# My cache or yours?
# Making storage more exclusive

Theodore M. Wong Gregory R. Ganger John Wilkes[1]

November 2000
CMU-CS-00-157

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Abstract**

Modern high-end disk arrays typically have several gigabytes of cache RAM. Unfortunately, most array caches employ management policies in which the same data blocks are cached at both the client and array levels of the cache hierarchy—that is, they are *inclusive*. As a result, 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.

This paper explores the potential benefits of *exclusive* caches, in which data blocks are either in the client or array cache, but never in both. Exclusivity helps to create the effect of a single, large unified cache. We propose an operation called DEMOTE for transferring data ejected from the client cache to the array cache, and explore its effectiveness in increasing cache exclusivity using simulation studies. We quantify the benefits of DEMOTE, the overhead it adds, and the effects of combining it with different cache replacement policies across a variety of workloads. The results show that we can obtain useful speedups for both synthetic and real-life workloads.

[1]Hewlett-Packard Laboratories, `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 also thank the members and companies of the Parallel Data Consortium (including EMC, Hewlett Packard, Hitachi, IBM, Intel, LSI Logic, Lucent Technologies, Network Appliance, PANASAS, Quantum, Seagate Technology, Sun Microsystems, Veritas Software, and 3Com) for their interest, insight, and support. 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:** file caching, disk array caching

# 1 Introduction

In this paper we are concerned with evaluating the effectiveness of a technique for improving the effectiveness of RAM caches in storage devices, such as disk arrays. Such disk arrays contain a RAM cache for several reasons: for use as a speed-matching buffer between the relatively slow disk drives and the fast client interconnects, to add asynchrony in the form of read-ahead and write-behind to client accesses (the write behind data is frequently mirrored in non-volatile memory to survive component failure and power outages), and to act as a low-latency pool of data that is accessed multiple times by its clients.

Read-ahead can be handled efficiently by buffers that are only a few times larger than the track size of the back-end disk drives in the array, and the space needed for write-behind is bounded by the burstiness of the write workload [22, 28]. Both are typically tiny compared to the underlying disk capacity. As a result, we are concerned here with caching for re-reads, because by far it represents the largest portion of the cache.

The common rule of thumb is to cache about 10% of the active storage. Table 1 of representative arrays and servers suggests that this is a luxury that is out of reach of even the most aggressive cache configurations if all of the stored data is active. Fortunately, we generally do not need so much cache: a study of UNIX file system loads [21] found that the active working set over a 24 hour period was only 3–7% of the total storage on average, and the 90th percentile reached only 6–16% of the total. This data is supported by a more recent study of deployed AutoRAID systems [32], in which the working set rarely exceeded the space available for RAID1 storage (about 10% of the total array capacity).

Unfortunately, interactions between the least-recently-used (LRU) cache replacement policy typically employed in both the client and the array often cause caching at the array level to be *inclusive*, meaning that data blocks will be duplicated at the two levels of the cache hierarchy. Because of this, read requests that miss in the client will often also miss in the array and incur a disk access penalty. For example, suppose that we have a client with 16 GB of cache connected to an array with 16 GB of cache. Next, suppose the client has a working set size of 32 GB. We might naïvely expect the working set of the client to fit in the aggregate of the client and array caches at steady state, but in general it will not, due to the array having to cache the blocks that are already at the client—in other words, cache inclusivity.

## 1.1 Exclusive caching

We evaluate an approach that reduces the amount of inclusive caching in a client-array system. Ideally, we would like to achieve perfectly *exclusive* caches, in which data blocks are either at the client or array level, but not both.

All read requests that miss in the client are sent to the array; the more disjoint the set of blocks the array cache holds is from the client, the more likely it is to have the desired block cached. Even though a data transfer across the client to array interconnect is still needed, increasing the hit fraction of the array cache offers a big performance gain because reading a block from a disk is many orders of magnitude slower than transferring it from the array to the client. Thus, reducing the aggregate cache miss rate (the fraction of requests that miss in all caches) can lead to a dramatic reduction in the mean request latency. We evaluate how close we can get to this desirable state of affairs, as well as how beneficial it is to do so.

| High-end storage systems | | |
|---|---|---|
| *System* | *Max array RAM* | *Max disk space* |
| EMC Symmetrix | 16 GB | 9 TB |
| IBM Shark | 16 GB | 11 TB |
| IBM RAMAC | 6 GB | 1.7 TB |
| HP XP256 | 16 GB | 9 TB |
| HP XP512 | 32 GB | 18 TB |
| **High-end processing systems** | | |
| *System* | *Max array RAM* | *Server type* |
| HP A-class | 4 GB | ISP server |
| HP N-class | 32 GB | Mid-range |
| IBM NUMA-Q | 64 GB | High-end |
| Sun E5000 | 14 GB | Mid-range |
| Sun E10000 | 64 GB | High-end |

Table 1: Some representative cache and server capacities

To do this, we introduce the DEMOTE operation into the protocol between the array and its clients. This operates as follows: when a client decides to eject a clean block from its cache (usually to accommodate a READ), it first uses a DEMOTE to transfer it to the array cache. The DEMOTE behaves a bit like a write: the array puts the demoted block into its read cache, ejecting another block if necessary to make space; it returns "success" immediately without transferring data if it has already cached the demoted block. But if the array cannot immediately accommodate the DEMOTE (for example, if it cannot make any free space for it), it simply rejects the demotion request, and does nothing. In all cases, the client then discards the block from its own cache.

To evaluate the performance of demotion-based protocols, we aim to answer the following questions:

- What increase in array hit fraction do we obtain by using DEMOTE?
- Does the cost of demoting blocks exceed the benefits obtained? (Such costs include increased client to array network traffic, and delays experienced by client requests that are waiting for a demotion to occur before they can proceed.)
- How sensitive are the results to variations in network bandwidth (the main contributor to demotion cost)?
- Do demotions help in a system with multiple clients?

The remainder of the paper is structured as follows. We begin with a survey of related work in Section 2. Section 3 presents a discussion on the potential benefits of exclusive caching for a set of representative workloads. Section 4 describes the alternative designs we evaluated, and the experimental setup used. Sections 4.3, 5 and 6 describe the results of these evaluations. We end with some observations and conclusions from our work.

# 2 Related work

The literature on caching in storage systems is very rich, so only representative samples can be provided here. Much of it concentrates on predicting the performance of an existing cache hierarchy [24, 25, 15, 3], describing existing systems [16, 11, 28], and determining when to flush write-back data to disk [13, 17, 30].

