averaged over eleven experiments involving pairs of hinting applications, and with data striped over one to ten disks, TIPTOE delivers 7% lower execution time than LRU-SP. Where the computation and I/O demands of each experiment are closely matched, in a two-disk array, TIPTOE delivers 18% lower execution time.

1 Introduction

Storage parallelism in the form of striping device drivers and disk arrays provides increased I/O bandwidth to match increasing processor performance. Unfortunately many workloads are composed of streams of computation interspersed with synchronous I/O calls employing only one disk of an array at a time. A powerful mechanism for overcoming this problem is system-managed prefetching and file cache management based on application disclosure[1] of future I/O accesses, which has been shown to reduce the elapsed time of I/O-intensive applications by up to 83% [PGG+95, CFL94b, CFKL95, PG94]. There are a variety of ways to obtain these disclosures. For the programmer, it is both easier and more effective to disclose future accesses than to implement application prefetching by asynchronous I/O [PGG+95]. Furthermore, recent work has shown that compilers can induce some programs to disclose their future accesses automatically [MDK96].

A system to deliver this functionality must provide solutions to two distinct problems. First, it must manage prefetching and caching within each stream of hinted accesses. Second, it must address the *allocation problem* of dividing disk and cache resources among multiple hinted and unhinted access streams.

In a recent collaboration with Kimbrel, Karlin, Cao et al [KTP+96] we addressed the first of those problems, analyzing prefetching and caching algorithms in the context of a single process disclosing all its accesses at startup. In this single-process, fully-hinted domain, prefetching and caching decisions are made without reference to global allocation; that is, each process assumes that all resources are dedicated to it. Restricted to this model, the bounded-depth prefetching algorithms of Patterson et al [PG94, PGG+95], which assume large disk arrays, may fail to prefetch deeply enough into the request stream when disk parallelism is limited or when the I/O workload is highly unbalanced across the disks of an array. On the other hand, the Aggressive algorithm of Cao et al [CFKL95], which prefetches deeply without regard to disk load, may incur substantial computational overhead by performing many unnecessary I/Os when there is ample disk bandwidth. To solve this problem our collaboration developed a new algorithm, Forestall, whose prefetch depth is based on a dynamic estimate of upcoming disk load. This algorithm performs well in the single-process fully-hinted domain on a variety of trace-driven simulation comparisons to the load-oblivious approaches mentioned above.

In this paper we study the second problem that must be addressed by a system for informed prefetching and caching: the allocation of resources among competing processes. The

*This research was supported in part by Advanced Research Projects Agency contracts DABT63-93-C-0054 and N00174-96-0002, in part by the Data Storage Systems Center under National Science Foundation grant number ECD-8907068, and in part by generous contributions from the member companies of the Parallel Data Consortium: Hewlett-Packard Laboratories, Symbios Logic Inc., Data General, Compaq, IBM Corporation, Seagate Technology, EMC Corporation, Storage Technology Corporation, and Digital Equipment Corporation. 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 supporting organization or the U.S. Government.

[1] A disclosure is a hint delivered in the language of the existing I/O interface; ie, in terms of filenames, file descriptors, and byte ranges.

TIP2 system of Patterson et al [PGG+95] demonstrated that resource allocation decisions can be made by weighing the benefit of providing resources to a consumer against the cost of taking them from a supplier. This *cost-benefit* framework provides a general, extensible method for reasoning about allocation decisions in the face of multiple processes, each offering arbitrary fractions of hinted and unhinted accesses. Essential to this cost-benefit approach is the ability to estimate benefit and cost in terms of impact on execution time.

Combining our experience with single-process fully-hinted prefetching and caching and TIP2's cost-benefit resource allocation, we present in this paper a more accurate system model for TIP2 that estimates upcoming hotspots due to load imbalance or insufficient aggregate bandwidth. This new algorithm called TIPTOE, or TIP with Temporal Overload Estimators, combines the multi-process advantages of cost-benefit allocation with the single-process advantages of dynamic, load-aware prefetching as demonstrated by Forestall.

Another strategy for resource allocation in an informed prefetching and caching system is LRU-SP, presented by Cao et al [CFL94a, Cao96], which extends traditional LRU replacement to a mechanism for selecting the buffer supplier forced to evict a cache block. It is straightforward to implement the Forestall algorithm with LRU-SP.

With TIPTOE and LRU-SP/Forestall we have comparable prefetching and caching components and dramatically different allocation strategies. This paper analyzes these two informed prefetching and caching systems, contrasting them to each other and to their predecessor systems, TIP2 [PGG+95] and LRU-SP/Aggressive [CFKL95].

For our experiments we collected traces of six I/O-intensive hinting applications (described in [PGG+95]) that range from databases to scientific computation to speech recognition. Our tracing tools capture and timestamp hints and accesses, allowing accurate modeling of the implications of late-arriving hints and unhinted accesses. Using a disk-accurate, trace-driven simulator, we compare multi-process simulations of applications under TIP2, TIPTOE, LRU-SP/Aggressive and LRU-SP/Forestall. Our primary result is that cost-benefit outperforms LRU-SP, improving execution time on average across our hinted two-process experiments by 13% relative to LRU-SP/Aggressive, 7% relative to LRU-SP/Forestall and 3% relative to TIP2. TIPTOE improves run-time by more than 25% in 14% of our fully-hinted two-process experiments relative to LRU-SP/Aggressive, 8% relative to LRU-SP/Forestall, and 1% relative to TIP2. We also consider experiments in which hinting and unhinting processes run together, and three-process experiments. The aggregate improvement over all experiments is 7.6% versus LRU-SP/Aggressive, 5.3% versus LRU-SP/Forestall and 3.6% versus TIP2. TIPTOE's primary advantage over TIP2 is that it does not suffer from the shortcomings identified in [KTP+96].

1.1 Related Work

The bulk of work on prefetching and caching has been based on inferring the future from the past starting with the seminal work that measured the effectiveness of the LRU replacement algorithm [Bel66]. Since then, sequential readahead has become a widely-used technique [FO71, MJLF84]. Others have shown how to extract much more complex access patterns from an examination of past accesses [LD97, GA95, CKV93, KE93, PZ91, TD91, Kor90]. Unfortunately, such speculative prefetching risks hurting, rather than helping, performance [Smi85]. As a result, history-based prefetching must be conservative and need not address the resource management issues related to very deep prefetching that are the focus of this paper.

Sometimes an application may have advance knowledge of resource needs and advise specific action [Tri79, SM88, CFL94b]. In contrast, we ask that applications disclose information and then allow the filesystem to make decisions in the presence of global resource knowledge. Mowry et al [MDK96] show that for some workloads, primarily scientific computing, it is possible for the compiler to generate these disclosures automatically. Others have developed richer languages for expressing and exploiting disclosure [SS95, Kot94, GJ91].

Large integrated applications may implement their own resource management. Some database researchers have shown how to use a query access plan to allocate buffers [NFS91, CR93]. 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 TIP2's cost-benefit approach to cache management.

2 Cost-Benefit Allocation and the TIPTOE Algorithm

In this section, we present TIPTOE (TIP with Temporal Overload Estimators), the major theoretical contribution of this paper. TIPTOE extends the TIP2 system within its cost-benefit framework. We give a review of this framework, reprise the assumptions made in the original TIP2, and show that when applications violate these assumptions TIP2 may miss opportunities to improve performance. These observations motivate the development of TIPTOE, which is based on a more accurate system model. We then give the details of the TIPTOE algorithm.

To summarize, TIP2's assumption of infinite disk parallelism leads to a model in which there is never any benefit from prefetching very deeply into a hint stream, and little benefit from caching deeply. In practice, there are occasions when prefetching more deeply substantially reduces elapsed time. TIPTOE quantifies the costs and benefits of deeper prefetching, allowing the allocator to trade off the resources that must be dedicated to performing a deep prefetch against the reduction in stall that will result.

2.1 Cost-Benefit Analysis

Our goal is to allocate resources to minimize *I/O service time*, the time it takes a read or write system call to complete. The managed resources are disks and file cache buffers. The consumers of these resources are demand accesses that miss in the cache and prefetches of hinted blocks. The two buffer suppliers are the traditional cache of unhinted blocks managed with the Least-Recently-Used replacement policy (the LRU cache) and the cache of blocks for which there are hints (the hinted cache).

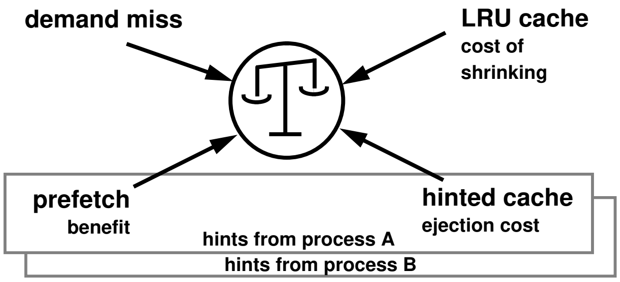


Figure 1: TIP2's informed cache manager schematic. Independent estimators express different strategies for reducing I/O service time. Demand misses need a buffer immediately to minimize the stall that has already started. Informed prefetching would like a buffer to initiate a read and avoid disk latency. To respond to these buffer requests, the buffer allocator compares their estimated benefit to the cost of taking a buffer from a buffer supplier. The LRU queue caches blocks for unhinted accesses. Informed caching holds on to the blocks that will be re-accessed soonest. The buffer allocator takes the least-valuable buffer held by any supplier to fulfill a buffer demand when the estimated benefit exceeds the estimated cost.

