My cache or yours? Making storage more exclusive

Theodore M. Wong John Wilkes [1]

June 2002
CMU-CS-02-186

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also appears in the Proceedings of the USENIX Annual Techincal Conference, 10–15 June 2002, Monterey, CA
Supercedes CMU-CS-00-157

Abstract

Modern high-end disk arrays often have several gigabytes of cache RAM. Unfortunately, most array caches use management policies which duplicate the same data blocks at both the client and array levels of the cache hierarchy: they are *inclusive*. Thus, the aggregate cache behaves as if it was only as big as the larger of the client and array caches, instead of as large as the sum of the two. Inclusiveness is wasteful: cache RAM is expensive.

We explore the benefits of a simple scheme to achieve *exclusive caching*, in which a data block is cached at either a client or the disk array, but not both. Exclusiveness helps to create the effect of a single, large unified cache. We introduce a DEMOTE operation to transfer data ejected from the client to the array, and explore its effectiveness with simulation studies. We quantify the benefits and overheads of demotions across both synthetic and real-life workloads. The results show that we can obtain useful—sometimes substantial—speedups.

During our investigation, we also developed some new cache-insertion algorithms that show promise for multi-client systems, and report on some of their properties.

[1] Hewlett-Packard Laboratories, Palo Alto, CA, wilkes@hpl.hp.com

We thank the researchers in the Storage Systems Program at Hewlett-Packard Laboratories for their continued guidance and support of this project, which began during a summer internship there. We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Network Appliance, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. We thank IBM and Intel for hardware grants supporting our research efforts.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the respective centers, companies, or universities.

Keywords: hierarchical storage cache, exclusive caching

1 Introduction

Disk arrays use significant amounts of cache RAM to improve performance by allowing asynchronous read-ahead and write-behind, and by holding a pool of data that can be re-read quickly by clients. Since the per-gigabyte cost of RAM is much higher than of disk, cache can represent a significant portion of the cost of modern arrays. Our goal here is to see how best to exploit it.

The cache sizes needed to accomplish read-ahead and write-behind are typically tiny compared to the disk capacity of the array. Read-ahead can be efficiently handled with buffers whose size is only a few times the track size of the disks. Write-behind can be handled with buffers whose size is large enough to cover the variance (burstiness) in the write workload [32, 39], since the sustained average transfer rate is bounded by what the disks can support—everything eventually has to get to stable storage. Overwrites in the write-behind cache can increase the front-end write traffic supported by the array, but do not intrinsically increase the size of cache needed.

Unfortunately, there is no such simple bound for the size of the re-read cache: in general, the larger the cache, the greater the benefit, until some point of diminishing returns is reached. The common rule of thumb is to try to cache about 10% of the active data. Table 1 suggests that this is a luxury out of reach of even the most aggressive cache configurations if all the stored data were to be active. Fortunately, this is not usually the case: a study of UNIX file system workloads [31] showed that the mean working set over a 24 hour period was only 3–7% of the total storage capacity, and the 90th percentile working set was only 6–16%. A study of deployed HP AutoRAID systems [43] found that the working set rarely exceeded the space available for RAID1 storage (about 10% of the total storage capacity).

Both array and client re-read caches are typically operated using the *least-recently-used* (LRU) cache replacement policy [11, 12, 35]; even though many proprietary tweaks are used in array caches, the underlying algorithm is basically LRU [4]. Similar approaches are the norm in client-server file system environments [15, 27].

Interactions between the LRU policies at the client and array cause the combined caches to be *inclusive*: the array (lower-level) cache duplicates data blocks held in the client (upper-level) cache, so that the array cache is providing little re-read benefit until it exceeds the effective size of the client caches.

Inclusiveness is wasteful: it renders a chunk of the array cache similar in size to the client caches almost useless. READ operations that miss in the client are more likely to miss in the array and incur a disk access penalty. For example, suppose we have a client with 16 GB of cache memory connected to a disk array with 16 GB of re-read cache, and suppose the workload has a total READ working set size of 32 GB. (This single client, single array case is quite common in high-end computer installations; with multiple clients, the effective client cache size is equal to the amount of unique data that the clients caches hold, and the same arguments apply.) We might naïvely expect the 32 GB of available memory to capture almost all of the re-read traffic, but in practice it would capture only about half of it, because the array cache will duplicate blocks that are already in the client [15, 27].

To avoid these difficulties, it would be better to arrange for the combined client and array caches to be *exclusive*, so that data in one cache is not duplicated in the other.

1.1 Exclusive caching

Achieving exclusive caching requires that the client and array caches be managed as one. Since accesses to the client cache are essentially free, while accesses to the array cache incur the round-trip network delay, the cost of an I/O operation at the client, and the controller overheads at the array, we can think of this setup as a cache hierarchy, with the array cache at the lower level. These costs are not large: modern storage area networks (SANs) provide 1–2 Gbit/s of bandwidth per link, and I/O overheads of a few hundred microseconds; thus, retrieving a 4 KB data block can take as little as 0.2 ms.

However, it would be impractical to rewrite client O/S and array software to explicitly manage both caches. It would also be undesirable for the array to keep track of precisely which blocks are in the client, since this metadata is expensive to maintain. However, we can approximate the desired behavior by arranging that the client (1) tells the array when it changes what it caches, and (2) returns data ejected from the upper-level cache to the lower-level one, rather than simply discarding it.

We achieve the desired behavior by introducing a DEMOTE operation, which one can think of as a possible extension to the SCSI command set. DEMOTE works as follows: when a client is about to eject a clean block from its cache

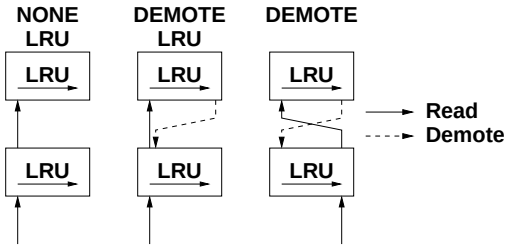

Figure 1: Sample cache management schemes. The top and bottom boxes represent the client and array cache replacement queues respectively. The arrow in a box points to the end closest to being discarded.

(e.g., to make space for a READ), it first tries to return the block to the array using a DEMOTE. A DEMOTE operation is similar to a WRITE operation: the array tries to put the demoted block into its re-read cache, ejecting another block if necessary to make space. Unlike a WRITE, the array short-circuits the operation (i.e., it does not transfer the data) if it already has a copy of the block cached, or if it cannot immediately make space for it. In all cases, the client then discards the block from its own cache.

Clients are trusted to return the same data that they read earlier. This is not a security issue, since they could easily issue a WRITE to the same block to change its contents. If corruption is considered a problem, the array could keep a cryptographic hash of the block and compare it with a hash of the demoted block, at the expense of more metadata management and execution time.

SANs are fast and disks are slow, so though a DEMOTE may incur a SAN block transfer, performance gains are still possible: even small reductions in the array cache miss rate can achieve dramatic reductions in the mean READ latency. Our goal is to evaluate how close we can get to this desirable state of affairs and the benefits we obtain from it.

1.2 Exclusive caching schemes

The addition of a DEMOTE operation does not in itself yield exclusive caching: we also need to decide what the array cache does with blocks that have just been demoted or read from disk. This is primarily a choice of cache replacement policy. We consider three combinations of demotions with different replacement policy at the array, illustrated in figure 1; all use the LRU policy at the client:

- NONE-LRU (the baseline scheme): clients do no demotions; the array uses the LRU replacement policy for both demoted and recently read blocks.
- DEMOTE-LRU: clients do demotions; the array uses the traditional LRU cache management for both demoted and recently read blocks.
- DEMOTE: clients do demotions; the array puts blocks it has sent to a client at the head (closest to being discarded end) of its LRU queue, and puts demoted blocks at the tail. This scheme most closely approximates the effect of a single unified LRU cache.

High-end arrays		
System	*Cache*	*Disk space*
EMC 8830	64 GiB	70 TB
IBM ESS	32 GiB	27 TB
HP XP512	32 GiB	92 TB
High-end servers		
System	*Memory*	*Type (CPUs)*
IBM z900	64 GiB	High-end (1–16)
Sun E10000	64 GiB	High-end (4–64)
HP Superdome	128 GiB	High-end (8–64)
HP rp8400	64 GiB	Mid-range (2–16)
HP rp7400	32 GiB	Mid-range (2–8)

Table 1: Some representative maximum-supported sizes for disk arrays and servers from early 2002. 1 GiB $= 2^{30}$ bytes.