Even though previous work has questioned the utility of caches outside the clients in distributed file systems [18], studies of real workloads demonstrate that read caching has considerable value in disk arrays, and that a small amount of non-volatile memory greatly improves write performance [28].

Choosing the correct cache replacement policy in an array can significantly improve its performance [25, 20, 13]. Some studies suggest using least-frequently-used [35, 10] or frequency-based [20] replacement policies instead of LRU in file servers. Most-recently-used (MRU) [14] or next-block prediction [19] policies often provide better performance for sequential loads. LRU or clocking policies [7] seem to yield acceptable results for database loads: for example, the IBM DB2 database system [26] implements an augmented LRU-style policy.

Our DEMOTE operation can be viewed as a very simple form of client-controlled caching policy [4]. The difference is that we provide no way for the client to control which blocks should be replaced in the array, and we largely trust the client's system to be well-behaved, since it is only likely to hurt itself if it misbehaves.

Recent studies of cooperative World Wide Web caching protocols [1, 12, 36] have looked at policies beyond LRU and MRU. Previously, analysis of web request traces [2, 5] show that their distributions of file popularity tend to follow Zipf's Law [37]. It is possible that protocols that are 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.

Peer-to-peer cooperative caching studies have relevance to the case of multiple clients sharing a single array. In the "direct client cooperation" model [6], active clients were allowed to offload excess blocks onto idle peers. No inter-client sharing occurred— this was simply a way to exploit otherwise unused memory. Finding the nodes with idle memory was one of the major design issues. This problem was further addressed in the work that implemented and evaluated a global memory management protocol called GMS [8, 31], in which cooperating nodes use approximate knowledge of the global memory state to make caching and ejection decisions that benefit both 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 [9]. Here, the database server keeps a directory of pages in the aggregate cache. This directory allows

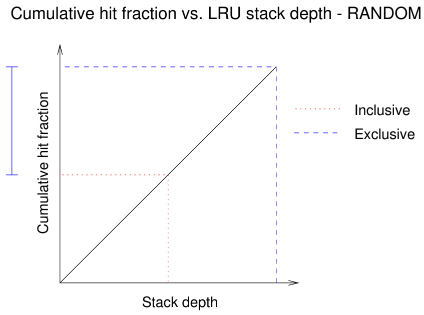



Figure 1: LRU stack depth graph for a random workload. The marker on the Y-axis shows the increase in the aggregate hit fraction from an exclusive policy.

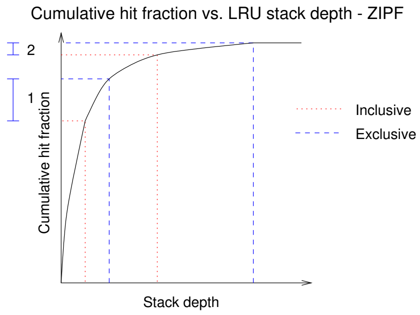



Figure 2: LRU stack depth graph for a Zipf-like workload. Markers 1 and 2 on the Y-axis show the increase in the aggregate hit fractions from an exclusive policy at two different client cache sizes.

the server to forward a page request from one client to another that has the data, request a client to demote rather than discard the last in-memory copy of a page, and preferentially discard pages that have already been sent to a client. In contrast, we do not track which client has what block, and so we cannot support client-to-client transfers. We simply take advantage of a high-speed network to do a demotion, rather than ask if it is worthwhile, and we do not require a (potentially large) data structure in the array to keep track of what is where. The reduction in complexity has a price: our solution is less able to exploit block sharing between clients.

In our work, we are tackling a simpler problem: determining whether a small, simple extension to existing cache management protocols can achieve worthwhile benefits. We do not need to determine whether the clients are idle, nor do we need to handle the complexity of performing load balancing between them, or between the array and the clients.

# 3 Estimating the upside of exclusive caching

In this section we examine the possible upside of exclusive caching, in order to answer the question “is the idea worth pursuing at all?”

For simplicity, we first consider a single-client system in which the client and the array have the same size caches, and use the same (LRU) replacement policy. For a perfectly inclusive array cache, all read requests that miss in the client will also miss in the array, whereas an exclusive caching scheme will turn some of those client misses into hits in the array cache. Thus, assuming that we do not change the client replacement policy to achieve exclusivity, any increase in the aggregate hit fraction (the fraction of requests that hit in either the client or array) is due to hits in the array.

We can use the LRU stack depth graph for a particular workload to determine the fraction of requests that will hit in the client and in an exclusive array. Then, the following expression gives us an estimate of the mean latency for inclusive and exclusive systems:

$$T_{mean} = (hit_c + hit_a + hit_d) * T_c + T_a * (hit_a + hit_d) + T_d * hit_d \qquad (1)$$

$T_c$ is client latency, $T_a$ is the array-to-client latency, $T_d$ is the disk-to-array latency, and $hit_c$, $hit_a$, and $hit_d$ are the hit fractions in the client, array, and disk.

We consider four types of workload for our single-client system below:

- *Random workloads*: The client reads from a working set of $N_{rand}$ blocks with a uniform probability of reading each block, which is behavior typical of transaction-processing benchmarks such as TPC-C [27]. There is a linear relationship between the aggregate cache size and the cumulative hit fraction for such workloads; thus, doubling the cache size (by switching from inclusive to exclusive caching) doubles the hit fraction, as shown in Figure 1.

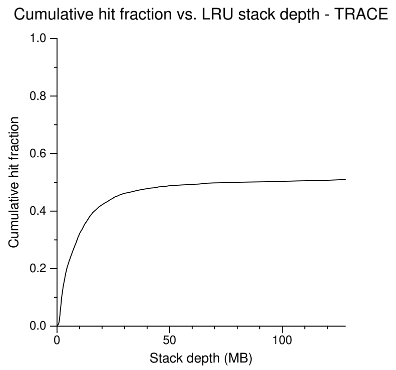



Figure 3: LRU stack depth graph for a traced workload.