An I/O resource manager must decide whether reallocating a buffer from a supplier to a consumer to initiate an I/O will reduce overall I/O service time. Employing cost-benefit analysis, we estimate the benefit (decrease in I/O service time) of using a buffer to initiate a disk access and the cost (increase in I/O service time) of taking a buffer from a buffer supplier. Figure 1 gives a schematic of TIP2's cost-benefit system. 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 estimates and reallocates buffers when doing so would reduce I/O service time.

For different estimates to be comparable, they must be expressed in the same terms. We therefore define a *common currency* for the expression of cost and benefit estimates that relates the goal of reducing I/O service time to the usage of the system buffer cache resource. We define the unit of buffer usage, or *bufferage*, as the occupation of one buffer for one inter-access period and call it one *buffer-access*. Then, we define the common currency as the *magnitude of the change in I/O service time per buffer-access.*

2.1.1 TIP2 Estimators

The TIP2 cost and benefit estimates given in [PGG+95] are derived from a specific system model. The model assumes that all application I/O accesses request a single file block that can be read in a single disk access; that system parameters such as disk access latency, T_{disk}, are constants; and that there is enough disk parallelism for there never to be any congestion (that is, there is no disk queueing). In the model, T_{cpu} is the inter-access application compute time; T_{hit} is the time to read a block from the cache; and T_{driver} is the computational overhead of allocating a buffer, queueing the request at the drive, and servicing the interrupt when the disk operation completes. T_{miss}, the time to service a demand miss, is then $T_{\text{miss}} = T_{\text{hit}} + T_{\text{driver}} + T_{\text{disk}}$.

To estimate the cost of taking a buffer from the LRU cache, TIP2 maintains an estimate of the hit ratio of the cache as a function of the length, n, of the LRU queue, $H(n)$. Then, removing a buffer from a cache of size n will increase the average I/O service time for unhinted accesses by $(H(n) - H(n-1))(T_{\text{miss}} - T_{\text{hit}})$.

If a hinted block is ejected from the cache, it will have to be prefetched back in at some later time. The cost of ejecting a hinted block is the cost of prefetching the block back in. This prefetch adds a CPU overhead, T_{driver}, and possibly some stall, T_{stall}, while the prefetch completes to the I/O service time. To express the cost in terms of the common currency, the ejection estimator averages this increase in service time over the number of accesses that the ejection frees a buffer.

In estimating the benefit of using a buffer to prefetch one access more deeply, the key observation is that the application's data consumption rate is finite. Even if the application performs no computation, it must read the data from the cache which requires T_{hit} time for each block. Under the assumption of no disk queueing, a prefetch completes in time T_{disk}. In that time, an application can perform at most $\hat{P} = T_{\text{disk}}/T_{\text{hit}}$ accesses; we call $\hat{P}$ the *prefetch horizon*. Under this analysis, there is no benefit to prefetching more deeply than the prefetch horizon because prefetches initiated now for data needed more than $\hat{P}$ accesses in the future can be issued later with no additional stall time.

2.1.2 Problems with TIP2; the Forestall Algorithm

In a recent collaboration with Kimbrel, Karlin, Cao, and others, we undertook a simulation study of prefetching and caching for single processes that hint all future accesses at the start of execution [KTP+96]. This study showed that, for some applications and some disk array sizes, TIP2's assumption of unlimited disk bandwidth results in unnecessary stall.

An alternative to TIP2 that achieves less stall in some of these situations is Cao et al's Aggressive algorithm [CFKL95]. Aggressive ejects block e to prefetch block p if the disk is currently idle, p is not in memory, e is in memory, and p occurs before e in the hint stream. Essentially, the algorithm prefetches as aggressively as reasonable, subject to disk bandwidth availability.

Figure 2 shows an example situation in which TIP2 leaves a single disk idle and incurs unnecessary stall whereas Aggressive prefetches as deeply as the disk allows and incurs less stall. In this example, TIP2's assumption of infinite parallelism does not hold and prefetches do not complete in time. This phenomenon also occurs when there is an ample number of disks, but the accesses are unevenly distributed over the disks. Effectively, only a portion of the disk array is in active use and that portion does not provide enough parallelism to avoid stall.

On the other hand, the study also showed that when there is sufficient disk parallelism, prefetching too aggressively may cause unnecessary disks accesses, each of which adds a CPU overhead of T_{driver} to the application's execution. Figure 3 shows how this phenomenon may occur.

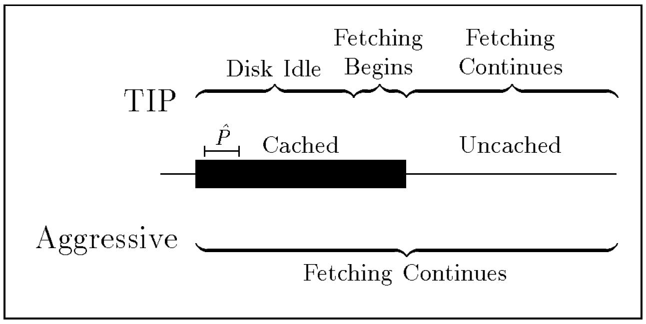


Figure 2: Lost opportunities: TIP2 prefetches a bounded number of accesses, $\hat{P}$, into the future. If a long sequence of accesses is cached, TIP2 lets the storage system go idle even if subsequent prefetching cannot satisfy the later uncached accesses and large stalls result. In contrast, Aggressive takes advantage of the lull in I/O activity during the read of the cached sequence to prefetch as many uncached blocks as possible.

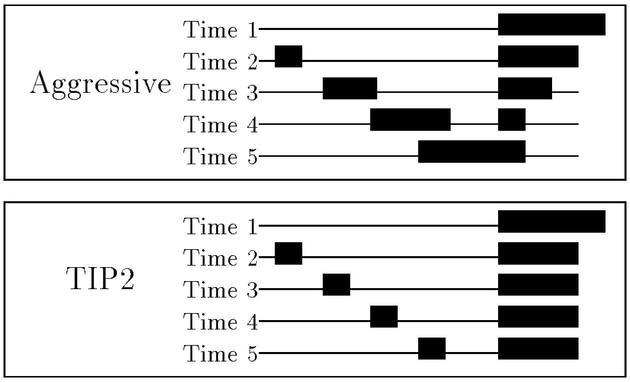


Figure 3: Wasted effort: Aggressive always ejects a cached block if it can take advantage of an idle disk to prefetch a closer block. When sufficient parallelism exists, there are often idle disks and Aggressive flushes distant, cached blocks to fill the cache with prefetched blocks. In applications with significant re-use this will incur unnecessary driver overhead by performing a disk I/O for each request, which can have a significant impact on overall execution time. In contrast, TIP2's bounded prefetching does not incur stall and retains the distantly cached blocks for re-use.

Here, TIP2's assumption of infinite parallelism is a reasonable approximation, whereas Aggressive's assumption that disk accesses have no overhead leads to increased application CPU time.

The study ultimately concluded that when stall is anticipated even far in the future, it is better to prefetch aggressively. On the other hand, when there is sufficient parallelism to avoid stall, it is better not to prefetch beyond the prefetch horizon. The Forestall algorithm presented in that paper incorporates these lessons and has performance comparable to the best of TIP2 and Aggressive. cases.

In this paper we extend the TIP2 system to incorporate these lessons into the cost-benefit framework. The result is TIPTOE.

2.2 TIPTOE

TIPTOE is built upon TIP2's cost-benefit framework, but it develops new estimators for the benefit of prefetching and the cost of ejecting hinted blocks. These new estimators no longer assume that there is infinite disk parallelism and that there is no benefit from prefetching beyond the prefetch horizon. We describe the new estimators in terms of the same system parameters used by TIP2 (T_{disk}, T_{hit} and so forth). We present the estimator for the benefit of prefetching in two distinct pieces. First, Section 2.2.1 describes how we anticipate upcoming hotspots that will lead to application stalls. Our approach follows the model of the Forestall algorithm [KTP+96]. Next, Section 2.2.2 shows how we estimate the system resources necessary to perform a deep prefetch. Finally, we combine these two values to express the benefit of deep prefetching in terms of TIP2's common currency.

2.2.1 Detecting Constrained Disks

Figure 4 is a graphical illustration of a scenario in which upcoming hotspots will cause stall. Disk a represents the ideal case: there is always enough time to prefetch the data without stalling. TIP2's bounded prefetching works well for this disk. Unfortunately, accesses may come in bursts such as the requests for blocks $b1$, $b2$ and $b3$ in the figure. Such a burst necessitates earlier prefetching that TIP2 would fail to perform. Nevertheless, the burst on disk b is small enough that we need not begin prefetching immediately; this disk is not yet *constrained*. Intuitively, a disk is constrained when there is not enough time to prefetch all missing blocks by the time they are needed (the formal definition is given below). On disk c, considering only accesses $c1$–$c4$ it appears that there is enough time to prefetch all the blocks. However, because access $c5$ comes so soon after $c4$, it is too late to avoid stalling for $c5$ even if prefetching begins immediately. Access $c5$ constrains disk c. The best we can do is to start prefetching now to minimize the stall for $c5$. Note that the access to $c6$ does not change the picture. Since the disk will be fully utilized just to minimize stall for $c5$, there will be no opportunity to prefetch deeply to reduce stall for any subsequent access. Thus, there is no reason to examine a hint sequence beyond a request that constrains the disk. On the other hand, if a disk is nearly-constrained, even a small burst of activity far in the future can constrain the disk and make it necessary to begin the entire prefetching schedule earlier in order to avoid stall.