We observe that the DEMOTE scheme is more exclusive than the DEMOTE-LRU scheme, and so should result in lower mean latencies. Consider what happens when a client READ misses in the client and array caches, and thus provokes a back-end disk read. With DEMOTE-LRU, the client and array will double-cache the block until enough subsequent READs miss and push it out of one of the caches (which will take at least as many READs as the smaller of the client and array queue lengths). With DEMOTE, the double-caching will only last only until the next READ that misses in the array cache. We thus expect DEMOTE to be more exclusive than DEMOTE-LRU, and so to result in lower mean READ latencies.

1.3 Objectives

To evaluate the performance of our exclusive caching approach, we aim to answer the following questions:

1. Do demotions increase array cache hit rates in single-client systems?
2. If so, what is the overall effect of demotions on mean latency? In particular, do the costs exceed the benefits? Costs include extra SAN transfers, as well as delays incurred by READs that wait for DEMOTEs to finish before proceeding.
3. How sensitive are the results to variations in SAN bandwidth?
4. How sensitive are the results to the relative sizes of the client and array caches?
5. Do demotions help when an array has multiple clients?

The remainder of the paper is structured as follows. We begin with a demonstration of the potential benefits of exclusive caching using some simple examples. We then explore how well it fares on more realistic workloads captured from real systems, and show that DEMOTE does indeed achieve the hoped-for benefits.

Multi-client exclusive caching represents a more challenging target, and we devote the remainder of the paper to an exploration of how this can be achieved—including a new way of thinking about cache insertion policies. After surveying related work, we end with our observations and conclusions.

2 Why exclusive caching?

In this section, we explore the *potential* benefits of exclusive caching in single-client systems, using a simple analytical performance model. We show that exclusive caching has the potential to double the effective cache size with client and array caches of equal size, and that the potential speedups merit further investigation.

We begin with a simple performance model for estimating the costs and benefits of caching. We predict the mean latency seen by a client application as

$$T_{mean} = T_c h_c + (T_a + T_c)\, h_a + \left(T_a + T_c + T_d\right) miss \tag{1}$$

where T_c and T_a are costs of a hit in the client and disk array caches respectively, T_d is the cost of reading a block from disk (since such a block is first read into the cache, and then accessed from there, it also incurs $T_a + T_c$), h_c and h_a are the client and array cache hit rates respectively (expressed as fractions of the total client READs), and $miss = 1 - (h_c + h_a)$ is the miss rate (the fraction of all READs that must access the disk). Since $T_c \approx 0$,

$$T_{mean} \approx T_a h_a + \left(T_a + T_d\right) miss \tag{2}$$

In practice, T_a is much less than T_d: $T_a \approx 0.2$ ms and $T_d \approx 4$–10 ms for non-sequential 4 KB reads.

We must also account for the cost of demotions. Large demotions will be dominated by data transfer times, small ones by array controller and host overheads. If we assume that a DEMOTE costs the same as a READ that hits in the

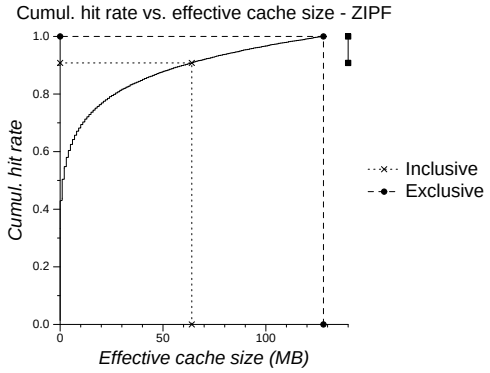

Figure 2: Cumulative hit rate *vs.* effective cache size for a Zipf-like workload, with client and array caches of 64 MB each and a working set size of 128 MB. The marker shows the additional array hit rate achieved with exclusive caching.

array, and that clients demote a block for every READ, then we can approximate the cost of demotions by doubling the latency of array hits. This is an upper bound, since demotions transfer no data if they abort, e.g., if the array already has the data cached. With the inclusion of demotion costs,

$$T_{mean} \approx 2T_a h_a + \left(2T_a + T_d\right) miss. \tag{3}$$

We now use our model to explore some simple examples, setting $T_a = 0.2$ ms and $T_d = 10$ ms throughout this section.

2.1 Random workloads

Consider first a workload with a spatially uniform distribution of requests across some working set (also known as *random*). We expect that a client large enough to hold half of the working set would achieve $h_c = 50\%$. An array with inclusive caching duplicates the client contents, and would achieve no additional hits, while an array with exclusive caching should achieve $h_a = 50\%$.

Equations 2 and 3 predict that the change from inclusive to exclusive caching would reduce the mean latency from $0.5(T_a + T_d)$ to T_a, i.e., from 5.1 ms to 0.2 ms.

2.2 Zipf workloads

Even workloads that achieve high client hit rates may benefit from exclusive caching. An example of such a workload is one with a Zipf-like distribution [49], which approximates many common access patterns: a few blocks are frequently accessed, others much less often. This is formalized as setting the probability of a READ for the i^{th} block proportional to $1/i^{\alpha}$, where α is a scaling constant commonly set to 1.

Consider the cumulative hit rate *vs.* effective cache size graph shown in figure 2 for the Zipf workload with a 128 MB working set. A client with a 64 MB cache will achieve $h_c = 91\%$. No additional hits would occur in the array with a 64 MB cache and traditional, fully inclusive caching. Exclusive caching would allow the same array to achieve an incremental $h_a = 9\%$; because $T_d >> T_a$, even small decreases in the miss rate can yield large speedups. Equations 2 and 3 predict mean READ latencies of 0.918 ms and 0.036 ms for inclusive and exclusive caching respectively—an impressive 25.5× speedup.

3 Single-client synthetic workloads

In this section, we explore the effects of exclusive caching using simulation experiments with synthetic workloads. Our goal is to confirm the intuitive arguments presented in section 2, as well as to conduct sensitivity analyses for how our demotion scheme responds to variations in the client-array SAN bandwidth and relative client and array cache sizes. Sections 4 and 5 present our results for real-life workloads.

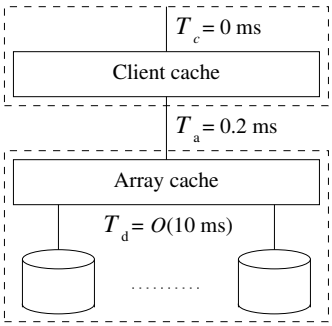

Figure 3: System simulated for the single-client workloads, with a RAID5 array and a 1 Gbit/s FibreChannel SAN.

Workload	*Client*	NONE-LRU	DEMOTE-LRU	DEMOTE
RANDOM	50%	8%	21%	46%
SEQ	0%	0%	0%	100%
ZIPF	86%	2%	4%	9%

Table 2: Client and array cache hit rates for single-client synthetic workloads. The client hit rates are the same for all the demotion variants, and can be added to the array hit rates to get the total cache hit rates.

Workload	NONE-LRU	DEMOTE-LRU	DEMOTE
RANDOM	4.77 ms	3.43 ms (1.39×)	0.64 ms (7.5×)
SEQ	1.67 ms	1.91 ms (0.87×)	0.48 ms (3.5×)
ZIPF	1.41 ms	1.19 ms (1.18×)	0.85 ms (1.7×)

Table 3: Mean READ latencies and speedups over NONE-LRU for single-client synthetic workloads.

3.1 Evaluation environment: Pantheon

To evaluate our cache management schemes, we began by using the Pantheon simulator [44], which includes calibrated disk models [33]. Although the Pantheon array models have not been explicitly calibrated, Pantheon has been used successfully in design studies of the HP AutoRAID disk array [45], so we have confidence in its predictive powers.

We configured Pantheon to model a RAID5 disk array connected to a single client over a 1 Gbit/s FibreChannel link, as shown in figure 3. For these experiments, we used a workload with 4 KB READs, and set $T_a = 0.2$ ms; the Pantheon disk models gave $T_d \approx 10$ ms.

The Pantheon cache models are extremely detailed, keeping track of I/O operations in 256 byte size units in order to model contention effects. Unfortunately, this requires large amounts of memory, and restricted us to experiments with only 64 MB caches. With a 4 KB cache block size, this means that the client and array caches were restricted to $N_c = N_a = 16384$ blocks in size.

To eliminate resource-contention effects for our synthetic workload results, we finished each READ before starting the next. In each experiment, we first "warmed up" the caches with a working-set size set of READs; the performance of these READs is not included in the results. Latency variances were all below 1%.