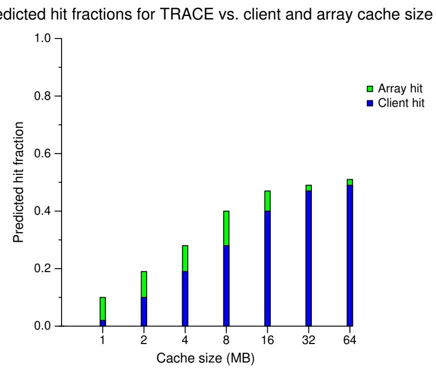



Figure 4: Predicted hit fraction vs. cache size. For each bar, the client and array have equal cache sizes, and the vertical numbers are the percent reductions in the miss rate from using an exclusive policy.

- *Sequential workloads*: The client requests a sequence of $N_{seq}$ blocks repeatedly. Such workloads benefit from an exclusive policy only if the sequence would not fit in the client alone, but would fit in the aggregate cache.
- *Zipf-like workloads*: The client reads from a working set of $N_{Zipf}$ blocks, where the probability of reading the $i^{th}$ block is proportional to $1/i$, yielding a Zipf-like distribution [37].

  When the client cache is not large enough to capture the most frequently-read blocks, an exclusive policy can provide a significant improvement in the aggregate hit fraction (marker 1 in Figure 2). As the client cache size increases to cover more of the working set, the benefit of an exclusive policy dwindles (marker 2).
- *Traced workloads*: The client replays a 1-week (May 30 – June 5 1992) read request trace from `snake`, a small HP-UX file server system with approximately 4 MB of cache [22] and an 8 KB block size. We generated the LRU stack depth graph by first warming up an infinite cache with the May 23 – May 29 week of trace data, and then collecting statistics on the following week's trace. The results are shown in Figure 3. We truncated the stack depth at 128 MB, given the relatively small cache and disk sizes of the original system by today's standards.

  To estimate the benefit of an exclusive policy, we used Figure 3 to compute the hit fraction for various cache sizes. We then summarized the results in Figure 4.

  Figure 4 shows that if the original 4 MB client cache had been connected to an array with a cache of the same size, it would have obtained a 9% hit fraction with an exclusive policy. As expected, the benefits of the exclusive caching policy decreases as the client size increases, as it did for the Zipf workload.

From the above analysis, we concluded that there was indeed merit in pursuing a more detailed study across a wider range of workloads, and the remainder of this paper reports on our results from doing just that.

# 4 Experiments

We divided our experimental evaluation of exclusive caching into the following phases:

1. **Basic evaluation**: we confirmed the basic effectiveness of the DEMOTE operation, by performing a detailed performance simulation of its behavior with the workloads described above.
2. **Sensitivity analysis**: we investigated the effects of reducing the available network bandwidth to determine the point at which the overhead of performing the DEMOTE operation (primarily due to increased network traffic) outweighs the benefits of increasing the cache exclusivity.

| *Protocol* | *Notes* |
|---|---|
| LRU - LRU - none | Baseline protocol |
| LRU - MRU - none | |
| MRU - LRU - none | |
| MRU - MRU - none | |
| LRU - LRU - demote | |
| LRU - MRU - demote | Proposed prototype |
| MRU - LRU - demote | |
| MRU - MRU - demote | |

Figure 5: The explored designs.

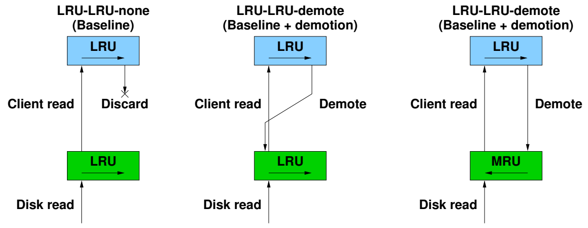



Figure 6: Promotions and demotions in three potential designs. The top box shows the client replacement queue, while the bottom one shows the array queue. The arrows in the boxes point to the "eject me next" end of the queue.

3. **Multiple-client evaluation**: we repeated our analysis on multiple-client traced workloads, and again validated our predictive models, and showed that reductions in latency are available in real multi-client systems.

Since the primary purpose of the demotion protocols is to improve the read request response time, rather than merely to increase the cache hit fractions, we report on the mean latency per request as well as the hit fractions at each client and the array.

## 4.1 The design space

The basic design choice we explored was between systems that did demotions and those that did not. In addition, given the existing data on the inappropriateness of LRU replacement policies for certain workloads, we decided that we would explore the cross-product of LRU and MRU replacement policies at the client and the server. Our *baseline* protocol employed LRU at both the client and array with no demotions, a setup that is typical of several distributed file systems [10]. Our *proposed* protocol, which supported block demotion, employed LRU at the client and MRU at the array.

The cache replacement policies and the choice on whether to demote blocks gave us eight possible designs. In following sections, we refer to each design by its client and array replacement policies, and by whether or not demotions are occurring. Thus, the baseline is also referred to as LRU-LRU-none, and the proposed protocol as LRU-MRU-demote. We summarize the designs in Figure 5.

The array always stores demoted blocks such that it will eject them in LRU order if it receives no further requests for them, which affects how the blocks are added to the replacement queue. Figure 6 shows this for three common cases. An array employing an LRU replacement policy adds disk-read blocks and demoted blocks to the "eject me last" end of its queue, while an array employing an MRU replacement policy adds disk-read blocks to the "eject me next" end, but adds demoted blocks to the "eject me last" end. We therefore expect the LRU-MRU-demote protocol to cause the aggregate cache to behave as a unified LRU cache.

We note that when an array with an LRU policy receives a request for a block, it retains that block for at least $N_a$ more block additions to the replacement queue, since it will move the first block to the most-recently used end of the queue. In contrast, an array with an MRU policy ejects the first block on the next addition. In either case that first block is double-cached at the array and the client level for at least one request. We will return to these details later.

Figure 6 omits designs in which the array adds demoted blocks such that it will eject them in MRU order. For them, the client would have been better off discarding them instead of demoting them, because that is the next thing the array will do anyway. Such designs are fully inclusive, and so all requests that miss in the client will also miss in the array. We confirmed by experiment that these policies performed poorly across all the synthetic workloads described below, and do not consider them further.

## 4.2 Evaluation environment

For the first evaluation of our prototype protocols, we used the Pantheon simulator [33], which includes calibrated disk models [23]. This simulator was used successfully in design studies of the AutoRAID array [34], so we have

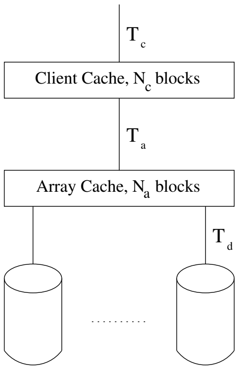



Figure 7: Evaluation system setup.

confidence in its predictive powers.