Informally, then, the critical idea of deep prefetching is to add up the time it will take to prefetch all uncached blocks before a given request and if this time is greater then the expected time until the request arrives then the disk is constrained. Immediately initiating deep prefetching on the constrained disk can reduce application stall on the request causing the constraint, but the benefit must be compared to the cost of the resources necessary to complete the prefetch. We describe how to detect constraint, and then describe the associated benefit of deep prefetching.

Detecting constraint requires estimations of three things: the blocks that will have to be prefetched, the time it will

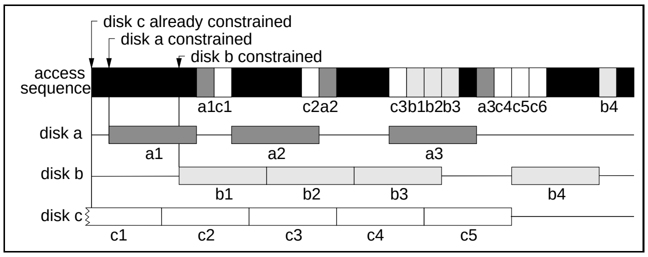


Figure 4: Constrained disks. The upper bar represents the future request sequence; black segments represent requests to cached blocks, and all other segments represent requests for missing blocks on one of the three disks. The three lower bars represent schedules for each disk that satisfy all requests in time without incurring stall. Disk a represents the ideal situation because prefetching for each block can begin one fetch time before the block will be consumed. Disk b contains a burst of requests $b1$, $b2$ and $b3$, but we do not need to begin fetching those blocks until we reach the line marked "disk b constrained." The schedule for disk c shows that in order to service all requests without stall, we would have to begin prefetching in the past, and therefore we will incur stall at some point. We say that disk c is currently the only *constrained* disk of the three.

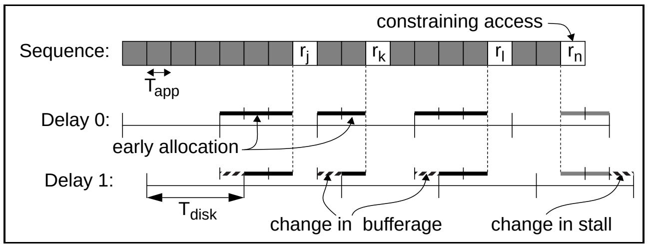


Figure 5: The Benefit of Deep Prefetching. The figure shows a hint sequence for which most accesses are cached but a few must be prefetched, the last of which, r_n, cannot be prefetched in time and therefore constrains the disk. The benefit of deep prefetching in terms of the common currency is the marginal change in stall time with respect to the change in buffer usage. Deep prefetching requires that buffers be allocated early for all of the accesses up to the constrained access. Each one-access delay in initiating deep prefetching adds one inter-access period, T_{app}, of stall, but it reduces by one access the time each of the deep-prefetching buffers must be held, under the assumption that buffers are released after they are read. If r_n is n accesses in the future, then, as shown in the text, the marginal change in stall time with respect to buffer usage is T_{disk}/n.

take to prefetch these blocks, and the time it will take the application to consume the intervening cached blocks. We determine which blocks must be prefetched based on the following simple cache model: a hinted block will be missing if it is not cached and it is the earliest hint for its block. Essentially, we make the optimistic assumption that the system will only have to prefetch each missing block once. Our estimate of T_{disk}, the time to fetch a block, is the average system I/O time up to that point in the trace. Finally, our estimate of T_{app}, the time between hinted accesses, is the average amount of computation performed by that process between hinted accesses up to that point in the trace.

With this background, we can now formally specify how the system identifies a constrained disk. Let $r_1 r_2 \ldots$ be the sequence of hints. Let $\textsc{Incore}(i)$ be a boolean variable representing our estimate, described above, of whether request i will be in cache or will need to be prefetched. Finally, let disk(i) be the disk holding request r_i. Then, disk d is constrained by request i if and only if r_i is the first reference to an uncached block and

$$\sum_{j \leq i | disk(j)=d} \textsc{Incore}(j) \cdot T_{\mathrm{disk}} > i \cdot T_{\mathrm{app}}. \qquad (1)$$

This specification of a constrained disk is very similar to that used by the Forestall algorithm in the earlier study [KTP+96], but it differs in two details. First, the $\textsc{Incore}$ function used here only counts the first access to a block as a miss instead of assuming all accesses to currently uncached blocks will have to be prefetched. Second, we do not vary our estimate of T_{disk} in a manner that that corresponds to Forestall's overestimation F', of the ratio between T_{disk} and T_{app}.

Even with our simplifying assumptions, identifying a constrained disk could still be very expensive since it could require revisiting every hint every time deep prefetching is considered. In practice, this is not necessary. We will revisit this issue after we complete the derivation of the benefit of deep prefetching in terms of the common currency.

2.2.2 The Benefit of Deep Prefetching

Having computed that access r_c constrains disk d, TIPTOE must determine the benefit of allocating buffers for deep prefetching for that constraint. Expressing this benefit in the common currency requires determining the change in stall time for a given change in buffer usage. Figure 5 gives an example that shows the relation between buffer usage and stall time. Delaying deep prefetching one access adds one inter-access period, T_{app}, of stall, but reduces by one access the time each of the deep-prefetch buffers must be held, assuming that buffers are released after being read. If the constraining access is n accesses in the future, then from the definition of a constrained disk, there must be $(nT_{\mathrm{app}})/T_{\mathrm{disk}}$ such deep prefetch buffers. Then, the marginal change in stall with respect to buffer usage is $T_{\mathrm{app}}/(nT_{\mathrm{app}}/T_{\mathrm{disk}}) = T_{\mathrm{disk}}/n$. This is the common-currency benefit of deep prefetching.

$$\Delta T_{\mathrm{deep\_pf}}(x) = \frac{T_{\mathrm{disk}}}{x}. \qquad (2)$$

2.3 The Cost of Ejecting from a Constrained Disk

Recall that ejecting a hinted block will require that the block be prefetched back at a later time. Thus, the change in I/O service time is sum of the T_{driver} overhead for prefetching the block back and any stall that will be incurred on the eventual access to the ejected block. On a constrained disk, there is no opportunity to prefetch in advance and mask the latency of the access. Thus, the access will add a full T_{disk} of stall to the application's I/O service time. If the hint indicates the block will be accessed in x accesses, then the ejection does free a buffer for the x accesses until it is needed to fault the block back in. Averaging the change in I/O service time over these x buffer-accesses of savings in buffer usage, we find that, in terms of the common currency, the cost of ejecting a block from a constrained disk is:

$$\Delta T_{\text{eject\_constrained}}(x) = \frac{T_{\text{driver}} + T_{\text{disk}}}{x}. \qquad (3)$$

Note that the cost of ejection is greater than the benefit of deep prefetching as given in Equation 2 by a T_{driver}/x term. This difference adds some hysteresis which reduces the likelihood of thrashing the same block in and out of the cache.

2.4 TIPTOE Implementation

To avoid recalculating constraints from scratch at each access, our implementation breaks the hint sequence into segments called "epochs." Each epoch keeps track of the number of accesses and missing blocks within the epoch. The contribution of the entire epoch of hints to Equation 1 can then be calculated in a single step. Whenever a block is loaded or evicted, we perform a single constant-time operation to find the earliest hint for that block, look up the associated epoch, and modify the count of missing blocks within that epoch. To detect a constraint, it is only necessary to sum the contributions of the epochs and not the hints individually. Epochs reduce algorithmic overhead by a large constant factor. In our implementation, the target size for epochs is 100 accesses, though the algorithm admits other implementations that place distant hints into larger epochs.

Because deep prefetching is only relevant over large numbers of accesses, it is not necessary to reconsider the deep prefetching decision at every access. Instead, we further reduce overhead by recomputing constraints only every 5 accesses.

With or without epochs, the computational overhead is bounded, not by the number of hints, but by the benefit of prefetching. Because the benefit of prefetching from a constrained disk falls off with the distance to the constraint, there comes a point where, even if a constraint were found, the benefit of prefetching for it would not be sufficient to warrant allocating a buffer for deep prefetching. There is no need ever to examine the hints beyond this point. The exact point at which this happens depends dynamically on the current cost of taking a buffer from a cache supplier.

TIP2 considers issuing prefetches whenever a change in cost or benefit calculations creates the chance that there might now be sufficient benefit from prefetching to merit a buffer. Most often, this occurs when the application consumes a hinted block which shifts all remaining hints one access closer. In particular, it shifts a hint from beyond the prefetch horizon to within the prefetch horizon which raises the benefit of prefetching from zero to some positive value. Thus, TIP2's prefetching is fundamentally gated by the application's consumption of data.

In contrast, deep prefetching must be disk-aware because it takes advantage of disk idleness to prefetch more deeply regardless of application activity. Thus, TIPTOE also considers prefetching whenever a disk goes idle. At that time, TIPTOE identifies the earliest missing block on the idle disk. If the hint falls within the prefetch horizon, it bids for a buffer with TIP2's benefit function. If the hint is beyond the prefetch horizon, the prefetcher bids for a buffer with the deep prefetching benefit, if any. If multiple disks are idle, it considers prefetching first for the missing block across all the idle disks that comes earliest in the hint sequence.