Our chief metric for evaluating the exclusive caching schemes is the mean latency of a READ at the client; we also report on the array cache hit rate. For each result, we present both absolute latencies and a *speedup* ratio, which is the baseline (NONE-LRU) mean latency divided by the mean latency for the current experiment. Although the difficulties of modeling partially closed-loop application behavior are considerable [16], a purely I/O-bound workload should see its execution time reduced by the speedup ratio.

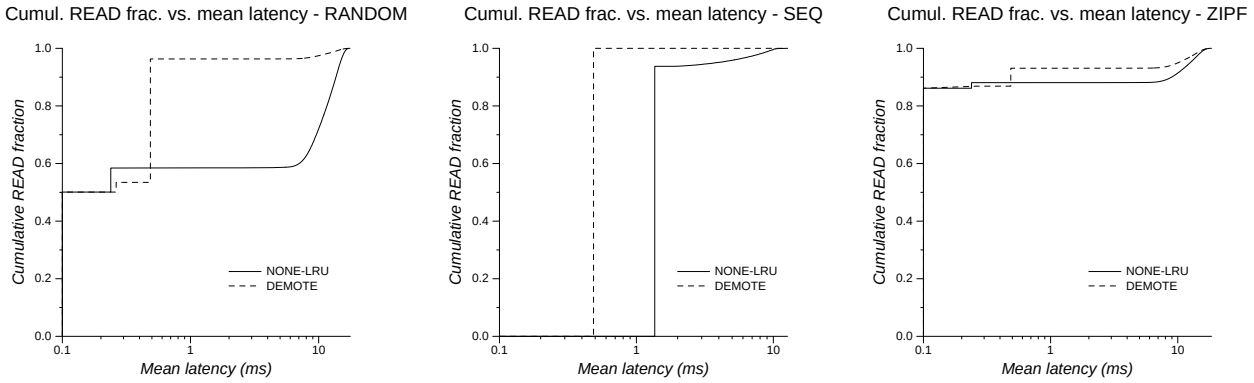

Figure 4: Cumulative READ fraction *vs.* mean READ latency for the RANDOM, SEQ, and ZIPF workloads with NONE-LRU and DEMOTE.

3.2 The RANDOM synthetic workload

For this test, the workload consisted of one-block READs uniformly selected from a working set of N_{rand} blocks. Such random access patterns are common in on-line transaction-processing workloads (e.g., TPC-C, a classic OLTP benchmark [38]).

We set the working set size to the sum of the client and array cache sizes: $N_c = N_a = 16384$, $N_{rand} = 32768$ blocks, and issued N_{rand} warm-up READs, followed by $10 \times N_{rand}$ timed READs.

We expected that the client would achieve $h_c = 50\%$. Inclusive caching would result in no cache hits at the array, while exclusive caching should achieve an additional $h_a = 50\%$, yielding a dramatic improvement in mean latency.

The results in table 2 validate our expectations. The client achieved a 50% hit rate for both inclusive and exclusive caching, and the array with DEMOTE achieved an additional 46% hit rate. 4% of READs still missed with DEMOTE, because the warm-up READs did completely fill the client cache. Also, since NONE-LRU is not fully inclusive (as previous studies demonstrate [15]), the array with NONE-LRU still achieved an 8% hit rate.

As predicted in section 1.2, DEMOTE-LRU did not perform as well as DEMOTE. DEMOTE-LRU only achieved $h_a = 21\%$, while DEMOTE achieved $h_a = 46\%$, which was a 7.5× speedup over NONE-LRU, as seen in table 3.

Figure 4 compares the cumulative latencies achieved with NONE-LRU and DEMOTE. For DEMOTE, the jump at 0.4 ms corresponds to the cost of an array hit plus the cost of a demotion. In contrast, NONE-LRU got fewer array hits (table 2), and its curve has a significantly smaller jump at 0.2 ms, which is the cost of an array cache hit without a demotion.

3.3 The SEQ synthetic workload

Sequential accesses are common in scientific, decision-support and data-mining workloads. To evaluate the benefit of exclusive caching for such workloads, we simulated READs of sequential blocks from a working set of N_{seq} contiguous blocks, chosen so that the working set would fully occupy the combined client and array caches: $N_c = N_a = 16384$, and $N_{seq} = N_c + N_a - 1 = 32767$ blocks (the -1 accounts for double-caching of the most recently read block). We issued N_{seq} warm-up READs, followed by $10 \times N_{seq}$ timed one-block READs.

We expected that at the end of the warm-up period, the client would contain the blocks in the second half of the sequence, and an array under exclusive caching would contain the blocks in the first half. Thus, with DEMOTE, all subsequent READs should hit in the array. On the other hand, with NONE-LRU and DEMOTE-LRU, we expected that the array would always contain the same blocks as the client; neither the client nor the array would have the next block in the sequence, and all READs would miss.

Again, the results in table 2 validate our expectations. Although no READs ever hit in the client, they all hit in the array with DEMOTE. The mean latency for DEMOTE-LRU was higher than for NONE-LRU because it pointlessly demoted blocks that the array discarded before they were reused. Although all READs missed in both caches with NONE-LRU and DEMOTE-LRU, the mean latencies of 1.67 ms and 1.91 ms respectively were less than the random-access disk

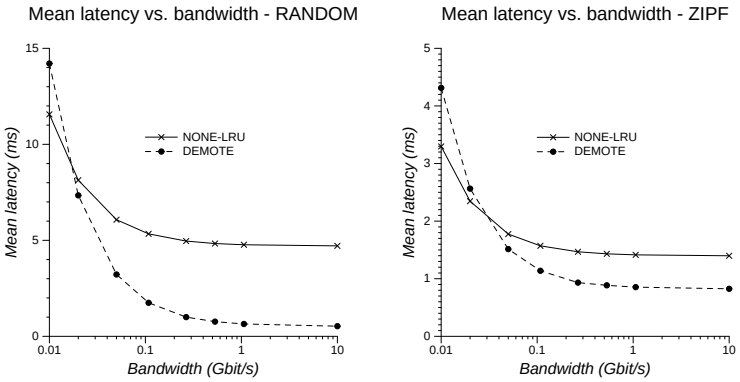

Figure 5: Mean READ latency *vs.* SAN bandwidth for the RANDOM and ZIPF workloads.

latency T_d thanks to read-ahead in the disk drive [33].

The cumulative latency graph in figure 4 further demonstrates the benefit of DEMOTE over NONE-LRU: all READs with DEMOTE had a latency of 0.4 ms (the cost of an array hit plus a demotion), while all READs with NONE-LRU had latencies between 1.03 ms (the cost of a disk access with read-ahead caching) and 10 ms (the disk latency T_d, incurred when the READ sequence wraps around). Overall, DEMOTE achieved a 3.5× speedup over NONE-LRU, as seen in table 3.

3.4 The ZIPF synthetic workload

Our Zipf workload sent READs from a set of N_{Zipf} blocks, with $N_{Zipf} = 1.5(N_c + N_a)$, so for $N_c = N_a = 16384$, $N_{Zipf} = 49152$. This resulted in three equal size sets of $N_{Zipf}/3$ blocks: Z_0 for the most active third (which received 90% of the accesses), Z_1 for the next most active (6% of the accesses), and Z_2 for the least active (the remaining 4% of the accesses). We issued N_{Zipf} warm-up READs, followed by $10 \times N_{Zipf}$ timed READs.

We expected that at the end of the warm-up set, the client cache would be mostly filled with blocks from Z_0 with the highest request probabilities, and that an array under exclusive caching would be mostly filled with the blocks from Z_1 with the next highest probabilities. With our test workload, exclusive caching schemes should thus achieve $h_c = 90\%$ and $h_a = 6\%$ in steady state. On the other hand, the more inclusive caching schemes (NONE-LRU and DEMOTE-LRU) would simply populate the array cache with the most-recently read blocks, which would be mostly from Z_0, and thus achieve a lower array hit rate h_a.

The results in table 2 validate our expectations. The client always achieved $h_c = 86\%$ (slightly lower than the anticipated 90% due to an incomplete warm-up). But there was a big difference in h_a: DEMOTE achieved 9%, while NONE-LRU achieved only 2%.

The cumulative latency graph in figure 4 supports this: as with RANDOM, the curve for DEMOTE has a much larger jump at 0.4 ms (the cost of an array hit plus a demotion) than NONE-LRU does at 0.2 ms (the cost of an array hit alone). Overall, DEMOTE achieved a 1.7× speedup over NONE-LRU, as seen in table 3. This may seem surprising given the modest increase in array hit rate, but is more readily understandable when viewed as a decrease in the overall miss rate from 12% to 5%.