We configured the simulator to model a simple RAID5 four-disk array (five-disk for the traced workload experiments), connected to a single client over a 1 Gbit/s Fibre Channel link, as shown in Figure 7. For 4 KB read requests, we set $T_c = 0, T_a = 0.2$ ms (which includes both the array controller and Fibre Channel network costs), and observed the disk models in the simulator as giving an average $T_d \approx 10$ ms.

The Pantheon cache models are extremely detailed, keeping track of disk I/O operations in disk-sector-sized units. This requires large amounts of memory, and meant that we had to restrict both the client and array caches to 64 MB each. For a block size of 4 KB, this gives us the number of blocks the client and array can hold as $N_c = N_a = 16384$ blocks.

## 4.3 Evaluation workloads

The following synthetic and traced workloads were used in evaluation and sensitivity experiments, and were chosen to show the maximal benefit available from exclusive caching. In each case, we constructed the workloads to allow one read request to finish before the next began.

## 4.4 The RANDOM workload

We set $N_{rand}$ for the random workload in Section 3 equal to the combined size of the client and array caches, so with $N_c = N_a = 16384$ blocks, $N_{rand}$ = 32768 blocks. We issued $N_{rand}$ one-block read requests to warm up the system, followed by $10 \times N_{rand}$ one-block requests.

We expected that half of the read requests for this workload would hit in the client, so protocols that maximized the number of remaining requests that hit in the array would perform the best. The results in Figures 8 and 9 validate this expectation—we see that the best-performing protocols, LRU-MRU-demote and MRU-MRU-demote, have the highest array hit counts and the lowest mean latencies. That both of these protocols should do well is unsurprising, since for random workloads the choice of client replacement policy matters less than ensuring that blocks are not duplicated in the system. This fact is reflected in the latency reductions: for example, the LRU-MRU-demote proposed protocol is $7.5\times$ faster than the baseline case.

Figure 10 compares the cumulative latencies of the baseline and proposed protocol. For the proposed protocol, the jump in the cumulative request fraction at 0.4 ms corresponds to a combined hit in the array cache plus the cost of a demotion. In contrast, the baseline protocol gets fewer array hits (as seen in Figure 8), and its curve has a significantly smaller jump at 0.2 ms, which is the array cache hit time without demotions.

Although we might expect LRU-LRU-demote to transform the aggregate cache into a unified LRU cache, and do as well as LRU-MRU-demote or MRU-MRU-demote, it will in fact always be more inclusive. As noted above, the most recently demand-read block is double-cached until it gets ejected; with LRU-LRU-demote, the block remains double-cached until the client demotes it or the array ejects it. With a random workload whose working set is twice the size of the client or array caches, this on average occurs only after $\min(N_c, N_a)/2$ requests miss, in contrast to the

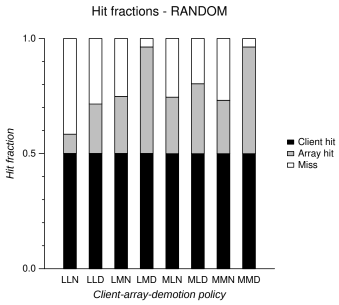



Figure 8: Hit fractions for RANDOM.

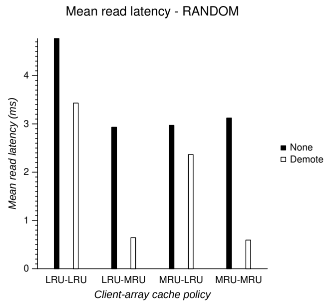



Figure 9: Mean latencies for RANDOM (variances in the means are below $1\%$).

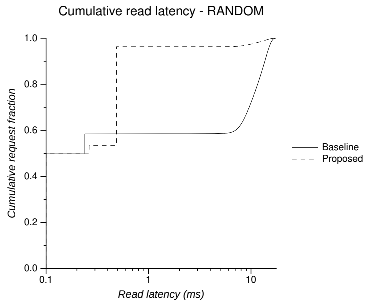



Figure 10: Cumulative latencies for RANDOM with baseline and proposed protocols.

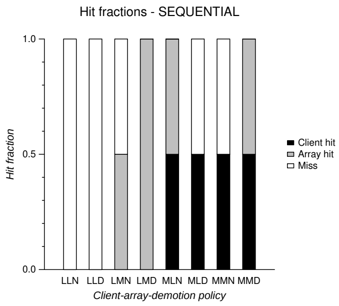



Figure 11: Hit fractions for SEQUENTIAL

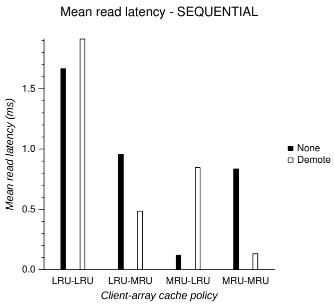



Figure 12: Mean latencies for SEQUENTIAL (variances in the means are below $1\%$).

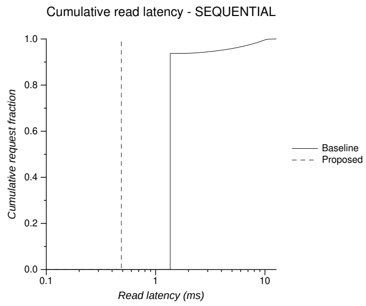



Figure 13: Cumulative latencies for SEQUENTIAL with baseline and proposed protocols.

single miss required by LRU-MRU-demote. The effect is to partially pollute the array cache with blocks that should otherwise be evicted.

## 4.5 The SEQUENTIAL workload

We set $N_{seq}$ for the sequential workload in Section 3 so that the working set fully occupies the aggregate caches, so for $N_c = N_a = 16384$ blocks, $N_{seq}$ = 32767. As mentioned in Section 4.1, we needed to allow for the most recently referenced block being double-cached, hence $N_{seq} \neq N_c + N_a$ blocks.

To assist in analyzing the caches' behavior, we define $S_0$ to contain the first half of the blocks in the sequence, and $S_1$ to contain the second half. As for RANDOM, we issued $N_{seq}$ one-block read requests to warm up the system, followed by $10 \times N_{seq}$ one-block requests.

Figures 11, 12 and 13 show the hit fractions, resulting mean latencies, and cumulative distributions of latencies for SEQUENTIAL.