Should the prefetcher issue just one prefetch at a time per disk, or should it issue more? Modern disks and their low-level device drivers are capable of reordering fetches to reduce average disk service time and increase effective disk bandwidth. A prefetcher can exploit this capability by queuing multiple prefetches at the device. In general, longer queues provide greater reordering opportunities and larger reductions in average disk service time. Moreover, the same positioning effects that make disk fetch reordering effective suggest that cache evictions should be sensitive to disk location. For many workloads, especially sequential ones, proximity in the hint sequence is a good indicator of proximity on disk. In such cases, simultaneously evicting larger numbers of neighboring blocks in the hint sequence can reduce disk service time for the refetches of the evicted blocks.

A number of costs offset these benefits of simultaneously issuing large numbers of prefetches. First, large queues tie up cache buffers that might be better used to cache data for re-use. Second, filling a large queue forces earlier replacement decisions which may itself reduce cache effectiveness. Lastly, very deep queues allow early prefetches to be reordered behind many other, later prefetches which may lead to unnecessary stall.

TIPTOE strikes a balance and issues groups of up to 16 requests, a technique that Cao has called batching [CFKL95]. When TIPTOE issues prefetches, it keeps bidding for buffers for the earliest remaining uncached block until either the batch is full, or the benefit of prefetching for that disk is not great enough to win any more buffers. At that point, the prefetcher issues the batch of requests and that disk is no longer considered idle. The prefetcher continues to try to fill batches for the remaining idle disks.

3 LRU-SP Resource Allocation

Pei Cao's LRU-SP [CFL94a, CFL94b, Cao96] algorithm is an alternative to TIP2's cost-benefit strategy for allocating buffers among multiple processes in the presence of hints disclosing future accesses. The goal of the algorithm is to adapt the time-tested LRU algorithm's fairness and performance qualities to this new domain. LRU-SP uses a global LRU queue to partition the cache buffers among the competing processes. It then applies a prefetching and caching algorithm within each partition.

Figure 6 is a schematic diagram of the LRU-SP resource allocation system. When a buffer is required, either for a demand read or for a prefetch, LRU-SP finds the process that owns the Least Recently Used block of the com-

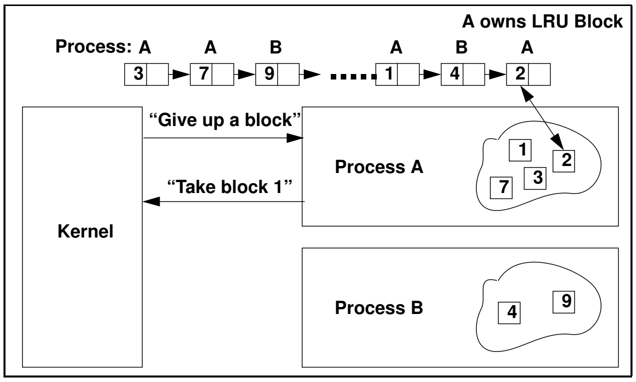


Figure 6: LRU-SP resource allocation algorithm. When the kernel needs a block it finds the process holding the global Least-Recently-Used block, in this case Process A. The kernel then asks Process A for a block, suggesting the LRU block. Process A may either accept the kernel's suggestion and give up block 2, or may use information not available to the kernel to choose a different block for eviction, in this case block 1.

plete buffer cache[2] and asks that process if it would like to eject that block. The process may choose to give up the LRU block itself, or may make a different decision based on application-specific information. If all processes give up the block suggested by the kernel then LRU-SP becomes the standard LRU buffer replacement policy. However, if a process chooses to give up an alternate block, that process will again hold the global LRU block and would be asked to give up another block when the kernel requires one. To address this difficulty the kernel swaps the LRU block into the donated block's position in the LRU queue. Finally, to prevent a malicious or foolish process from mis-using this *swapping* capability to gain an unfair share of the buffer cache, a *placeholder* structure keeps track of swaps. If the donated block is re-accessed before the swapped block, the swapped block is immediately ejected. The resulting algorithm is called LRU-SP, LRU with swapping and placeholders.

LRU-SP's partitioning of the cache makes it easy to combine different prefetching and caching algorithms with it. In the evaluations that follow, we consider two combinations. LRU-SP/Aggressive was the first combination described in the literature [CFKL95]. It uses the Aggressive deep prefetching algorithm described above to make prefetching and caching decisions within an individual process' partition. Because our recent collaboration with the inventors of LRU-SP/Aggressive found that the Forestall algorithm outperforms Aggressive [KTP+96], we also consider LRU-SP/Forestall. In the implementation of LRU-SP/Forestall considered here, we use the same algorithm for detecting constraints as is used in TIPTOE. This allows a more direct comparison between the LRU-SP and cost-benefit resource allocation algorithms.

[2] The owner of a block is the last process to access the block.

4 Methodology

Our study is based on trace-driven simulation of six hinting applications. In this section, we describe our simulator, the applications we have traced, and the trace-collection methodology.

4.1 Simulation Environment

Our simulator is built on top of the Berkeley RaidSim [CP90, LK91] simulator. RaidSim can simulate various flavors of RAID disk arrays using a disk geometry module to determine disk access times. In our simulations, we run with data striped over an array of from 1–10 disks with no parity and a stripe unit of eight 8K blocks. The geometry module simulates the performance of the HP97560 disk drive [RW94]. We use the CSCAN request scheduling discipline.

We augmented RaidSim to include a buffer cache module layered on top of the disk array and implemented modules for all four of our algorithms. We also added a module to drive RaidSim from traces instead of relying on randomly generated workloads. We took advantage of RaidSim's support for multiple threads to allow the concurrent simulation of multiple separately-scripted processes.

4.2 Applications

We drive the simulator with traces of a suite of six I/O-intensive applications. These applications were used in the evaluation of the original TIP2 system [PGG+95], and are carefully documented in that paper. We give a brief description here. The applications have all been modified to provide hints as they run, and our traces capture these hints as they arrive. The overwhelming majority of these hints are accurate. We do not explore how to prefetch when many of the hints are inaccurate.

Davidson is a computational-physics program that computes, by successive refinement, the extreme eigenvalue-eigenvector pairs of a large, sparse, matrix stored on disk [SF94]. In our benchmark, Davidson sequentially reads a 16.3 MByte matrix 61 times.

Gnuld is version 2.5.2 of the Free Software Foundation's object code linker. Our benchmark links the 562 object files of a Digital UNIX kernel. These object files comprise approximately 64 MB, and produce an 8.8MB kernel.

Postgres is version 4.2 of an extensible, object-oriented, relational database system from the University of California at Berkeley [SR86, SRH90]. 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 (32MB) and is indexed (5 MB). We run two cases. In the first (Postgres1), 20% of the outer relation finds a match in the inner relation. In the second (Postgres2), 80% find a match. One output tuple is written sequentially for every tuple match.

Sphinx is a high-quality, speaker-independent, continuous-voice, speech-recognition system [LHR90]. In our experiments, Sphinx recognizes an 18-second recording commonly used in Sphinx regression testing.

Agrep is a variant of grep written by Wu and Manber at the University of Arizona [WM92]. It is a full-text pattern matching program that performs approximate matches. In our benchmark, AGREP sequentially searches 1349 kernel source files occupying 1922 disk blocks for a simple string that does not occur in any of the files.

XDataSlice (XDS) is an interactive scientific visualization tool developed at the National Center for Supercomputer Applications at the University of Illinois [Nat89]. In our benchmark, XDS renders 25 random slices through a dataset of 512^3 32-bit floating point values requiring 512 MByte of disk storage.

4.3 Traces

The six applications (seven counting the two Postgres cases) were run on a Digital 3000/600 workstation containing a 175 MHz Alpha (21064) processor, 128 MByte of memory, and a single HP 2247 1GB disk attached via a fast SCSI-2 adapter. The Digital Unix 3.2g-3 kernel was modified to trace read, write, open, and close system calls as well as hint delivery. Trace records included a file name or vnode number and, where appropriate, the byte ranges for the call. We also traced task switches to obtain accurate, per task, CPU usage information. Traces were recorded into a 20 MByte statically-allocated buffer.

Off-line, we obtained the actual on-disk layout for files referenced in the trace so that we could map byte ranges to disk blocks. There were a very small number of temporary files whose maps we could not obtain. These files were assumed to lie at a random location on the disk, but did not account for a significant fraction of the operations. We post-processed the traces into a script that specifies the disk block or blocks touched by a request and the inter-access process CPU time.

Because our traces capture per-process CPU time and high-level file requests, we can run multiple scripts simultaneously to explore the interaction of multiple processes.

There are two effects that this simulation environment fails to capture. First, the traces do not capture either memory usage information or paging activity. When multiple processes are run together, they may compete for virtual memory causing an increase in paging activity that would not occur in our simulator. The second effect is an inaccuracy in the inter-access CPU time that arises because the traces do not record when disk access occurred. Thus, we cannot subtract the CPU time required to initiate a disk access. When the simulator initiates a disk access, it adds a disk driver overhead, T_{driver}, to the process' CPU time. This has the effect of slightly dilating the process CPU time for accesses that caused disk accesses on the original system.

Table 1 shows the balance of reads, writes and hints for each trace. All the applications are read-dominated and able to hint a substantial fraction of their total I/O's.

4.4 Single Process Performance

As a baseline for our multi-process experiments, we present the performance of our four test algorithms when applied to each of the benchmark applications individually. Figure 7 presents the results. Overall, they affirm the results of the earlier study [KTP+96].