3.5 SAN bandwidth sensitivity analysis

Exclusive caching using demotions relies on a low-latency, high-bandwidth SAN to allow the array cache to perform as a low-latency extension of the client cache. The more this expectation is violated (i.e., as SAN latency increases), the less benefit we expect to see—possibly to the point where demotions are not worth doing. To explore this effect, we conducted a sensitivity analysis, using Pantheon to explore the effects of varying the simulated SAN bandwidth from 10 Gbit/s to 10 Mbit/s on the NONE-LRU and DEMOTE schemes.

Our experiments validated our expectations. Figure 5 shows that at very low effective SAN bandwidths (less than 20–30 Mbit/s), NONE-LRU outperformed DEMOTE, but DEMOTE won as soon as the bandwidth rose above this threshold. The results for RANDOM and ZIPF are similar, except that the gap between the NONE-LRU and DEMOTE curves

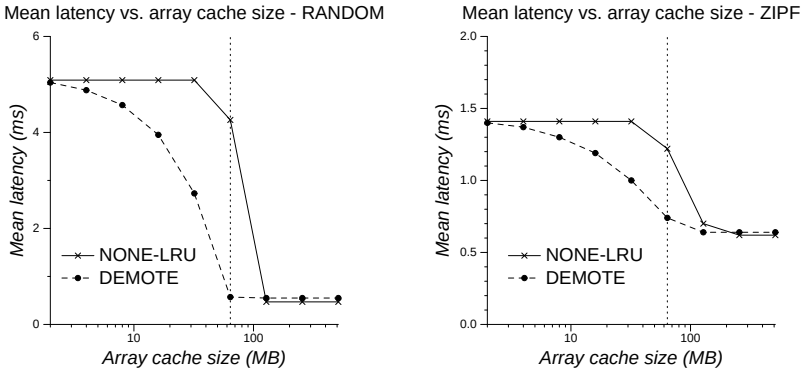

Figure 6: Mean READ latency *vs.* array cache size for the RANDOM and ZIPF workloads. The client cache size was fixed at 64 MB. The 64 MB size is marked with a dotted line.

for high-bandwidth networks is smaller for ZIPF since the increase in array hit rate (and the resultant speedup) was smaller.

3.6 Evaluation environment: `fscachesim`

For subsequent experiments, we required a simulator capable of modeling multi-gigabyte caches, which was beyond the abilities of Pantheon. To this end, we developed a simulator called `fscachesim` that only tracks the client and array cache contents, omitting detailed disk and SAN latency measurements. `fscachesim` is simpler than Pantheon, but its predictive effects for our study are similar: we repeated the experiments described in sections 3.2 and 3.4 with identical workloads, and confirmed that the client and array hit rates matched exactly. We used `fscachesim` for all the experimental work described in the remainder of this paper.

3.7 Cache size sensitivity analysis

In the results reported so far, we have assumed that the client cache is the same size as the array cache. This section reports on what happens if we relax this assumption, using a 64 MB client cache and RANDOM and ZIPF.

We expected that an array with the NONE-LRU inclusive scheme would provide no reduction in mean latency until its cache size exceeds that of the client, while one with the DEMOTE exclusive scheme would provide reductions in mean latency for any cache size until the working set fits in the aggregate of the client and array caches.

The results in figure 6 confirm our expectations. Maximum benefit occurs when the two caches are of equal size, but DEMOTE provides benefits over roughly a 10:1 ratio of cache sizes on either side of the equal-size case.

3.8 Summary

The synthetic workload results show that DEMOTE offers significant potential benefits: 1.7–7.5× speedups are hard to ignore. Better yet, these benefits are mostly insensitive to variations in SAN bandwidth and only moderately sensitive to the client:array cache size ratio.

Since our results showed that DEMOTE-LRU never outperformed DEMOTE, we did not consider it further. We also investigated schemes with different combinations of LRU and *most-recently-used* (MRU) replacement policies at the client and array in conjunction with demotions, and found that none performed as well as DEMOTE.

4 Single-client real-life workloads

Having demonstrated the benefits of demotion-based exclusive caching for synthetic workloads, we now evaluate its benefits for real-life workloads, in the form of traces taken from the running systems shown in table 4.

Some of the traces available to us are somewhat old, and cache sizes considered impressive then are small today. Given this, we set the cache sizes in our experiments commensurate with the time-frame and scale of the system from which the traces were taken.

Workload	*Date*	*Capacity*	*Cache*	*Clients*	*Length*	*Warm-up*	*I/Os*
CELLO99	1999	300 GB	2 GB	1	1 month	1 day	61.9 M
DB2	—	5.2 GB	—	8	2.2 hours	30 min	3.7 M
HTTPD	1995	0.5 GB	—	7	24 hours	1 hr	1.1 M
OPENMAIL	1999	4260 GB	2 GB	6	1 hour	10 min	5.2 M
TPC-H	2000	2100 GB	32 GB	1	1 hour	10 min	7.0 M

Table 4: Real-life workload data, with date, storage capacity, array cache size, client count, trace duration, and I/O count. 'Warm-up' is the fraction of the trace used to pre-load the caches in our experiments. For DB2 and HTTPD, working set size instead of capacity is shown. '—' are unknown entries.

Workload	*Client*	NONE-LRU		DEMOTE	
CELLO99	54%	1%	2.34 ms	13%	1.83 ms (1.28×)
DB2	4%	0%	5.01 ms	33%	3.57 ms (1.40×)
HTTPD	86%	3%	0.53 ms	10%	0.24 ms (2.20×)

Table 5: Client and array hit rates and mean latencies for single-client real-life workloads. Client hit rates are the same for all schemes. Latencies are computed using equations 2 and 3 with $T_a = 0.2$ ms and $T_d = 5$ ms. Speedups for DEMOTE over NONE-LRU are also shown.

We used `fscachesim` to simulate a system model similar to the one in figure 3, with cache sizes scaled to reflect the data in table 4. We used equations 2 and 3 with $T_a = 0.2$ ms, $T_d = 5$ ms to convert cache hit rates into mean latency predictions. This disk latency is more aggressive than that obtained from Pantheon, to reflect the improvements in disk performance seen in the more recent systems. We further assumed that there was sufficient SAN bandwidth to avoid contention, and set the cost of an aborted demotion to 0.16 ms (the cost of SAN controller overheads without an actual data transfer).

As before, our chief metric of evaluation is the improvement in the mean latency of a READ achieved by demotion-based exclusive caching schemes.

4.1 The CELLO99 real-life workload

The CELLO99 workload comprises a trace of every disk I/O access for the month of April 1999 from an HP 9000 K570 server with 4 CPUs, about 2 GB of main memory, two HP AutoRAID arrays and 18 directly connected disk drives. The system ran a general time-sharing load under HP-UX 10.20; it is the successor to the CELLO system Ruemmler and Wilkes describe in their analysis of UNIX disk access patterns [32]. In our experiments, we simulated 2 GB client and array caches.

Figure 7 suggests that that switching from inclusive to exclusive caching, with the consequent doubling of effective cache size from 2 GB to 4 GB, should yield a noticeable increase in array hit rate. The results shown in table 5 demonstrate this: using DEMOTE achieved $h_a = 13\%$ (compared to $h_a = 1\%$ with NONE-LRU), yielding a 1.28× speedup—solely from changing the way the array cache is managed.

4.2 The DB2 real-life workload

The DB2 trace-based workload was generated by an eight-node IBM SP2 system running an IBM DB2 database application that performed join, set and aggregation operations on a 5.2 GB data set. Uysal *et al.* used this trace in their study of I/O on parallel machines [40].

The eight client nodes accessed disjoint sections of the database; for the single-client workload experiment we combined all these access streams into one.

DB2 exhibits a behavior between the sequential and random workload styles seen in the SEQ and RANDOM synthetic workloads. The graph for DB2 in figure 7 suggests that a single 4 GB cache would achieve about a 37% hit rate, but that a split cache with 2 GB at each of the client and array would achieve almost no hits at all with inclusive caching; thus, DEMOTE should do much better than NONE-LRU. The results shown in table 5 bear this out: DEMOTE achieved a 33% array hit rate, and a 1.40× speedup over NONE-LRU.