The behavior of the system with SEQUENTIAL is significantly different from RANDOM or ZIPF. This is because the client can do best by employing an MRU policy: recently-consumed blocks will be used furthest in the future, so using an LRU policy is counter-productive. This is a well-known result for sequential workloads [14], and is confirmed by the mean latency results in Figure 12.

In contrast to the previous workload, LRU-MRU-demote delivers poor performance compared to some of the other prototypes. It keeps everything cached in the aggregate cache, but all of its hits are in the array (Figure 11), causing it to lose out to MRU-LRU-demote, which hits in the client for half of the sequential request cycle.

Again, LRU-LRU-demote does not maintain a unified LRU, as the following example below shows. Not only are we always missing the next block in the request sequence, but we also need to perform a full transfer of the demoted

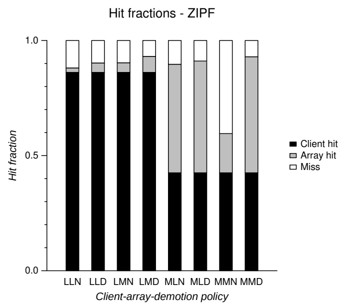



Figure 14: Hit fractions for ZIPF.

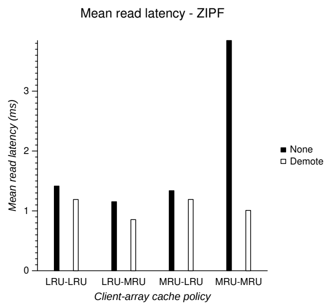



Figure 15: Mean latencies for ZIPF (variances in the means are below 2%).

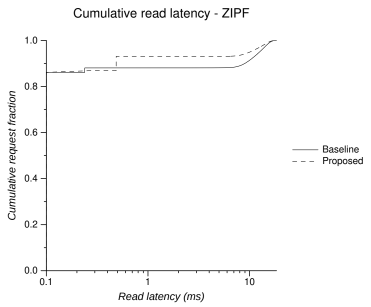



Figure 16: Cumulative latencies for ZIPF with baseline and proposed protocols.

block. Consider a system where the client and array can each hold three blocks, and the sequential workload consists of the five blocks A–E. The next block to eject is at the left of the queues shown.

| *Cache* | *Start* | *Next* | *Final* |
|---|---|---|---|
| Client | C D E | D E - | D E A |
| Array | D B E | B E C | E C A |

We begin with the caches after an initial warm-up sequence (state *Start*), and with A as the next block to read. To make room, the client demotes the LRU block C, and the array puts it at the longest-lived end of its replacement queue (state *Next*). The array then ejects block B to make way for A (state *Final*)—but must then retrieve B from disk on the next request.

The best design is MRU-LRU-none. This is because after a single warm-up set of $N_{seq}$ requests, the client contains the $N_{seq}$/2 least recently requested blocks (the blocks in $S_0$), and the array contains the $N_{seq}$/2 most recently requested ($S_1$). Thus, of the next $N_{seq}$ requests, the first $N_{seq}$/2 hit in the client, and the second $N_{seq}$/2 hit in the array. Since the MRU client ejects one $S_0$ block to make way for blocks read from $S_1$, we go to disk once per $N_{seq}$ requests.

MRU-MRU-demote exhibits similar behavior, but it incurs the additional overhead of demoting blocks to the array. Here, blocks read from $S_1$ are immediately demoted on the next request; since these demoted blocks are already in the array, the array only reorders its replacement queue instead of transferring the block.

### 4.6 The ZIPF workload

We set $N_{Zipf}$ for the Zipf-like workload in Section 3 to be one and a half times the block capacity of the aggregate cache, which for $N_c = N_a = 16384$ is 49152 blocks. We then subdivided the blocks into three regions: $Z_0$ for the most active blocks, $Z_1$ for the next-most active blocks, and $Z_2$ for the least active blocks. For our value of $N_{Zipf}$, we see that $Z_0$, $Z_1$, and $Z_2$ will each have $\frac{1}{3}N_{Zipf}$ blocks. The corresponding probabilities of selecting a block from each set will be 90%, 6%, and 4%.

With this size and subdivision, we expect an inclusive policy on average to double-cache blocks from $Z_0$ that the client should already hold, while an exclusive policy should cache blocks from $Z_1$. In both cases, neither the client nor the array will cache blocks from $Z_2$. We issued $N_{Zipf}$ one-block requests to warm up the system, followed by $10 \times N_{Zipf}$ one-block requests.

At first sight, this distribution might seem too skewed towards cacheable data: shouldn't the amount of cacheable data be about 10% of the whole? The answer is that in many environments where caching is used the access skews are such that disk array caches achieve about 80% hit fractions. This is only possible if the active working set is much smaller than the whole—an observation borne out by analysis of real traces [21].

Figures 14, 15 and 16 show the hit fractions, resulting mean latencies, and cumulative distributions of latencies for the ZIPF.

The best performance here is obtained from LRU-MRU-demote. As expected, it transforms the aggregate cache into a unified LRU cache, which caches blocks from $Z_0$ and $Z_1$ in preference to ones from $Z_2$.

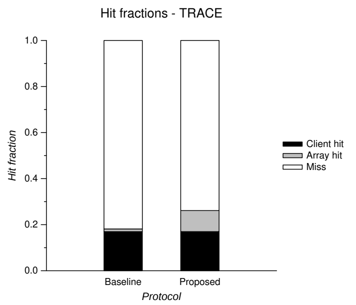



Figure 17: Hit fractions for TRACE.

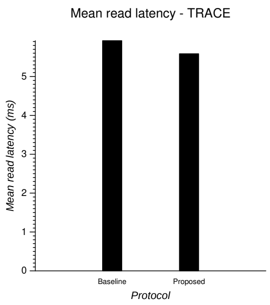



Figure 18: Mean latencies for TRACE (variances in the means are below 1%).

| *Workload* | *Base* | *Proposed* | *Speedup* |
|---|---|---|---|
| RANDOM | 4.77 ms | 0.64 ms | 7.5× |
| SEQ. (LRU client) | 1.67 ms | 0.48 ms | 3.5× |
| SEQ. (MRU client) | 0.12 ms | 0.13 ms | 0.9× |
| ZIPF | 1.41 ms | 0.85 ms | 1.7× |
| TRACE | 5.92 ms | 5.59 ms | 1.06× |

Table 2: Summary of the synthetic workload results. It shows mean latencies, together with the ratio of baseline to proposed protocol latencies. The "SEQ. (MRU client)" line shows the results of the MRU-LRU-none and MRU-MRU-demote protocols.