Trace	Reads	Writes	Hints	Total
DAVIDSON	133656	1403	127429	262488
GNULD	18348	2621	14106	35075
POSTGRES1	8695	141	4455	13291
POSTGRES2	31264	522	16325	48111
SPHINX	77428	24	74700	152152
AGREP	4270	0	2921	7191
XDS	46348	0	45241	91589

Table 1: Breakdown of Traces by Operations

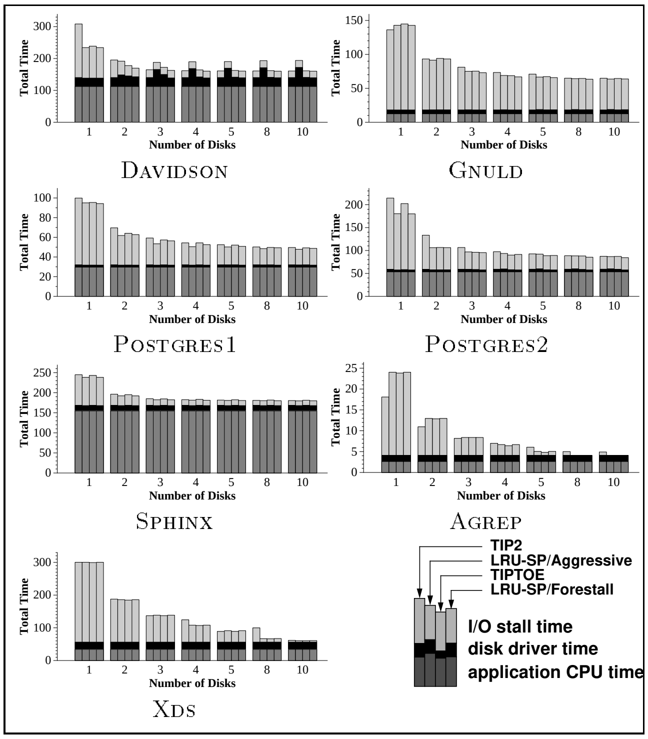


Figure 7: Single-Process Trace Results

As before, TIP2 sometimes fails to take advantage of transient disk idleness to perform deep prefetching. This is particularly visible when DAVIDSON is running on a single disk. TIP2 caches a portion of the dataset and lets the disk go idle while it reads the cached data. The effect is also visible when XDS runs on four and eight disks. Here, strided accesses result in a highly unbalanced distribution of accesses when the array size is a power of two. There are stretches of hundreds of accesses that only touch two of four disks in the array. TIP2's bounded prefetching lets the other two disks go idle even though, over the long term, they are constrained. Deep prefetching in TIPTOE and both LRU-SP algorithms relieves this problem.

We also see that on larger arrays, LRU-SP/Aggressive prefetches too aggressively for DAVIDSON, and achieves little re-use. This leads to unnecessary prefetches which add substantial CPU overhead to the elapsed time. As expected, neither TIPTOE nor LRU-SP/Forestall suffers from this problem.

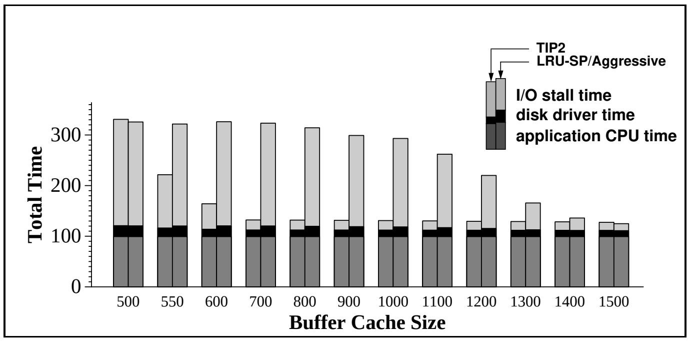


Figure 8: XDS takes buffers from a synthetic process with high re-use.

These experiments reveal a couple of other effects. Because TIP2 may have a full prefetch horizon of 63 requests queued at the disk whereas the other algorithms have a maximum of 16, TIP2 sometimes benefits more from request reordering than the others do. This effect is particularly visible when AGREP is running on one or two disks. It is visible to a lesser extent when GNULD runs on a single disk.

Finally, we note that TIP2 performs poorly for POSTGRES2 on one and two disks, and for TIPTOE on one disk. TIP2 on a single disk evicts hinted cache blocks that lie beyond the prefetch horizon, and caches 2400 fewer hinted cache blocks than LRU-SP/Aggressive. TIPTOE's hinted cache estimator (Equation 3) is not subject to this problem; it increases its estimate of the value of the hinted cache because the disk is constrained. However, its LRU hit rate estimator keeps track of accesses from the beginning of the trace, which results in an average LRU hit rate that is locally incorrect during the second phase. If the estimator is modified to allow aging of LRU hit probabilities, TIPTOE's performance on one disk is identical to the LRU-SP algorithms.

5 Multi-Process Trace Results

5.1 Synthetic Workloads and Caching

In this section we demonstrate through a micro-benchmark that LRU-based allocation may give buffers to applications that consume data quickly but have limited re-use, rather than to applications that consume more slowly but derive higher overall benefit from buffers because they have higher re-use. To first approximation, LRU-SP allocates buffers to processes according to their access rate; thus, LRU-SP will cache data for an application that has a high data consumption rate but low re-use.. We consider the XDS trace described above running in parallel with a process that repeatedly reads a 500-block file sequentially. This artificial process computes for 10ms between each read; XDS computes for 700 microseconds between reads on average. Figure 8 shows the results of varying cache size from 500 blocks to 1500 blocks.

As expected, with 500 cache buffers no algorithm derives any caching benefit because some blocks are used to prefetch for the hinting process. With 550 cache buffers, however, the cost-benefit estimators conclude that caching data for the non-hinting process is more important than prefetching ahead for the hinting process. LRU-SP splits the buffer cache according to the relative rates of the processes, giving much of the cache to the hinting process even though it does not re-use its data. The high rate of XDS relative to the non-hinting process means that the cache must become quite large (1500 buffers) before LRU-SP will allocate sufficient buffers to hold the 500 buffer working set.

Disks	TIP	LRU-SP/AGG	LRU-SP/Forestall
1	1.032	1.050	1.051
2	1.040	1.121	1.103
3	1.025	1.090	1.057
4	1.047	1.078	1.029
5	1.038	1.083	1.029
8	1.047	1.077	1.012
10	1.009	1.067	1.006
*	1.034	1.081	1.041

Table 2: Summary of results for two hinting processes. This table gives ratio of elapsed time for an algorithm to the elapsed time for TIPTOE. Numbers are the geometric mean of the ratios for the eleven experiments. The last line, marked with a '*', is the mean over all array sizes.

