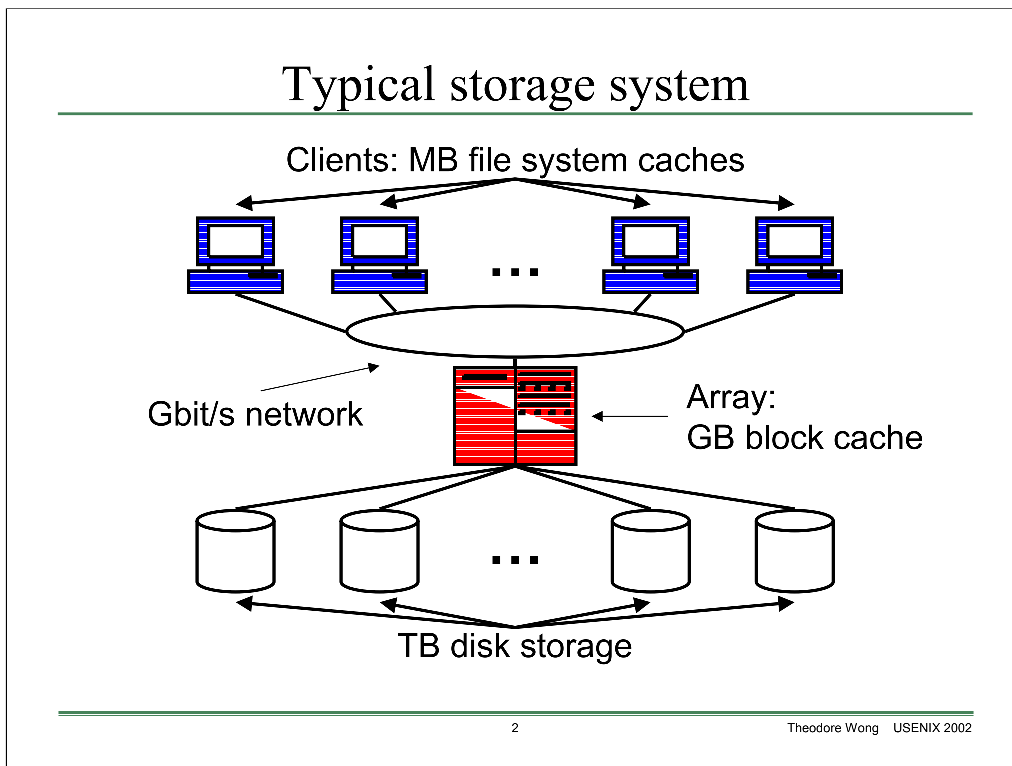
My cache or yours?
Making storage more exclusive

Theodore Wong (Carnegie Mellon University)
John Wilkes (HP Labs)

USENIX 2002



Let's begin by considering a typical storage system.

At the top, you have front-end clients, which typically have hundreds of megabytes of file system cache RAM.

At the bottom, you have a back-end storage array, which has gigabytes of cache RAM.

The array sits in front of a disk farm with terabytes of long-term storage.

Finally, you have a fast network connecting the clients to the array. The network latency is orders of magnitude lower than the disk latency, to the point where you can think of the array cache as being a low-cost extension of the client caches.

(Cache sizes have increased since we began our study, but the relative sizes of the client and array are about right.)

Now, you might expect that the total effective cache is equal to the aggregate of the client and array caches. Unfortunately, that is usually not the case.

Motivation

> "Your cache ain't nuthin' but trash."
> *–Muntz and Honeyman*

- Array cache is *inclusive*
  - Blocks duplicated in the client and array
- We make the array cache *exclusive*
  - Blocks either at the client or array