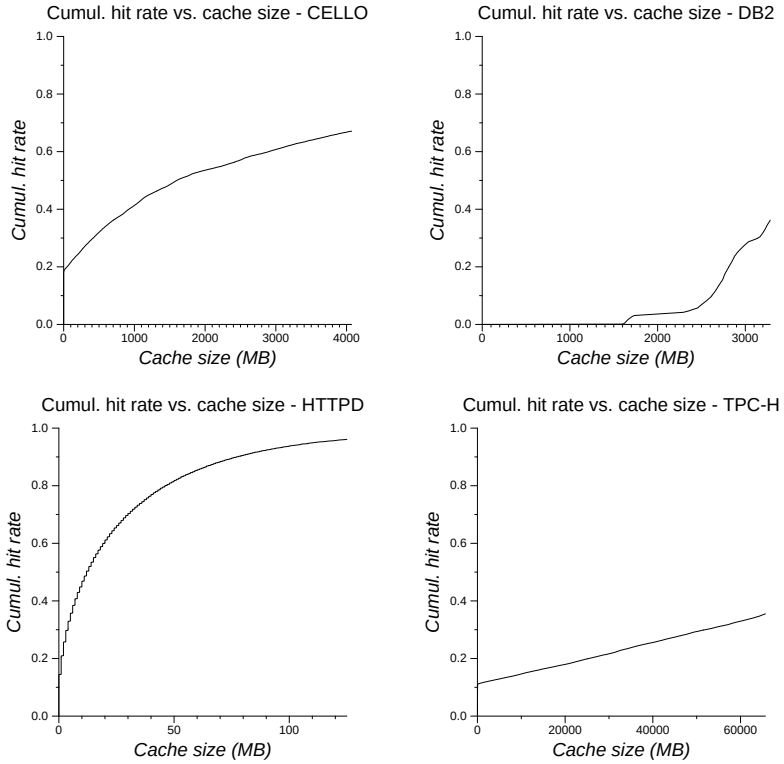

Figure 7: Cumulative hit rate *vs.* cache size graphs for single-client real-life workloads.

4.3 The HTTPD real-life workload

The HTTPD workload was generated by a seven-node IBM SP2 parallel web server [22] serving a 524 MB data set. Uysal *et al.* also used this trace in their study [40]. Again, we combined the client streams into one.

HTTPD has similar characteristics to ZIPF. A single 256 MB cache would hold the entire active working set; we elected to perform the experiment with 128 MB of cache split equally between the client and the array in order to obtain more interesting results. An aggregate cache of this size should achieve $h_c + h_a \approx 95\%$ according to the graph in figure 7, with the client achieving $h_c \approx 85\%$, and an array under exclusive caching the remaining $h_a \approx 10\%$.

Table 5 shows that the expected benefit indeed occurs: DEMOTE achieved a 10% array hit rate, and an impressive 2.2× speedup over NONE-LRU.

4.4 The TPC-H real-life workload

The TPC-H workload is a 1-hour portion of a 39-hour trace of a system that performed an audited run [18] of the TPC-H database benchmark [37]. This system illustrates high-end commercial decision-support systems: it comprised an 8-CPU (550MHz PA-RISC) HP 9000 N4000 server with 32 GB of main memory and 2.1 TB of storage capacity, on 124 disks spread across 3 arrays (with 1.6 GB of aggregate cache) and 4 non-redundant disk trays. The host computer was already at its maximum-memory configuration in these tests, so adding additional host memory was not an option. Given that this was a decision-support system, we expected to find a great deal of sequential traffic, and relatively little cache reuse. Our expectations are borne out by the results.

In our TPC-H experiments, we used a 16 KB block size, a 32 GB client cache, and a 2 GB array cache as the baseline, and explored the effects of changing the array cache size up to 32 GB. Table 6 shows the results.

The traditional, inclusive caching scheme showed no improvement in latency until the array cache size reached 32 GB, at which point we saw a tiny (1%) improvement.

With a 2 GB array cache, DEMOTE yielded a slight slowdown (0.97× speedup), because it paid the cost of doing demotions without increasing the array cache hit rate significantly. However, DEMOTE obtained a 1.04× speedup at

Array size	*Client*	NONE-LRU		DEMOTE	
2 GB	23%	0%	4.01 ms	1%	4.13 ms (0.97×)
16 GB	23%	0%	4.01 ms (1.00×)	6%	3.86 ms (1.04×)
32 GB	23%	1%	3.97 ms (1.01×)	13%	3.54 ms (1.13×)

Table 6: Client and array hit rates and mean latencies for single-client TPC-H for different array caches. Client hit rates and cache sizes (32 GB) are the same for all schemes. Latencies are computed using equations 2 and 3 with $T_a = 0.2$ ms and $T_d = 5$ ms. Speedups are with respect to a 2 GB array cache with NONE-LRU.

16 GB, and a 1.13× speedup at 32 GB, while the inclusive caching scheme showed no benefits. This data confirms that cache reuse was not a major factor in this workload, but indicates that the exclusive caching scheme took advantage of what reuse there was.

4.5 Summary

The results from real-life workloads support our earlier conclusions: apart from the TPC-H baseline, which experienced a small 0.97× slowdown due to the cost of non-beneficial demotions, we achieved up to a 2.20× speedup.

We find these results quite gratifying, given that extensive previous research on cache systems enthusiastically reports performance improvements of a few percent (e.g., a ∼1.12× speedup).

5 Multi-client systems

Multi-client systems introduce a new complication: the sharing of data between clients. Note that we are deliberately not trying to achieve client-memory sharing, in the style of protocols such as GMS [13, 42]. One benefit is that our scheme does not need to maintain a directory of which clients are caching which blocks.

Having multiple clients cache the same block does not itself raise problems (we assume that the clients wish to access the data, or they would not have read it), but exploiting the array cache as a shared resource does: it may no longer be a good idea to discard a recently read block from the array cache as soon as it has been sent to a client. To help reason about this, we consider two boundary cases here. Of course, real workloads show behavior between these extremes.

Disjoint workloads: The clients each issue READs for non-overlapped parts of the aggregate working set. The READs appear to the array as if one client had issued them, from a cache as large as the aggregate of the client caches. To determine if exclusive caching will help, we use the cumulative hit rate *vs.* cache size graph to estimate the array hit rate as if a single client had issued all READs, as in section 2.

Conjoint workloads: The clients issue exactly the same READ requests in the same order at the exact same time. If we arbitrarily designate the first client to issue an I/O as the leader, and the others as followers, we see that READs that hit in the leader also will hit in the followers. The READs appear to the array as if one client had issued them from a cache as large as an individual client cache.

To determine if the leader will benefit from exclusive caching, we use the cumulative hit rate *vs.* cache size graph to estimate the array hit rate as if the leader had issued all READs, as in section 2.

To determine if the followers will benefit from exclusive caching, we observe that all READs that miss for the leader in the array will also cause the followers to stall, waiting for that block to be read into the array cache. As soon as it arrives there, it will be sent to the leader, and then all the followers, before it is discarded. That is, the followers will see the same performance as the leader.

In systems that employ demotion, the followers waste time demoting blocks that the leader has already demoted. Fortunately, these demotions will be relatively cheap because they need not transfer any data.

5.1 Adaptive cache insertion policies

Our initial results using the simple demotion-based exclusive caching scheme described above to multi-client systems were mixed. At first, we evaluated NONE-LRU and DEMOTE in a multi-client system similar to the one shown in figure 3, with the single client shown in that figure simply replaced by N clients, each with $1/N$ of the cache memory of the single client. As expected, workloads in which clients shared few or no blocks (disjoint workloads) benefitted from DEMOTE.

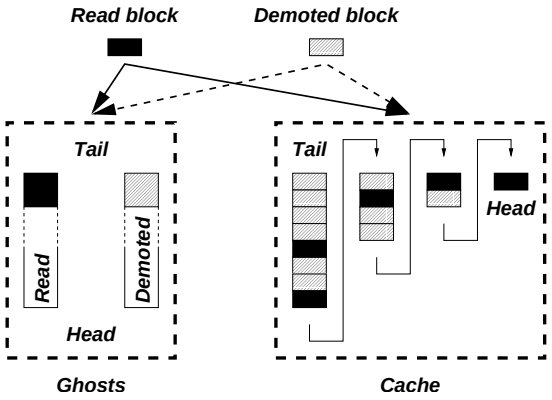

Figure 8: Operation of read and demoted ghost caches in conjunction with the array cache. The array inserts the metadata of incoming read (demoted) blocks into the corresponding ghost, and the data into the cache. The cache is divided into segments of either uniform or exponentially-growing size. The array selects the segment into which to insert the incoming read (demoted) block based on the hit count in the corresponding ghost.