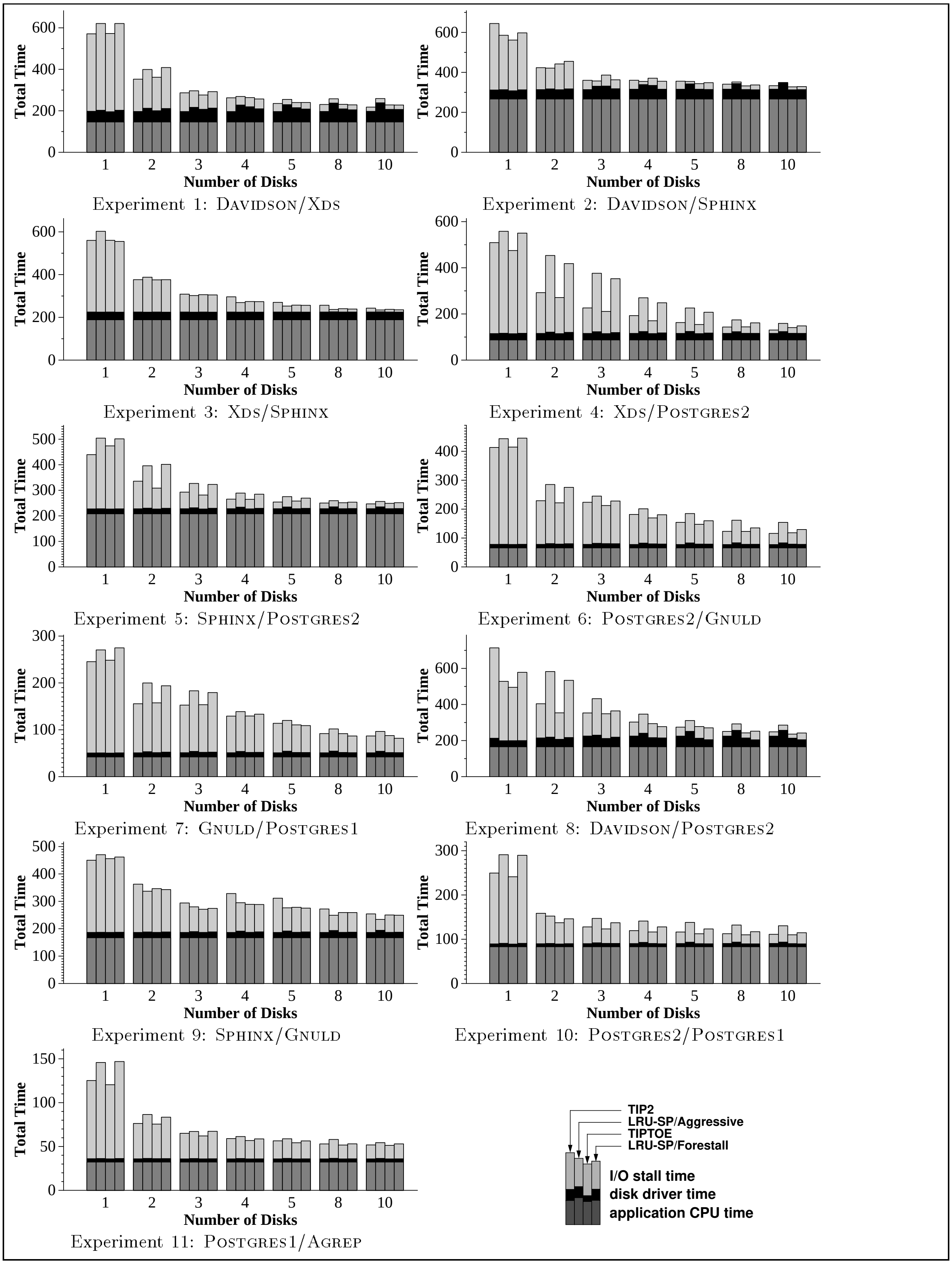
5.2 Two Hinting Processes

Our primary experimental results are given in Figure 9, which shows execution times for our test suite of eleven pairs of processes. These pairs were chosen from the $\binom{7}{2} = 21$ possible pairs based on greatest similarity in overall standalone execution time. To summarize the results, Table 2 shows, for each array size, the geometric mean of the factor by which TIPTOE performs better than each of the other algorithms, taken over the eleven experiments.

We have distilled our experience with informed caching and prefetching into a few lessons which these experimental results illustrate. The primary lesson is that LRU-SP induces a partition of the buffer cache that, to first order, depends on the relative access rates of the processes, but that relative access rate is not a good predictor of caching value. In contrast, cost-benefit partitions the cache based on estimates of the value of each piece of data. This algorithmic difference is responsible for the greatest performance differences in the graphs, and was the initial reason we undertook this study. Other lessons have significant impact, but often in particular situations such as when a disk is routinely constrained or unconstrained. Throughout the discussion we refer to experiments by the numbers in Figure 9.

5.2.1 Experience with Informed Prefetching and Caching

Lesson 1: *Access rate is not a good predictor of caching value.* Our experiments reveal two situations in which differences between data rates and re-use characteristics result in LRU-SP and cost-benefit finding different allocations. First, if a trace displays a substantial fraction of unhinted reads but consumes data slowly, LRU-SP will not dedicate buffers to LRU caching and will suffer demand misses. Cost-benefit's LRU estimator will detect re-use and publish a



Experiment 1: Davidson/Xds

Experiment 2: Davidson/Sphinx

Experiment 3: Xds/Sphinx

Experiment 4: Xds/Postgres2

Experiment 5: Sphinx/Postgres2

Experiment 6: Postgres2/Gnuld

Experiment 7: Gnuld/Postgres1

Experiment 8: Davidson/Postgres2

Experiment 9: Sphinx/Gnuld

Experiment 10: Postgres2/Postgres1

Experiment 11: Postgres1/Agrep

Figure 9: Multi-Process Trace Results

high value, reflecting the advantage of using buffers for LRU caching, and the cost-benefit allocator will grow the LRU cache in response. Second, if one hinting process shows significant re-use and another hinting process shows little re-use, but their relative access rates are not similarly disproportionate, the hinted cache of the first process will be larger under cost-benefit than under LRU-SP. Cost-benefit will send the post-consumption blocks of the low re-use process to the LRU queue where estimation will show that they are not used and can be evicted with low cost;[3] it will assign a higher value to the hinted blocks of the high re-use process, so will allow the high re-use process a larger fraction of the cache. LRU-SP will partition the cache based on process rates, giving a large fraction to the low re-use process.

The first situation, in which a process exhibits a large number of unhinted reads but does not consume data quickly, appears in experiments 4, 5, 6, 7, 10, and 11 (most notably experiment 4). The POSTGRES1 and POSTGRES2 traces, which issue large numbers of unhinted accesses, appear in all these experiments. These traces also include a phase of unhinted accesses followed by a phase of hinted accesses so the cache management algorithm must also be adaptive to this change in application behavior. In experiment 4, for instance, LRU-SP/Aggressive suffers a factor of 7.7 more demand misses then TIPTOE on two disks, and still shows 82% more demand reads on ten disks even though sufficient bandwidth exists to prefetch all hinted data without stall.

Experiment 1 demonstrates the second situation described above, in which two hinting processes compete for buffers but one process has more re-use and therefore uses buffers more effectively. Average computation time per operation is 0.83 ms for DAVIDSON and 0.72 ms for XDS. However, when each process runs alone XDS re-uses only 3.5% of its data while DAVIDSON's re-use is 38%. On two disks, LRU-SP/ Aggressive shows 43% fewer hinted cache hits than TIPTOE because it dedicates buffers to holding XDS data blocks that will not be re-used.

Lesson 2: *On unconstrained disks, hinted blocks can always be prefetched in time so caching them is not as important as caching for unhinted accesses.* Experiments 4, 5, 6, 7, 10, and 11 show an increase in stall for LRU-SP/Aggressive and, in some instances, LRU-SP/Forestall on larger arrays. None of this stall results from hinted reads that have not completed; it all results from unhinted reads, some of which could have been cached if sufficient buffer resources had been given to the LRU queue. In experiment 6 on ten disks, for instance, LRU-SP/Aggressive incurs 2.2 times as many demand misses as TIPTOE: 4303 versus 1851. These misses translate directly into stall if they cannot be overlapped against computation in another process.

Lesson 3: *On constrained disks, hinted blocks that are ejected cannot be re-fetched without stall so caching them is as important as caching for unhinted accesses.* TIP2's hinted cache estimator assumes disks are unconstrained and estimates that ejecting a hinted block beyond the prefetch horizon will only add T_{driver} overhead. TIPTOE modifies this estimator so that if the block lies on a constrained disk, the cost of ejecting the block includes the stall to re-fetch it (see Equation 3). This effect is significant in experiments 1, 2 and 8. In experiment 8 on a single disk, for instance, TIPTOE caches 37% more data blocks for DAVIDSON despite TIP2's much more conservative prefetching policy because TIPTOE values the blocks more highly.

Lesson 4: *Eviction decisions impact locality of "re-fetched" data.* Hinted cache data that is evicted must be fetched back later. A prefetching scheme may attempt to select data for eviction so as to increase disk locality when the data must be read back in. As we mentioned above, a full treatment of this topic requires a theoretical model of non-constant disk service time so a treatment within TIPTOE is beyond our scope; nonetheless, we observe the phenomenon in simulations. As described in Section 2.4, Cao et al in their presentation of Aggressive[CFKL95] describe a mechanism they call "batching" in which the prefetching algorithm waits for the disk to go idle and then submits up to B requests, where the batchsize B is a parameter of the algorithm. LRU-SP, TIPTOE and LRU-SP/Forestall all adopt this scheme in our implementation. On cyclic datasets, the B evicted elements are typically the most recently consumed blocks. Since neighboring blocks in the access stream display locality on the disk, this scheme allows the blocks to be refetched with low average disk service time. In experiment 2, for instance, TIP2's average I/O service time is 7% larger than the other algorithms on a single disk. The difference is not even larger because TIP2's conservative prefetching does not evict many blocks from the hinted cache. However, LRU-SP/Aggressive's average disk service time increases by 15% in experiment 1 when its batching mechanism is turned off and it instead submits its prefetches to the driver one at a time.

Lesson 5: *Constraint-aware prefetching only reasons about* known *constraints.* The SPHINX trace typically gives small batches of hints. There are 113 batches containing 449 hints as the program reads a large dictionary file at the beginning of the trace, then 50 other batches containing more than 64 hints, and the remainder of the batches (41% of the total batches) follow the distribution shown in Figure 10. Experiments 2 and 9 show LRU-SP/Aggressive exhibiting less stall than TIPTOE and LRU-SP/Forestall on various array sizes because the Forestall algorithms, noting that the small batch of hints received so far do not require prefetching, assume that future batches will not cause the disk to become constrained. LRU-SP/Aggressive, on the other hand, begins prefetching immediately. In experiment 9 with ten disks, for instance, TIPTOE and LRU-SP/Forestall incur 1.55 and 2.05 times as many prefetches that have not completed when the read arrives as LRU-SP/Aggressive does.

Lesson 6: *Deeper disk queues yield lower average disk service times.* Both TIP2 and TIPTOE are based upon a system model that assumes a constant disk service time, so modeling of queue sorting and locality is beyond the scope of our theoretical analysis. However our simulator performs CSCAN sorting in the queues and our disk simulator includes such non-constant effects as seek, rotate and transfer

[3] Section 5.4 shows that, if these post-consumption blocks exhibit different re-use patterns than unhinted demand read blocks, it is simple to integrate a separate "posthint estimator" into the cost-benefit framework.