The cumulative latency graph in Figure 16 shows similar characteristics as Figure 10 did for RANDOM. The curve for the proposed protocol has a jump at 0.4 ms not seen in the baseline curve, since the prototype achieves more hits in the array.

As with RANDOM, LRU-LRU-demote also transforms the aggregate cache into a unified LRU cache, but has a slightly higher mean latency and lower array hit count because the array takes longer to discard doubly-cached blocks. The mean latency for LRU-LRU-demote is 1.19 ms, compared to a mean latency of 0.85 ms for the proposed prototype.

## 4.7 The TRACE workload

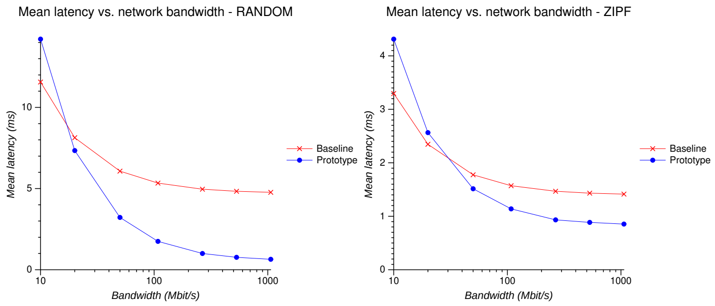
We use the traced workload as described in Section 3. Unlike the system used to evaluate the synthetic workload behavior, we set the client and array cache size to 4 MB, so that we could measure the benefits of exclusive caching to the original traced system.

Figures 17 and 18 show the hit fractions and mean latencies for TRACE. We allowed each read request to run to completion before the next was issued, and compared the performance of the baseline protocol to the proposed protocol.

The results seen here confirm the estimates we made in Section 3. The proposed protocol achieves a 7.3% hit fraction in the array cache (compared with less than 1% before), and corresponding 1.06× reduction in the mean latency.

## 4.8 Evaluation summary

Table 2 shows the results so far. The exclusive caching model shows considerable promise, except for purely sequential workloads where an MRU client outperforms everything else.




(a) RANDOM (b) ZIPF

Figure 19: Latency vs. bandwidth for RANDOM and ZIPF

# 5 Sensitivity analysis

In this section, we explore the effect of relaxing the assumption of a low latency, high bandwidth link between the array and the client, by varying the bandwidth setting in our simulator from 1 Gbit/s down to 10 Mbit/s. The proposed protocol relies on a low-latency link to make the array cache perform like a low-access-penalty extension of the client cache. We would expect the benefit of cache hits in the array to decrease as the client-array access latency increases. Demotions will take longer—possibly to the point where they are no longer worth performing.

Our experiments for RANDOM verified our expectations. We see in Figure 19 that at low bandwidth, the baseline protocol outperforms the one that does demotions. With even a small increase in bandwidth, the exclusive-caching protocol wins out. The results for ZIPF are similar to those for RANDOM, except that the gap between the baseline and prototype curves for high-bandwidth networks is smaller since the increase in cache hit fractions is smaller.

# 6 Multiple-client systems

Multiple-client systems introduce a new variable: the fraction of requests that are shared across the clients. Because we do not attempt to maintain a full map of the memories in the clients, our approach can result in blocks being cached in multiple clients. This is a result of our deliberate focus on a much simpler mechanism than global management systems such as GMS [8, 31].

To assist with reasoning about the effects of caching behavior in multi-client systems, consider two boundary cases:

- *Fully disjoint requests*: The clients each issue read requests for non-overlapping portions of the aggregate working set. The requests will appear to the array as if they were issued by a single client with a cache as large as the sum of the individual caches.

  We can predict the benefit of exclusive caching from the LRU stack depth graph for the array, which we generate by treating all of the read requests as if the single client had issued them. The analysis then proceeds as for single-client systems in Section 3.

- *Fully conjoint requests*: The clients run in lock-step, issuing the same read requests in the same order at the same time. If we designate one of the clients as the leader, we observe that all requests that hit in the leader's cache will also hit in the others. So, to estimate the resulting behavior for the leader, we use the LRU stack depth graph to estimate the hit fractions for inclusive and exclusive systems as for single-client systems in Section 3. The mean latency for the leader is then given by the following (using Equation 1, and simplifying by assuming $T_c \approx 0$):

$$T_{mean-l} = T_a * (hit_a + hit_d) + T_d * hit_d \tag{2}$$

For the followers, we observe that all of the leader's requests that miss in the array will become an array cache hit for the followers, since they will all end up pipelined just at the end of the leader's successful requests that bring blocks into the array's cache. So, for them:

$$T_{mean-f} = T_a * (1 - hit_c) \tag{3}$$

For $C_n$ clients, the mean latency is then $T_{mean} = (T_{mean-l} + (C_n - 1) \times T_{mean-f})/C_n$. The only wrinkle here is that for exclusive protocols the followers will waste time trying to demote blocks that the leader has already demoted.

For the multiple-client experiments, we required a simulator that was capable of modelling gigabyte-sized caches, which was beyond Pantheon's abilities on the workstations we were using to run it. To handle this, we used a simpler simulator called `fscachesim` that only tracked the contents of the clients and array caches, without performing the detailed network and disk latency measurements that Pantheon provided. To verify the accuracy of `fscachesim`, we ran the `TRACE` workload through a simulated single-client system and check that its hit-fraction results matched those from Pantheon.

To estimate the mean latency from hit fractions for the multiple-client experiments, we used the latencies from the Pantheon models, substituted them in to Equation 1, and then used these values in `fscachesim`:

$$T_{mean} = 0.2 * (hit_d + hit_a) + 10 * hit_d \tag{4}$$

Equation 4 does not account for the added costs of maintaining exclusivity by demoting blocks. To approximate this case, we assumed that a demotion roughly doubled the cost of a request that hit in the array cache, and so Equation 1 becomes:

$$T_{mean} = 0.4 * (hit_d + hit_a) + 10 * hit_d \tag{5}$$

For our experiments, we selected two non-scientific workloads used in a study of I/O requirements for applications on parallel machines [29]. Our preliminary results, presented below, show that multiple-client systems can also benefit from exclusive caching protocols.