Unfortunately, workloads in which clients shared blocks performed worse with DEMOTE than with NONE-LRU, because shared workloads are not conjoint in practice: clients do not typically READ the same blocks in the same order at the same time. Instead, a READ for block X by one client may be followed by several READs for other blocks before a second READ for X by another client. Recall that with DEMOTE the array puts blocks read from disk at the head of the LRU queue, i.e., in MRU order. Thus, the array is likely to eject X before the READ from the later client.

We made an early design decision to avoid the complexities of schemes that require the array to track which clients had which blocks and request copies back from them—we wanted to keep the client-to-array interaction as simple, and as close to standard SCSI, as possible.

Our first insight was that the array should reserve a portion of its cache to keep blocks recently read from disk "for a while", in case another client requests them. To achieve this, we experimented with a segmented LRU (SLRU) array cache [21]—one with *probationary* and *protected* segments, each managed in LRU fashion. The array puts newly inserted blocks (read and demoted) at the tail of the probationary segment, and moves them to the tail of the protected segment if a subsequent READ hits them. The array moves blocks from the head of the protected segment to the tail of the probationary one, and ejects blocks from the head of the probationary segment.

SLRU improved performance somewhat, but the optimal size of the protected segment varied greatly with the workload: the best size was either very small (less than 8% of the total), or quite large (over 50%). These results were less robust than we desired.

Our second insight is that the array can treat the LRU queue as a continuum, rather than as a pair of segments: inserting a block near the head causes that block to have a shorter expected lifetime in the queue than inserting it near the tail. We can then use different insertion points for demoted blocks and disk-read blocks. (Pure DEMOTE is an extreme instance that only uses the ends of the LRU queue, and SLRU is an instance where the insertion point is a fixed distance down the LRU queue.)

Our experience with SLRU suggested that the array should select the insertion points *adaptively* in response to workload characteristics instead of selecting them statically. For example, the array should insert demoted blocks closer to the tail of its LRU queue than disk-read blocks if subsequent READs hit demoted blocks more often. To support this, we implemented *ghost caches* at the array for demoted and disk-read blocks.

A ghost cache behaves like a real cache except that it only keeps cache metadata, enabling it to simulate the behavior of a real cache using much less memory. We used a pair of ghost caches to simulate the performance of hypothetical array caches that only inserted blocks from a particular source—either demotions or disk reads. Just like the real cache, each ghost cache was updated on READs to track hits and execute its LRU policy.

We used the ghost caches to provide information about which insertion sources are the more likely to insert blocks that are productive to cache, and hence where in the real cache future insertions from this source should go, as shown in figure 8.) This was done by calculating the insertion point in the real cache from the relative hit counts of the

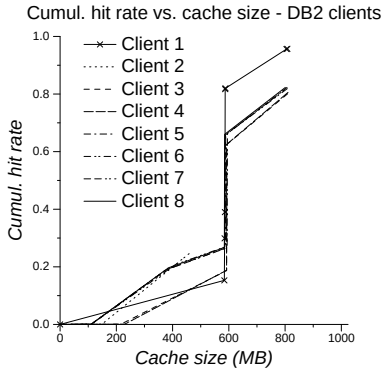

Figure 9: Cumulative hit rate *vs.* cache size for DB2 clients.

Client	1	2	3	4	5	6	7	8
	NONE-LRU							
Mean lat.	5.20	4.00	4.62	4.66	4.66	4.68	4.66	4.66
	DEMOTE (mean speedup 1.50×)							
Mean lat.	1.30	4.12	3.44	3.41	3.39	3.38	3.40	3.38
Speedup	4.00×	0.97×	1.34×	1.37×	1.38×	1.39×	1.37×	1.38×
	DEMOTE-ADAPT-UNI (mean speedup 1.27×)							
Mean lat.	2.15	4.12	3.57	3.53	4.09	4.07	4.07	4.05
Speedup	2.42×	0.97×	1.29×	1.32×	1.14×	1.15×	1.15×	1.15×
	DEMOTE-ADAPT-EXP (mean speedup 1.32×)							
Mean lat.	1.79	4.12	3.55	3.51	3.94	4.05	3.92	3.99
Speedup	2.91×	0.97×	1.30×	1.33×	1.18×	1.16×	1.19×	1.17×

Table 7: Per-client mean latencies (in ms) for multi-client DB2. Latencies are computed using equations 2 and 3 with $T_a = 0.2$ ms and $T_d = 5$ ms. Speedups over NONE-LRU, and the geometric mean of all client speedups, are also shown.

ghost caches. To do so, we assigned the value 0 to represent the head of the real array LRU queue, and the value 1 to the tail; the insertion points for demoted and disk-read blocks were given by the ratio of the hit rates seen by their respective ghost caches to the total hit rate across all ghost caches.

To make insertion at an arbitrary point more computationally tractable, we approximated this by dividing the real array LRU queue into a fixed number of segments N_{segs} (10 in our experiments), multiplying the calculated insertion point by N_{segs}, and inserting the block at the tail of that segment.

We experimented with uniform segments, and with exponential segments (each segment was twice the size of the preceding one, the smallest being at the head of the array LRU queue). The same segment-index calculation was used for both schemes, causing the scheme with segments of exponential size to give significantly shorter lifetimes to blocks predicted to be less popular.

We designated the combination of demotions with ghost caches and uniform segments at the array as DEMOTE--ADAPT-UNI, and that of demotions with ghost caches and exponential segments as DEMOTE-ADAPT-EXP. We then re-ran the experiments for which we had data for multiple clients, but separated out the individual clients.

5.2 The multi-client DB2 workload

We used the same DB2 workload described in section 4.2, but with the eight clients kept separate. Each client had a 256 MB cache, so the aggregate of client caches remained at 2 GB. The array had 2 GB of cache.

Each DB2 client accesses disjoint parts of the database. Given our qualitative analysis of disjoint workloads, and the speedup for DB2 in a single-client system with DEMOTE, we expected to obtain speedups in this multi-client system. If we assume that each client uses one eighth (256 MB) of the array cache, then each client has an aggregate of 512 MB to hold its part of the database, and we expected from figure 9 that exclusive caching would obtain a significant increase in array hit rates, with a corresponding reduction in mean latency.

Our results shown in table 7 agree: DEMOTE achieved an impressive 1.50× speedup over NONE-LRU. DEMOTE--ADAPT-UNI and DEMOTE-ADAPT-EXP achieved only 1.27–1.32× speedups, since they were more likely to keep

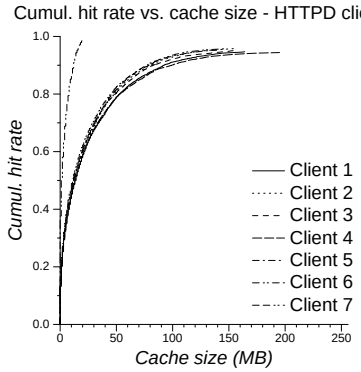

Figure 10: Cumulative hit rate *vs.* cache size for HTTPD clients.

No. clients	1	2	3	4	5	6	7
No. blocks	13173	8282	5371	5570	6934	24251	5280
% of total	19%	12%	8%	8%	10%	35%	8%

Table 8: Histogram showing the number of blocks shared by x HTTPD clients, where x ranges from 1 to 7 clients.

Client	1	2	3	4	5	6	7
	NONE-LRU						
Mean lat.	0.90	0.83	0.82	0.89	0.79	0.76	0.19
	DEMOTE (mean slowdown 0.55×)						
Mean lat.	1.50	1.41	1.44	1.48	1.43	1.33	0.46
Speedup	0.60×	0.59×	0.57×	0.60×	0.55×	0.57×	0.41×
	DEMOTE-ADAPT-UNI (mean slowdown 0.91×)						
Mean lat.	0.99	0.92	0.91	0.98	0.87	0.86	0.20
Speedup	0.91×	0.90×	0.90×	0.91×	0.90×	0.89×	0.94×
	DEMOTE-ADAPT-EXP (mean speedup 1.18×)						
Mean lat.	0.81	0.73	0.74	0.79	0.68	0.67	0.12
Speedup	1.12×	1.13×	1.10×	1.13×	1.16×	1.13×	1.52×

Table 9: Per-client mean latencies (in ms) for multi-client HTTPD. Latencies are computed using equations 2 and 3 with $T_a = 0.2$ ms and $T_d = 5$ ms. Speedups over NONE-LRU, and the geometric mean of all client speedups, are also shown.

disk-read blocks in the cache, reducing the cache available for demoted blocks, and thus making the cache less effective for this workload.

5.3 The multi-client HTTPD workload

We returned to the original HTTPD workload, and separated the original clients. We gave 8 MB to each client cache, and kept the 64 MB array cache as before.