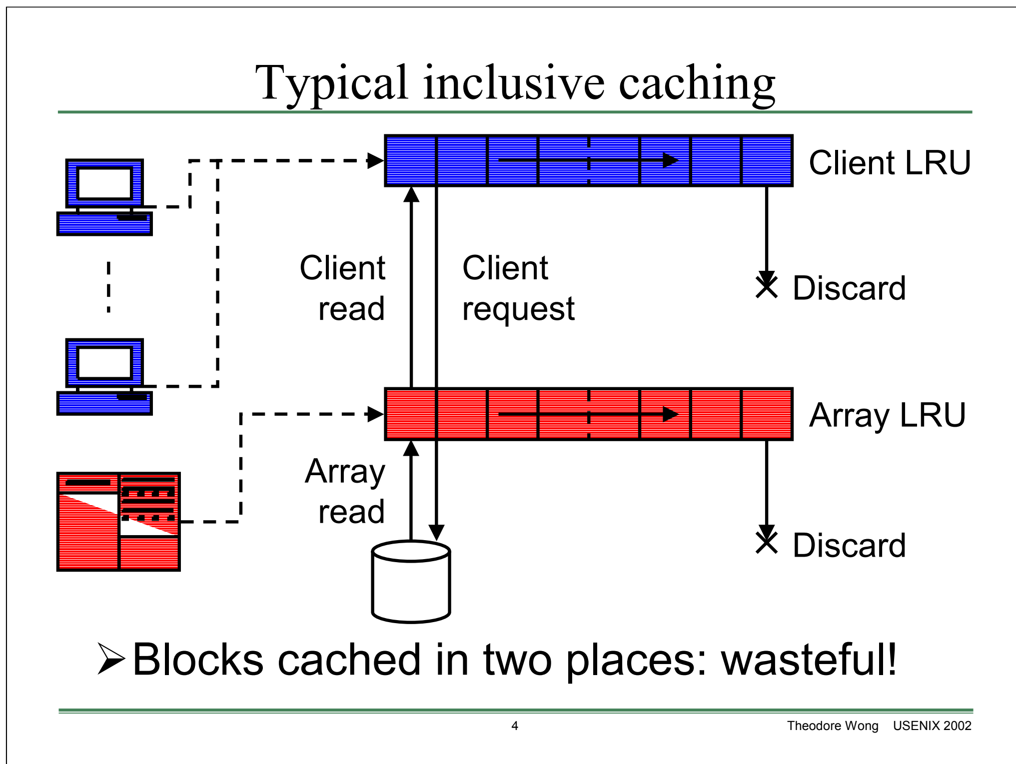
As Muntz and Honeyman observed, "your cache ain't nuthin' but trash!".

The problem is that the array cache is typically inclusive, which means that much of its contents duplicates that of the clients.

Inclusive caching reduces the effective size of the aggregate cache, which is wasteful since array cache isn't cheap.

Inclusive caching hurts performance, since more clients requests may go out to the disk than is necessary.

What we have done is design and evaluate schemes that make the array cache exclusive, which means that its contents is distinct from that of the client.



Let’s simplify the previous picture a bit, by ignoring inter-client sharing, and taking all of the client caches and thinking of them as a single, large cache.

Even thought each individual client cache is smaller than the array cache, in aggregate the client cache may be about as large as the array cache.

The boxes on the right are simply LRU caches: we put data blocks in at the tail, and eject them from the head.