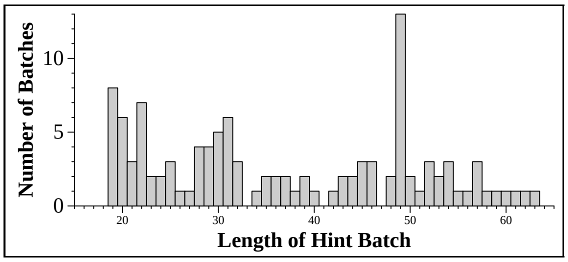


Figure 10: Distribution of hint batch sizes for the SPHINX trace.

latencies, SCSI bus overhead and on-disk readahead buffering. Therefore we exhibit effects resulting from the policy used by each prefetching algorithm to determine when exactly to submit prefetches to the disk driver. TIP2's policy is to submit prefetches out to the prefetch horizon, which in our implementation is 63; thus, TIP2 will commonly keep 63 buffers at the disk queue. The other algorithms submit up to sixteen requests whenever the disk goes idle in order to attain the benefits discussed in Lesson 4, but in doing so they typically generate shorter disk queues. Experiments 5, 7 and 9 display this effect. In experiment 5, for instance, TIP2's average disk service time is 18% faster than TIPTOE's on a single disk.

Lesson 7: *Leaving a constrained disk idle leads to additional stall.* This effect was documented in [KTP+96] for the single-process case. We discuss it in Figure 2, and mention it here because it arises in the two-process case as well. Experiments 2 and 8 both show TIP2 performing worse than TIPTOE on a single disk; in experiment 2, for instance, on a single disk TIP2 has 7% more prefetches still in progress when the corresponding read arrives than TIPTOE because TIPTOE is willing to perform deep prefetches when the disk goes idle. This effect alone is not responsible for all the difference between the two algorithms in these experiments; Lesson 3 is the primary contributor to the disparity.

Lesson 8: *Submitting an I/O requires* T_{driver} *computational overhead.* This effect was also documented in the single-process case in [KTP+96]. We discuss it in Figure 3, and it appears in experiments 1, 2, 5, 8 and 9 on larger array sizes. In experiment 1, for instance, LRU-SP/Aggressive on ten disks incurs 52% more driver overhead than TIPTOE.

Lesson 9: *Over-aggressive prefetching may result in eviction of prefetched but unread data.* LRU-SP/Aggressive prefetches deeply even when no disk is constrained. If a prefetching process is running alongside either another prefetching process or a process with significant demand reads, the prefetching process might fetch a block and then be asked to give it up to provide a block to the other process. Experiments 1, 4, 8 and 9 display this behavior; experiment 8 is the most consistent example with LRU-SP/Aggressive on five disks evicting 26% of its data before reading it. These evictions do not increase stall; in fact, LRU-SP/Aggressive stalls less waiting for prefetches to complete than any other algorithm. However, again on five disks, LRU-SP/Aggressive incurs 80% more T_{driver} overhead than TIPTOE, adding 13% to the overall execution time. The other instances within this trace, and in the other specified experiments, are less significant.

Disks	TIP	LRU-SP/AGG	LRU-SP/Forestall
1	0.982	1.001	1.003
2	1.090	1.033	1.031
3	1.063	1.038	1.029
4	1.075	1.036	1.037
5	1.042	1.045	1.038
8	1.024	1.034	1.023
10	1.008	1.034	1.018
*	1.040	1.032	1.025

Table 3: Two-process experiments, 1 process gives hints, the other does not. Average slowdown of overall execution time for other algorithms with respect to TIPTOE taken over 22 experiments.

5.3 Experiments with Unhinting Processes and More Multiprogramming

In this subsection we consider two modifications to the experiments of Section 5.2. First, we re-run each of the 11 two-process experiments with only one of the two processes giving hints. This results in 22 experiments. For TIP2, LRU-SP/Aggressive and LRU-SP/Forestall, and each array size, we compute the ratio of the algorithm's execution time to TIPTOE's execution time and take the geometric mean of these values over the 22 experiments; these results are reported in Table 3. The graphs demonstrate the same lessons described in Section 5.2, but differences of 25% in overall execution time relative to any of the other algorithms occur in only 5% of the cases, as opposed to 16% of the cases in the fully-hinted case (a case is defined as a particular experiment on a particular disk array size). As the table shows, TIPTOE performs better overall but the difference is not marked.

The second modification we consider is the addition of a third process. We construct five experiments by taking the five sets of three processes that are most similar in overall execution time. We then run each of these five experiments in all three configurations of two hinting processes and one unhinting process. The results are given in Table 4. Again, TIPTOE performs better overall, and the relative speedup is larger in this case than in the two-process case. In many of these experiments servicing of unhinted reads represents the dominant component of execution time, but in 13% of the cases there are significant differences (25% or more, as computed above) in overall execution time. In general, the addition of a third process increases the benefit seen by TIPTOE and by cost-benefit.

5.4 Cost-Benefit Analysis and Post-Consumption Hints

When a hinted read arrives and there is no future hint for the same data, the system must decide how long the block should be kept in memory. In the original TIP2 system the block was added to the tail of the LRU queue under the assumption that, since it was recently accessed, it might be

Disks	TIP	LRU-SP/AGG	LRU-SP/Forestall
1	0.998	1.038	1.040
2	1.094	1.136	1.136
3	1.055	1.155	1.132
4	1.054	1.110	1.079
5	1.033	1.114	1.074
8	1.024	1.104	1.057
10	0.997	1.097	1.036
*	1.036	1.107	1.078

Table 4: Three-process experiments, two processes give hints, one does not. Average slowdown of other algorithms with respect to TIPTOE taken over five three-process workloads run three times to consider each process not giving hints.

accessed again in the near future. However, if an XDS-like process is streaming through a large amount of hinted data with minimal re-use, and another process like POSTGRES2 is performing unhinted reads with strong locality[4] this policy will "dilute" the LRU queue with buffers that are never re-used. The opposite policy is to take lack of hints for a block as a "release" of the block, and place the block on the head of the LRU queue for immediate eviction. However under this policy a process such as SPHINX, which offers hints in small batches just before the data is required, would be prone to flush blocks that might soon be hinted.

The cost-benefit framework provides a simple, elegant solution to this problem. Rather than releasing these problematic "posthint" buffers to the LRU queue, the system instead releases them to a separate posthint queue which maintains an independent estimate of the value of its buffers. If the posthint buffers are often re-read, as in SPHINX's case, the allocator will choose to grow the posthint cache at the expense of the LRU cache. On the other hand if the posthint buffers are never accessed but unhinted accesses demonstrate re-use, as in the case of POSTGRES2 and XDS, the allocator will instead choose to dedicate resources to the LRU cache.

In general, the cost-benefit framework allows the system designer to identify subclasses of a resource that display uniform or similar patterns of behavior or re-use. The designer can then tailor estimators to each subclass, such as posthint buffers or unhinted buffers. Buffers can be members of multiple classes, and can be valued by multiple estimators,[5] and the allocator will automatically incorporate any new estimates into the global valuation described earlier.

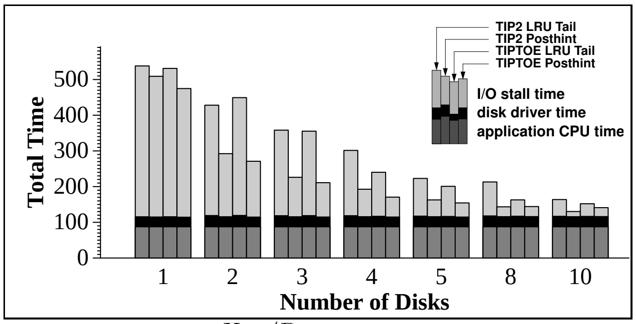
Figure 11 shows an example of the posthint estimator compared to the original TIP2 system that placed posthint buffers onto the tail of the LRU queue.

Figure 12 compares the posthint estimator to the opposite approach of releasing posthint buffers to the head of the LRU queue for immediate eviction.

Tables 5 and 6 show the aggregate results of the posthint estimator compared to the LRU-tail and LRU-head schemes.

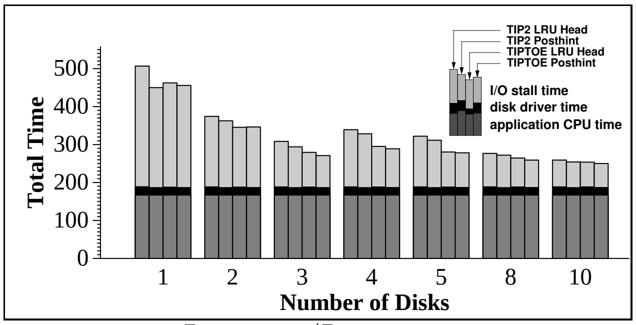
[4] The same situation could arise within a single process.

[5] For instance, a block could be read as a demand miss, placed into the LRU queue, and then have a hint arrive that says it will be read again in 300 accesses. The LRU estimator and the hinted cache estimator will independently value the buffer, and if either estimator assigns high cost the buffer will not be evicted.



XDS/POSTGRES2

Figure 11: When XDS's hinted blocks are released to the tail of the free list, they mingle with POSTGRES2's unhinted reads, reducing the LRU hit rate. When XDS's hinted blocks are instead released to the posthint estimator, the allocator grows the LRU cache at the expense of the posthint cache.



POSTGRES2/POSTGRES1

Figure 12: Both Postgres traces hint the read of the outer relation and then re-read the outer relation without giving hints for it; placing post-consumption hinted buffers onto the head of the LRU queue reduces the number of cache hits for the second set of reads.

For each experiment we compute the ratio of overall execution without the posthint estimator time to overall execution time with the posthint estimator. For each array size we then take the geometric mean of these ratios over all the experiments. We use the same seven single-process experiments and eleven multi-process experiments descried in Sections 4.4 and 5.