Figure 10 indicates that the per-client workloads are somewhat similar to the ZIPF synthetic workload. As shown in section 3.4, disk-read blocks for such workloads will in general have low probabilities of being reused, while demoted blocks will have higher probabilities. On the other hand, as shown by the histogram in table 8, clients share a high proportion of blocks, and tend to exhibit conjoint workload behavior. Thus, while the array should discard disk-read blocks more quickly than demoted blocks, it should not discard them immediately.

Given this analysis, we expected DEMOTE to post less impressive results than adaptive schemes, and indeed it did, as shown in table 9: a 0.55× slowdown in mean latency over NONE-LRU. On the other hand, DEMOTE-ADAPT-EXP achieved a 1.18× speedup. DEMOTE-ADAPT-UNI achieved a 0.91× slowdown, which we attribute to demoted blocks being much more valuable than disk-read ones, but the cache with uniform segments devoting too little of its space to them compared to the one with exponential segments.

5.4 The OPENMAIL workload

The OPENMAIL workload comes from a trace of a production e-mail system running the HP OpenMail application for 25,700 users, 9,800 of whom were active during the hour-long trace. The system consisted of six HP 9000 K580

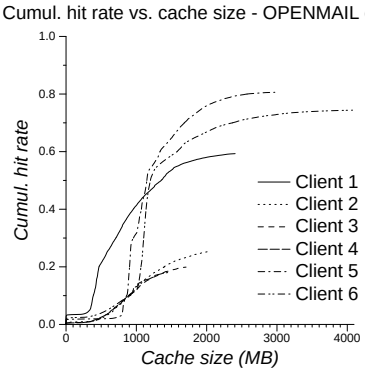

Figure 11: Cumulative hit rate *vs.* cache size for OPENMAIL.

Client	1	2	3	4	5	6
	NONE-LRU					
Mean lat.	2.96	4.52	4.54	4.47	1.79	1.78
	DEMOTE (mean speedup 1.15×)					
Mean lat.	2.32	4.08	4.27	4.35	1.27	1.67
Speedup	1.28×	1.11×	1.06×	1.03×	1.41×	1.07×
	DEMOTE-ADAPT-UNI (mean speedup 1.07×)					
Mean lat.	2.66	4.34	4.42	4.46	1.44	1.79
Speedup	1.11×	1.04×	1.03×	1.00×	1.24×	0.99×
	DEMOTE-ADAPT-EXP (mean slowdown 0.88×)					
Mean lat.	3.03	4.60	4.60	4.54	2.53	2.47
Speedup	0.97×	0.98×	0.99×	0.98×	0.71×	0.72×

Table 10: Per-client mean latencies (in ms) for OPENMAIL. Latencies are computed using equations 2 and 3 with $T_a = 0.2$ ms and $T_d = 5$ ms. Speedups over NONE-LRU, and the geometric mean of all client speedups, are also shown.

servers running HP-UX 10.20, each with 6 CPUs, 2 GB of memory, and 7 SCSI interface cards. The servers were attached to four EMC Symmetrix 3700 disk arrays. At the time of the trace, the servers were experiencing some load imbalances, and one was I/O bound.

Figure 11 suggests that 2 GB client caches would hold the entire working set for all but two clients. To obtain more interesting results, we simulated six clients with 1 GB caches connected to an array with a 6 GB cache.

OPENMAIL is a disjoint workload, and thus should obtain speedups from exclusive caching. If we assume that each client uses a sixth (1 GB) of the array cache, then each client has an aggregate of 2 GB to hold its workload, and we see from figure 11 that an array under exclusive caching array should obtain a significant increase in array cache hit rate, and a corresponding reduction in mean latency.

As with DB2, our results (table 10) bear out our expectations: DEMOTE, which aggressively discards read blocks and holds demoted blocks in the array, obtained a 1.15× speedup over NONE-LRU. DEMOTE-ADAPT-UNI and DEMOTE-ADAPT-EXP fared less well, yielding a 1.07× speedup and 0.88× slowdown respectively.

5.5 Summary

The clear benefits from single-client workloads are not so easily repeated in the multi-client case. For largely disjoint workloads, such as DB2 and OPENMAIL, the simple DEMOTE scheme does well, but it falls down when there is a large amount of data sharing. On the other hand, the adaptive demotion schemes do well when simple DEMOTE fails, which suggests that a mechanism to switch between the two may be helpful.

Overall, our results suggests that even when demotion-based schemes seem not to be ideal, it is usually possible to find a setting where performance is improved. In the enterprise environments we target, such tuning is an expected part of bringing a system into production.

6 Related work

The literature on caching in storage systems is large and rich, so we only cite a few representative samples. Much of it focuses on predicting the performance of an existing cache hierarchy [6, 24, 35, 34], describing existing I/O systems [17, 25, 39], and determining when to flush write-back data to disk [21, 26, 41]. Real workloads continue to demonstrate that read caching has considerable value in arrays, and that a small amount of non-volatile memory greatly improves write performance [32, 39].

We are not the first to have observed the drawbacks of inclusive caching. Muntz *et al.* [27, 28] show that intermediate-layer caches for file servers perform poorly, and much of the work on cache replacement algorithms is motivated by this observation [21, 24, 30, 48]. Our DEMOTE scheme, with alternative array cache replacement policies, is another such remedy.

Choosing the correct cache replacement policy in an array can improve its performance [19, 21, 30, 35, 48]. Some studies suggest using least-frequently-used [15, 46] or frequency-based [30] replacement policies instead of LRU in file servers. MRU [23] or next-block prediction [29] policies have been shown to provide better performance for sequential loads. LRU or clocking policies [10] can yield acceptable results for database loads; for example, the IBM DB2 database system [36] implements an augmented LRU-style policy.

Our DEMOTE operation can be viewed as a very simple form of a client-controlled caching policy [7], which could be implemented using the "write to cache" operation available on some arrays (e.g., those from IBM [3]). The difference is that we provide no way for the client to control which blocks the array should replace, and we trust the client to be well-behaved.

Recent studies of cooperative World Wide Web caching protocols [1, 20, 47] look at policies beyond LRU and MRU. Previously, analyses of web request traces [2, 5, 8] showed the file popularity distributions to be Zipf-like [49]. It is possible that schemes tuned for these workloads will perform as well for the sequential or random access patterns found in file system workloads, but a comprehensive evaluation of them is outside the scope of this paper. In addition, web caching, with its potentially millions of clients, is targeted at a very different environment than our work.

Peer-to-peer cooperative caching studies are relevant to our multi-client case. In the "direct client cooperation" model [9], active clients offload excess blocks onto idle peers. No inter-client sharing occurs—cooperation is simply a way to exploit otherwise unused memory. The GMS global memory management project considers finding the nodes with idle memory [13, 42]. Cooperating nodes use approximate knowledge of the global memory state to make caching and ejection decisions that benefit a page-faulting client and the whole cluster.

Perhaps the closest work to ours in spirit is a global memory management protocol developed for database management systems [14]. Here, the database server keeps a directory of pages in the aggregate cache. This directory allows the server to forward a page request from one client to another that has the data, request that a client demote rather than discard the last in-memory copy of a page, and preferentially discard pages that have already been sent to a client. We take a simpler approach: we do not track which client has what block, and thus cannot support inter-client transfers—but we need neither a directory nor major changes to the SCSI protocol. We rely on a high-speed network to perform DEMOTE eagerly (rather than first check to see if it is worthwhile) and we do not require a (potentially large) data structure at the array to keep track of what blocks are where. Lower complexity has a price: we are less able to exploit block sharing between clients.

7 Conclusion

We began our study with a simple idea: that a DEMOTE operation might make array caches more exclusive and thus achieve better hit rates. Experiments with simple synthetic workloads support this hypothesis; moreover, the benefits are reasonably resistant to reductions in SAN bandwidth and variations in array cache size. Our hypothesis is further supported by 1.04–2.20× speedups for most single-client real-life workloads we studied—and these are significantly larger than several results for other cache improvement algorithms.

The TPC-H system parameters show why making array caches more exclusive is important in large systems: cache memory for the client and arrays represented 32% of the total system cost of $1.55 million [18]. The ability to take full advantage of such large investments is a significant benefit; reducing their size is another.

Using multiple clients complicates the story, and our results are less clear-cut in such systems. Although we saw up to a 1.5× speedup with our exclusive caching schemes, we incurred a slowdown with the simple DEMOTE scheme when clients shared significant parts of the working set. Combining adaptive cache-insertion algorithms with demotions yielded improvements for these shared workloads, but penalized disjoint workloads. However, we believe that it would not be hard to develop an automatic technique to switch between these simple and adaptive modes.