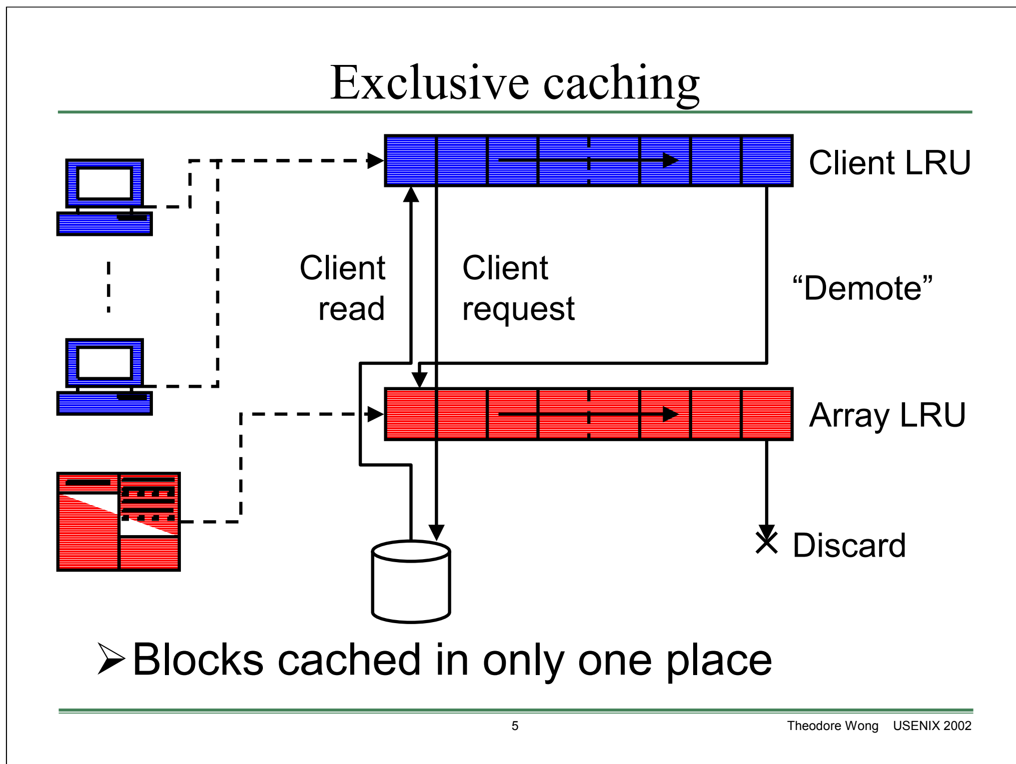
Now, consider what happens when the client issues a request for a block that isn’t in any of the caches.

First, the array reads the block off of a disk, and puts it into its cache.

Then, the client reads the block out of the array, and puts in into its cache.

Now, the block will be duplicated in both caches until the client issues enough requests to push the block out of one of the caches.

This duplication is wasteful: if the client doesn’t have a block in its cache, the array is likely not to have it either. If the client does have the block, it doesn’t matter if the array has it or not.



Let's return to our abstract storage system.

Again, consider what happens when the client issues a request for a block that isn't in any of the caches.

Now, the client reads the block straight to the tail of its cache, basically bypassing the array.

The array cache only acts as a buffer between the disk and the network. We'll come back to this point later.

When the block reaches the head of the client cache, the client uses a new demote operation to transfer the block to the tail of the array cache. The array discards the block when it reaches the head of its cache.

Excellent. We have stopped the duplication of blocks, and made the array cache into an extension of the client cache.

Outline

- Single-client evaluation
- Multi-client evaluation
- Conclusions

For the rest of this talk, I’ll present results of evaluations of our exclusive caching scheme using both simple and complex models of storage hierarchies.

I’ll first present the results of simple experiments that use our abstract, single-client model.

Then, I’ll present the results of experiments with multiple clients and realistic workloads.

Single-client evaluation

- Verify that exclusive caching works
- Study caching schemes in simple systems:
  - Single client cache, equal in size to array cache
  - Read-only workloads
- Analyze sensitivity to:
  - Reduced bandwidth
  - Larger and smaller array cache sizes

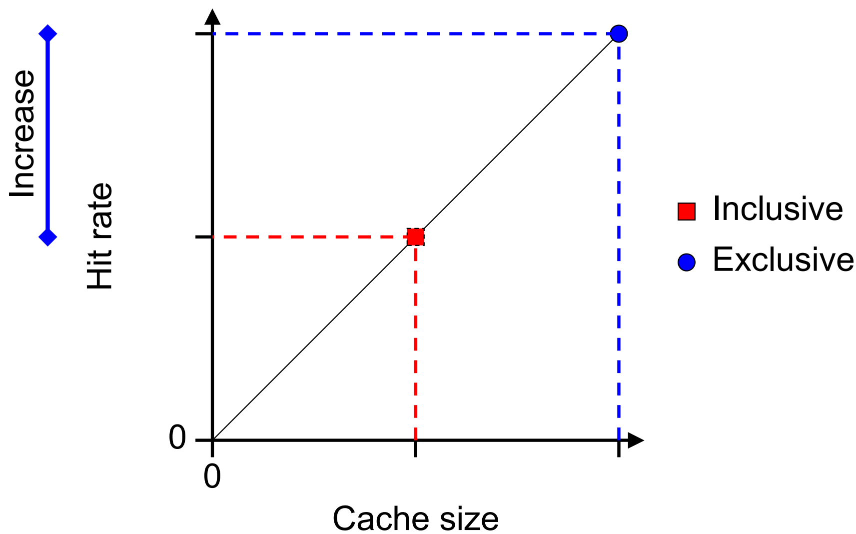
The primary goal of the single-client evaluation is to confirm that our reasoning about exclusive caching schemes is sound, and that we have the potential to obtain useful performance improvements.