6 Conclusion

Systems for informed prefetching and caching based on application disclosure of future I/O accesses address two distinct issues. First, how should each hinting process use the limited resources available to it for prefetching and caching; and second, how should system resources be allocated among multiple competing processes. An earlier paper with collaborators [KTP+96] addressed the first problem and showed that prefetching algorithms that are conservative or aggressive without regard to disk load perform worse than algorithms that adapt their prefetching behavior dynamically based on load. We leverage upon these results to address the second problem by incorporating adaptive load-aware

Disks	TIP	TIPTOE
1	1.031	1.016
2	1.077	1.082
3	1.057	1.061
4	1.054	1.039
5	1.044	1.045
8	1.043	1.028
10	1.031	1.050
*	1.048	1.046

Table 5: Average improvement of releasing posthint buffers to the posthint estimator rather than to the tail of the LRU queue.

Disks	TIP	TIPTOE
1	1.086	1.011
2	1.090	1.006
3	1.085	1.008
4	1.092	1.013
5	1.100	1.018
8	1.104	1.017
10	1.096	1.036
*	1.093	1.016

Table 6: Average improvement of releasing posthint buffers to the posthint estimator rather than to the head of the LRU queue.

prefetching into the two existing systems for informed resource allocation: the TIP2 system of [PGG+95], and the LRU-SP system of [CFL94a, Cao96]. Integrating adaptive prefetching into TIP2 is significantly more challenging because it requires an estimate of the benefit of deep prefetching in terms of reduction in I/O service time per unit of buffer resource. However, we found that the resulting algorithm, TIPTOE (TIP with Temporal Overload Estimators), yields lower overall execution times than LRU-SP/Forestall because benefit estimation is a better predictor of caching value than application access rate.

We also discovered a number of more specific lessons. Our experiments confirm the conclusions of [KTP+96], and extend the scope of those results to the multi-process domain. Specifically, we found that leaving a constrained disk idle leads to additional stall and that over-aggressive prefetching from unconstrained disks leads to unnecessary I/Os and higher associated CPU overhead.

Next, we found that cache replacement decisions should be adaptive to disk load just as prefetching should be. On unconstrained disks, hinted accesses may be prefetched without stall, so it is less important to cache for them than it is to cache for unhinted accesses which stall whenever they miss in the cache. On the other hand, on constrained disks, both hinted and unhinted accesses stall so it is equally important to cache for both of them. A related lesson in the I/O-bound case is that cache replacement decisions affect the locality of blocks that will need to be re-fetched back later. Higher locality leads to lower disk service times and better performance. Incorporating sensitivity to the layout of ejected blocks into the cost-benefit framework is an area for future research.

Our experiments were performed using disk-accurate, trace-driven simulation on traces drawn from six hint-generating I/O-intensive applications described in [PGG+95]. TIPTOE performs the best of the algorithms we studied, improving execution time on average across our hinted two-process experiments by 13% relative to LRU-SP/Aggressive, 7% relative to LRU-SP/Forestall and 3% relative to TIP2. TIPTOE improves run-time by more than 25% in 14% of our fully-hinted two-process experiments relative to LRU-SP/Aggressive, 8% relative to LRU-SP/Forestall, and 1% relative to TIP2. We also consider experiments in which hinting and unhinting processes run together, and three-process experiments. The aggregate improvement over all experiments is 7.6% versus LRU-SP/Aggressive, 5.3% versus LRU-SP/Forestall and 3.6% versus TIP2. TIPTOE's primary advantage over TIP2 is that it does not suffer from the shortcomings identified in [KTP+96].

In the future, we would like to extend this work to manage virtual memory pages and prefetching over the network from distributed filesystems.

7 Acknowledgments

We thank David Rochberg for collecting and post-processing the traces, and the rest of the PDL group at CMU for providing support and cycles. Thanks also to Tracy Kimbrel, Anna Karlin and Pei Cao for contributing to our understanding of the problem. Finally, we thank the referees for many helpful suggestions.

References

[Bel66] L.A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.

[Cao96] Pei Cao. *Application-Controlled File Caching and Prefetching*. PhD thesis, Princeton University, 1996.

[CFKL95] P. Cao, E.W. Felten, A. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Proceedings of the ACM SIGMETRICS*, May, 1995.

[CFL94a] P. Cao, E.W. Felten, and K. Li. Application-controlled file caching policies. In *1994 Usenix Summer Technical Conference*, pages 171–182, June, 1994.

[CFL94b] P. Cao, E.W. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation, Monterey, CA*, pages 165–178, November, 1994.

[CKV93] K. Curewitz, P. Krishnan, and J.S. Vitter. Practical prefetching via data compression. In *Proceedings of the 1993 ACM Conference on Management of Data (SIGMOD)*, pages 257–266, May, 1993.

[CP90] Peter M. Chen and David A. Patterson. Maximizing performance in a striped disk array. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 322–331. IEEE Computer Society Press, May 1990.

[CR93] C. Chen and N. Roussopoulos. Adaptive database buffer allocation using query feedback. In *Proc. of the 19th VLDB Conference, Dublin, Ireland*, 1993.

[FO71] R. J. Feiertag and E. I. Organisk. The Multics Input/Output system. In *Proc. of the 3rd Symp. on Operating System Principles*, pages 35–41, 1971.

[GA95] J. Griffioen and R. Appleton. Performance measurements of automatic prefetching. In *Proc. of the ISCA International Conference on Parallel and Distributed Computing Systems*, September 1995.

[GJ91] A.S. Grimshaw and E.C. Loyot Jr. ELFS: Object-oriented extensible file systems. Technical Report Computer Science Technical Report No. TR-91-14, University of Virginia, 1991.

[KE93] David Kotz and Carla Schlatter Ellis. Practical prefetching techniques for multiprocessor file systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, January, 1993.

[Kor90] Kim Korner. Intelligent caching for remote file service. In *Proceedings of the 10th Intl. Conf. on Distributed Computing Systems*, pages 220–226, 1990.

[Kot94] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proc. of the 1st USENIX Symp. on Operating Systems Design and Implementation, Monterey, CA*, pages 61–74, Nov. 1994.

[KTP+96] T. Kimbrel, A. Tomkins, R.H. Patterson, B. Bershad, P. Cao, E.W. Felten, G. Gibson, A. Karlin, and K. Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 19–34, 1996.

[LD97] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. In *1997 USENIX Annual Technical Conference*, January 1997.

[LHR90] Kai-Fu Lee, Hsiao-Wuen Hon, and Raj Reddy. An overview of the SPHINX speech recognition system. *IEEE Transactions on Acoustics, Speech and Signal Processing, (USA)*, 38(1):35–45, Jan, 1990.

[LK91] Edward K. Lee and Randy H. Katz. Performance consequences of parity placement in disk arrays. In *ASPLOS4*, pages 190–199. ACM, 1991.

[MDK96] T. Mowry, A. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, 1996.

[MJLF84] M. K. McKusick, W. J. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for Unix. *ACM Trans. on Computer Systems*, 2(3):181–197, Aug. 1984.

[Nat89] National Center for Supercomputing Applications. XDataSlice for the X window system. Technical Report http://www.nsca.uiuc.edu/, University of Illinois at Urbana-Champaign, 1989.

[NFS91] Raymond Ng, Christos Faloutsos, and Timos Sellis. Flexible buffer allocation based on marginal gains. In *Proc. of the 1991 ACM Conf. on Management of Data (SIGMOD)*, pages 387–396, 1991.

[PG94] R. Hugo Patterson and Garth Gibson. Exposing I/O concurrency with informed prefetching. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 7–16, September 1994. Unpublished version in lab.

[PGG+95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 79–95, December, 1995.

[PZ91] Mark Palmer and Stanley B. Zdonik. FIDO: A cache that learns to fetch. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 255–264, September, 1991.

[RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modelling. *IEEE Computer*, 27(3):17–28, March, 1994.

[SF94] A. Stathopoulos and C. F. Fischer. A Davidson program for finding a few selected extreme eigenpairs of a large, sparse, real, symmetric matrix. *Computer Physics Communications*, 79:268–290, 1994.

[SM88] Inc. Sun Microsystems. Sun OS reference manual, part number 800-1751-10, revision A, May 9, 1988.

[Smi85] A.J. Smith. Disk cache — miss ratio analysis and design considerations. *ACM Trans. on Computer Systems*, 3(3):161–203, Aug. 1985.

[SR86] M. Stonebraker and L.A. Rowe. The design of POSTGRES. In *Proceedings of the ACM SIGMOD 1986 International Conference on Management of Data, Washington, DC*, pages 28–30, 1986.

[SRH90] M. Stonebraker, L.A. Rowe, and M. Horohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March, 1990.

[SS95] D. Steere and M. Satyanarayanan. Using Dynamic Sets to overcome high I/O latencies during search. In *Proc. of the 5th Workshop on Hot Topics in Operating Systems, Orcas Island, WA*, pages 136–140, May 4–5, 1995.

[TD91] C. Tait and D. Duchamp. Detection and exploitation of file working sets. In *Proc. Eleventh Intl. Conf. on Distributed Computing Systems*, pages 2–9, IEEE, 1991.

[Tri79] K.S. Trivedi. An analysis of prepaging. *Computing*, 22:191–210, 1979.

[WM92] S. Wu and U. Manber. Agrep — a fast approximate pattern-matching tool. In *Proc. of the 1992 Winter USENIX Conference, San Francisco, CA*, pages 20–24, Jan, 1992.