## 6.1 The DB2 workload

This workload was generated by an IBM DB2 database application performing join, set and aggregation operations on a 5.2 GB dataset. This workload demonstrates the disjoint workload behavior. The LRU stack graphs for the individual nodes show mainly sequential behavior, while the aggregated stack-depth graph for all the workloads together looks more like a random load after a portion where the hit fraction is close to zero (from 0 to 1.6 GB stack depth): there is a roughly linear increase in hit fraction with stack depth. This is a classic example of a trace for which caching is essentially useless: even when the stack depth reaches the full size of the on-disk working set, nearly half the accesses are misses.[1]

Our results in Figure 21 show promise: for a system of eight 256 MB clients and a 2 GB array, we obtain an average improvement in mean latency of $1.3\times$.

## 6.2 The HTTPD workload

This workload is generated by a parallel web-server serving 524 MB of data from seven nodes. The LRU stack graph for each node and the aggregate graph (Figure 22) show Zipf-like distributions. There is also some sharing between nodes, so we would expect the results to lean towards the conjoint behavior model.

Again, we see that exclusive policies also benefit multiple-client systems in Figure 23: for a system of seven 64 MB clients and a 64 MB array, we obtain an average improvement in mean latency of $1.17\times$.

[1] Note to reviewers: this is also partly an artifact of the trace: we used 10% of the trace to warm up the cache, but even the full trace would not have been enough. Unfortunately, longer traces were not available to us. We are continuing to look for better examples.

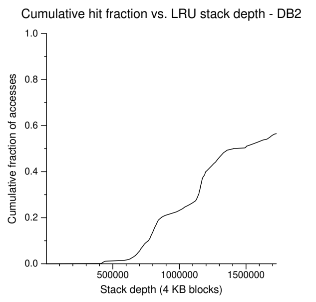



Figure 20: LRU stack depth graph for the DB2 workload.

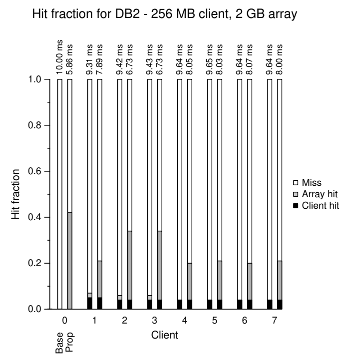



Figure 21: Fraction of requests issued by each client that hit either the client or array cache for the DB2 workload. The number at the top of each bar shows the mean latency as predicted by Equations 4 and 5. 'Prop' refers to the proposed protocol.

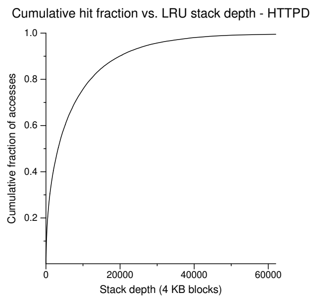



Figure 22: LRU stack depth graph for the HTTPD workload.

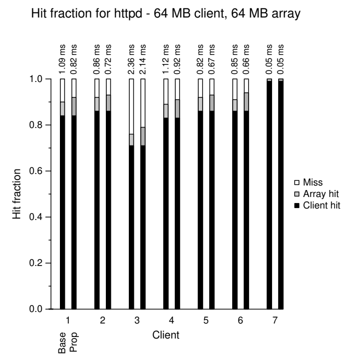



Figure 23: Fraction of requests issued by each client that hit either the client or array cache for the HTTPD workload. The number at the top of each bar shows the mean latency as predicted by Equations 4 and 5. 'Prop' refers to the proposed protocol.

## 7 Conclusion

This paper has explored some of the effects of a very simple, direct approach to making disk array caches more exclusive. Like many simple ideas, it is both surprisingly effective and rather obvious—in retrospect!

We have seen that exclusive caching can provide significant benefits when evaluated against synthetic workloads. We achieved speedups of up to $7.5\times$ for random and sequential workloads, and obtained a $1.7\times$ speedup for a Zipf-like workload from a $6\%$ increase in the array cache hit fraction.

These benefits survived severe reductions in the effective bandwidth of the interconnect link, although they were eventually negated by the cost of doing demotions—but only when the link had been slowed down by a factor of about 20, to below 50 Mbit/s

With the traced workload, the results were somewhat more modest: we observed a speedup of $1.06\times$. As expected, the benefits of demotion are dependent on the LRU stack behavior of the traced workload.

Exclusive policies may also benefit multiple-client systems. In our initial experiments, we obtained a $1.3\times$ speedup for database loads, and a $1.17\times$ speedup for HTTP daemon loads.

The results for this simple idea are encouraging: exclusive caching can indeed provide benefits to an interesting range of workloads, with remarkably little extra mechanism—just a DEMOTE operation that clients can use to transfer their unwanted blocks to the storage system.

## 8 Acknowledgements

We would like to thank Garth Gibson for providing invaluable feedback and support throughout the writing of this paper; Richard Golding for assisting with the design of the experiments and feedback; Bruce Worthington for adding the disk array model to Pantheon; Ralph Becker-Szendy, Dave Stewart, and Alistair Veitch for helping us to continue the good fight to improve Pantheon; IBM and Intel for providing the machines that these simulations ran on; Chris Colohan, Andrew Klosterman, and Paul Mazaitis for proofreading the final drafts.

## References

[1] ARLITT, M. F., CHERKASOVA, L., DILLEY, J., FRIEDRICH, R., AND JIN, T. Evaluating content management techniques for web proxy caches. *Performance Evaluation Review 27*, 4 (March 2000), 3–11.

[2] ARLITT, M. F., AND WILLIAMSON, C. L. Web server workload characterization: The search for invariants. In *Proc. of the ACM SIGMETRICS 1996 Intl. Conf. on Measurement and Modeling of Computing Systems* (July 1996), ACM SIGMETRICS, pp. 126–137.

[3] BUCK, D., AND SINGHA, M. An analytic study of caching in computer-systems. *Journal of Parallel and Distributed Computing 32*, 2 (February 1996), 205–214. (Erratum published in 34(2):233, May 1996).

[4] CAO, P., FELTEN, E. W., AND LI, K. Application-controlled file caching policies. In *Proc. of the USENIX Association Summer Conf.* (June 1994), USENIX Association, pp. 171–182.