In conclusion, we suggest that the DEMOTE scheme is worth consideration by system designers and I/O architects, given our generally positive results. Better yet, as SAN bandwidth and cache sizes increase, its benefits will likely increase, and not be wiped out by a few months of processor, disk, or memory technology progress.

8 Acknowledgments

We would like to thank Greg Ganger, Garth Gibson, Richard Golding, and several of our colleagues for their feedback and support, as well as all the authors of Pantheon. We would also like to thank Liddy Shriver, our USENIX shepherd, for her feedback and help.

References

[1] M. F. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin. Evaluating content management techniques for web proxy caches. *Performance Evaluation Review*, 27(4):3–11, Mar. 2000.

[2] M. F. Arlitt and C. L. Williamson. Web server workload characterization: The search for invariants. In *Proc. of SIGMETRICS 1996*, pages 126–137. July 1996.

[3] E. Bachmat, EMC. Private communication, Apr. 2002.

[4] D. Black, EMC. Private communication, Feb. 2002.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proc. of the 18th Ann. Joint Conf. of the IEEE Computer and Communications Soc.*, volume 1–3. Mar. 1999.

[6] D. Buck and M. Singha. An analytic study of caching in computer-systems. *Journal of Parallel and Distributed Computing*, 32(2):205–214, Feb. 1996. Erratum published in 34(2):233, May 1996.

[7] P. Cao, E. W. Felten, and K. Li. Application-controlled file caching policies. In *Proc. of the USENIX Assoc. Summer Conf.*, pages 171–182. June 1994.

[8] M. E. Crovella and A. Bestavros. Self-similarity in world wide web traffic evidence and possible causes. In *Proc. of SIGMETRICS 1996*, pages 160–169. July 1996.

[9] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation*, pages 267–280. Nov. 1994.

[10] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Trans. on Database Systems*, 9(4):560–595, Dec. 1984.

[11] EMC Corporation. Symmetrix 3000 and 5000 enterprise storage systems product description guide. http://www.emc.com/products/product_pdfs/pdg/symm_3_5_pdg.pdf, Feb. 1999.

[12] EMC Corporation. Symmetrix 8000 enterprise storage systems product description guide, Mar. 2001.

[13] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, and H. M. Levy. Implementing global memory management in a workstation cluster. In *Proc. of the 15th Symp. on Operating Systems Principles*, pages 201–212. Dec. 1995.

[14] M. J. Franklin, M. J. Carey, and M. Livny. Global memory management in client-server DBMS architectures. In *Proc. of the 18th Very Large Database Conf.*, pages 596–609. Aug. 1992.

[15] K. Froese and R. B. Bunt. The effect of client caching on file server workloads. In *Proc. of the 29th Hawaii International Conference on System Sciences*, pages 150–159, Jan. 1996.

[16] G. R. Ganger and Y. N. Patt. Using system-level models to evaluate I/O subsystem designs. *IEEE Trans. on Computers*, 47(6):667–678, June 1998.

[17] C. P. Grossman. Evolution of the DASD storage control. *IBM Systems Journal*, 28(2):196–226, 1989.

[18] Hewlett-Packard Company. HP 9000 N4000 Enterprise Server using HP-UX 11.00 64-bit and Informix Extended Parallel Server 8.30FC2: TPC-H full disclosure report, May 2000.

[19] S. Jiang and X. Zhuang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proc. of SIGMETRICS 2002*. June 2002.

[20] S. Jin and A. Bestavros. Popularity-aware greedy-dual-size web proxy caching algorithms. In *Proc. of the 20th Intl. Conf. on Distributed Computing Systems*, pages 254–261. Apr. 2000.

[21] R. Karedla, J. S. Love, and B. G. Wherry. Caching strategies to improve disk performance. *IEEE Computer*, 27(3):38–46, Mar. 1994.

[22] E. D. Katz, M. Butler, and R. McGrath. A scalable HTTP server: The NCSA prototype. *Computer Networks and ISDN Systems*, 27(2):155–164, Nov. 1994.

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

[24] B. McNutt. I/O subsystem configurations for ESA: New roles for processor storage. *IBM Systems Journal*, 32(2):252–264, 1993.

[25] J. Menon and M. Hartung. The IBM 3990 disk cache. In *Proc. of COMPCON 1988, the 33rd IEEE Intl. Computer Conf.*, pages 146–151, June 1988.

[26] D. W. Miller and D. T. Harper. Performance analysis of disk cache write policies. *Microprocessors and Microsystems*, 19(3):121–130, Apr. 1995.

[27] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems — or — your cache ain't nuthin' but trash. In *Proc. of the USENIX Assoc. Winter Conf.* Jan. 1992.

[28] D. Muntz, P. Honeyman, and C. J. Antonelli. Evaluating delayed write in a multilevel caching file system. In *Proc. of IFIP/IEEE Intl. Conf. on Distributed Platforms*, pages 415–429. Feb.–Mar. 1996.

[29] E. Rahm and D. F. Ferguson. Cache management algorithms for sequential data access. Research Report RC15486, IBM T.J. Watson Research Laboratories, Yorktown Heights, NY, 1993.

[30] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *Proc. of SIGMETRICS 1990*, pages 132–142. May 1990.

[31] C. Ruemmler and J. Wilkes. A trace-driven analysis of disk working set sizes. Tech. Rep. HPL-OSR-93-23, HP Laboratories, Palo Alto, CA, Apr. 1993.

[32] C. Ruemmler and J. Wilkes. UNIX disk access patterns. In *Proc. of the USENIX Assoc. Winter Conf.*, pages 405–420. Jan. 1993.

[33] C. Ruemmler and J. Wilkes. An introduction to disk drive modelling. *IEEE Computer*, 27(3):17–28, Mar. 1994.

[34] A. J. Smith. Bibliography on file and I/O system optimization and related topics. *Operating Systems Review*, 15(4):39–54, Oct. 1981.

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

[36] J. Z. Teng and R. A. Gumaer. Managing IBM Database 2 buffers to maximize performance. *IBM Systems Journal*, 23(2):211–218, 1984.

[37] Transaction Processing Performance Council. TPC benchmark H, Standard Specification Revision 1.3.0. http://www.tpc.org/tpch/spec/h130.pdf, June 1999.

[38] Transaction Processing Performance Council. TPC benchmark C, Standard Specification Version 5. http://www.tpc.org/tpcc/spec/tpcc_current.pdf, Feb. 2001.

[39] K. Treiber and J. Menon. Simulation study of cached RAID5 designs. In *Proc. of the 1st Conf. on High-Performance Computer Architecture*, pages 186–197. Jan. 1995.

[40] M. Uysal, A. Acharya, and J. Saltz. Requirements of I/O systems for parallel machines: An application-driven study. Tech. Rep. CS-TR-3802, Dept. of Computer Science, University of Maryland, College Park, MD, May 1997.

[41] A. Varma and Q. Jacobson. Destage algorithms for disk arrays with nonvolatile caches. In *Proc. of the 22nd Ann. Intl. Symp. on Computer Architecture*, pages 83–95. June 1995.

[42] G. M. Voelker, E. J. Anderson, T. Kimbrel, M. J. Feeley, J. S. Chase, and A. R. Karlin. Implementing cooperative prefetching and caching in a globally managed memory system. In *Proc. of SIGMETRICS 1998*, pages 33–43. June 1998.

[43] D. Voigt. HP AutoRAID field performance. HP World talk 3354, http://www.hpl.hp.com/SSP/papers/, Aug. 1998.

[44] J. Wilkes. The Pantheon storage-system simulator. Tech. Rep. HPL-SSP-95-14 rev. 1, HP Laboratories, Palo Alto, CA, May 1996.

[45] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Trans. on Computer Systems*, 14(1):108–136, Feb. 1996.

[46] D. L. Willick, D. L. Eager, and R. B. Bunt. Disk cache replacement policies for network fileservers. In *Proc. of the 13th Intl. Conf. on Distributed Computing Systems*, pages 2–11. May 1993.

[47] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. The scale and performance of cooperative web proxy caching. In *Proc. of the 17th Symp. on Operating Systems Principles*, pages 16–31. Dec. 1999.

[48] Y. Zhou and J. F. Philbin. The Multi-Queue replacement algorithm for second level buffer caches. In *Proc. of the USENIX Ann. Technical Conf.*, pages 91–104. June 2001.

[49] G. K. Zipf. *Human Behavior and Principle of Least Effort*. Addison-Wesley Press, Cambridge, MA, 1949.