[5] CROVELLA, M. E., AND BESTAVROS, A. Self-similarity in world wide web traffic evidence and possible causes. In *Proc. of the ACM SIGMETRICS 1996 Intl. Conf. on Measurement and Modeling of Computing Systems* (July 1996), ACM SIGMETRICS, pp. 160–169.

[6] DAHLIN, M. D., WANG, R. Y., ANDERSON, T. E., AND PATTERSON, D. A. Cooperative caching: Using remote client memory to improve file system performance. In *Proc. of the First Symp. on Operating Systems Design and Implementation* (November 1994), USENIX Association and ACM SIGOPS, pp. 267–280.

[7] EFFELSBERG, W., AND HAERDER, T. Principles of database buffer management. *ACM Transactions on Database Systems 9*, 4 (December 1984), 560–595.

[8] FEELEY, M. J., MORGAN, W. E., PIGHIN, F. H., KARLIN, A. R., , AND LEVY, H. M. Implementing global memory management in a workstation cluster. In *Proc. of the 15th Symp. on Operating Systems Principles* (December 1995), ACM SIGOPS, pp. 201–212.

[9] FRANKLIN, M. J., CAREY, M. J., AND LIVNY, M. Global memory management in client-server DBMS architectures. In *Proc. of the 18th Very Large Database Conf.* (August 1992), VLDB Endowment, pp. 596–609.

[10] FROESE, K., AND BUNT, R. B. The effect of client caching on file server workloads. In *Proc. of the29th Hawaii International Conference on System Sciences* (January 1996).

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

[12] JIN, S., AND BESTAVROS, A. Popularity-aware greedy-dual-size web proxy caching algorithms. In *Proc. of the 20th Intl. Conf. on Distributed Computing Systems* (April 2000), IEEE.

[13] KAREDLA, R., LOVE, J. S., AND WHERRY, B. G. Caching strategies to improve disk performance. *IEEE Computer 27*, 3 (March 1994), 38–46.

[14] KORNER, K. Intelligent caching for remote file service. In *Proc. of the 10th Intl. Conf. on Distributed Computing Systems* (May 1990), IEEE, pp. 220–226.

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

[16] MENON, J., AND HARTUNG, M. The IBM 3990 disk cache. In *Proc. of COMPCON 1988* (June 1988), pp. 146–151.

[17] MILLER, D. W., AND HARPER, D. T. Performance analysis of disk cache write policies. *Microprocessors and Microsystems 19*, 3 (April 1995), 121–130.

[18] MUNTZ, D., AND HONEYMAN, P. Multi-level caching in distributed file systems — or — your cache ain't nuthin' but trash. In *Proc. of the USENIX Association Winter Conf.* (January 1992), USENIX Association.

[19] RAHM, E., AND FERGUSON, D. F. Cache management algorithms for sequential data access. Research Report RC15486, IBM T.J. Watson Research Laboratories, Yorktown Heights, NY, 1993.

[20] ROBINSON, J. T., AND DEVARAKONDA, M. V. Data cache management using frequency-based replacement. In *Proc. of the ACM SIGMETRICS 1990 Intl. Conf. on Measurement and Modeling of Computing Systems* (May 1990), ACM SIGMETRICS, pp. 132–142.

[21] RUEMMLER, C., AND WILKES, J. A trace-driven analysis of disk working set sizes. Technical Report HPL-OSR-93-23, HP Laboratories, Palo Alto, CA, April 1993.

[22] RUEMMLER, C., AND WILKES, J. UNIX disk access patterns. In *Proc. of the USENIX Association Winter Conf.* (January 1993), USENIX Association, pp. 405–420.

[23] RUEMMLER, C., AND WILKES, J. An introduction to disk drive modelling. *IEEE Computer 27*, 3 (March 1994), 17–28.

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

[25] SMITH, A. J. Disk cache-miss ratio analysis and design considerations. *ACM Transactions on Computer Systems 3*, 3 (August 1985), 161–203.

[26] TENG, J. Z., AND GUMAER, R. A. Managing IBM Database 2 buffers to maximize performance. *IBM Systems Journal 23*, 2 (1984), 211–218.

[27] TRANSACTION PROCESSING COUNCIL. TPC benchmark C, Standard Specification Revision 3.5. `http://www.tpc.org/cspec.html`, October 1999. Accessed April 2000.

[28] TREIBER, K., AND MENON, J. Simulation study of cached RAID5 designs. In *Proc. of the First Conf. on High-Performance Computer Architecture* (January 1995), IEEE, IEEE Computer Society Press, pp. 186–197.

[29] UYSAL, M., ACHARYA, A., AND SALTZ, J. Requirements of I/O systems for parallel machines: An application-driven study. Technical Report CS-TR-3802, Dept. of Computer Science, University of Maryland, College Park, MD, May 1997.

[30] VARMA, A., AND JACOBSON, Q. Destage algorithms for disk arrays with nonvolatile caches. In *Proc. of the 22nd Annual Intl. Symp. on Computer Architecture* (June 1995), Assocation for Computing Machinery, pp. 83–95.

[31] VOELKER, G. M., ANDERSON, E. J., KIMBREL, T., FEELEY, M. J., CHASE, J. S., AND KARLIN, A. R. Implementing cooperative prefetching and caching in a globally managed memory system. In *Proc. of the ACM SIGMETRICS 1998 Intl. Conf. on Measurement and Modeling of Computing Systems* (June 1998), ACM SIGMETRICS, pp. 33–43.

[32] VOIGT, D. HP AutoRAID field performance. Presentation 3354 at HP World, August 1998.

[33] WILKES, J. The Pantheon storage-system simulator. Technical Report HPL-SSP-95-14 rev. 1, HP Laboratories, Palo Alto, CA, May 1996.

[34] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems 14*, 1 (February 1996), 108–136.

[35] WILLICK, D. L., EAGER, D. L., AND BUNT, R. B. Disk cache replacement policies for network fileservers. In *Proc. of the 13th Intl. Conf. on Distributed Computing Systems* (May 1993), IEEE, pp. 2–11.

[36] WOLMAN, A., VOELKER, G. M., SHARMA, N., CARDWELL, N., KARLIN, A., AND LEVY, H. M. The scale and performance of cooperative web proxy caching. In *Proc. of the 17th Symp. on Operating Systems Principles* (December 1999), ACM SIGOPS, pp. 16–31.

[37] ZIPF, G. K. *Human behavior and principle of least effort*. Addison-Wesley Press, Cambridge, MA, 1949.