To simplify reasoning about cache behavior, we consider the behavior of schemes in the abstract system with a single large client cache, equal in size to the array cache.

We consider read-only workloads throughout this study. For many workloads, write requests only form a small fraction of all requests. Also, the type of arrays we are modeling generally have a separate, non-volatile write cache.

Since our design relies on the existence of a high-bandwidth, low-latency network connecting the client and the array, we analyzed how our exclusive scheme performed under reduced bandwidth.

We also analyzed how our scheme performed when the array cache was larger or smaller than the client cache.



We can estimate the benefit of switching from an inclusive to an exclusive cache scheme by looking at the graphs of the hit rate vs. cache size for each workload.

This graph represents the hit rate for a random workload, in which hit rate rises linearly with cache size.

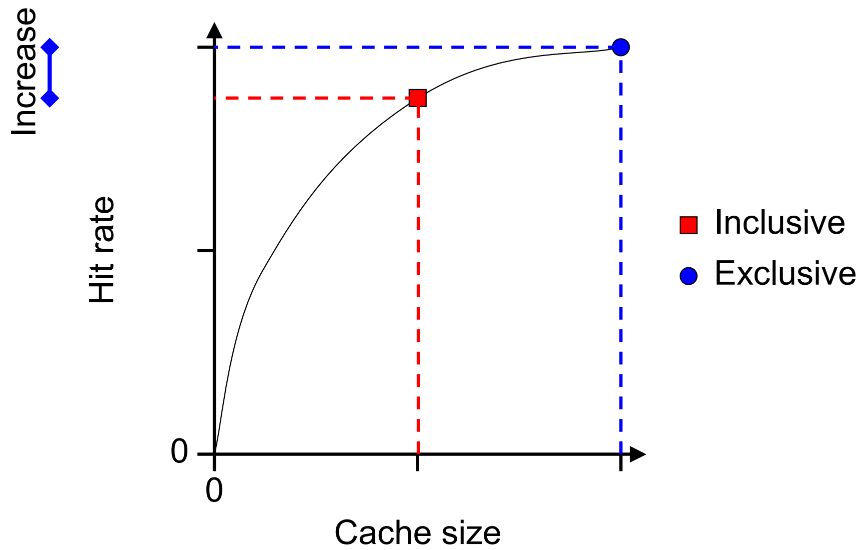
For the abstract single-client setup, we expect that an array with an inclusive scheme will contain the same data as the client, and all requests that miss at the client will also miss in the array.

This red line here show the hit rate in the client.

Conversely, we expect an array with an exclusive scheme to contain different data than the client. We just sum up their cache sizes to find the cumulative hit rate.

Since the first set of requests hit in the client, the remaining requests must hit in the array. This marker bar here represents the increase in the overall hit rate due to hits in the array.

Predicting benefits: Zipf-like



We can perform a similar analysis for Zipf-like workloads, in which the probability of accessing block i is proportional to $1/i$. We see this kind of access pattern in file and web server systems.

We see that there's still an improvement with exclusive caching schemes, but the benefit is smaller since most requests already hit in the client.

Single-client workloads: Synthetic

- RANDOM (e.g., transaction processing)
- ZIPF (e.g., web server; file server)

- Simulated in Pantheon

For our initial single-client experiments, we constructed a set of synthetic workloads that we predicted should perform poorly with inclusive schemes, but quite well with exclusive schemes. Again, I stress that we did this to ensure the sanity of our design and theoretical understanding.

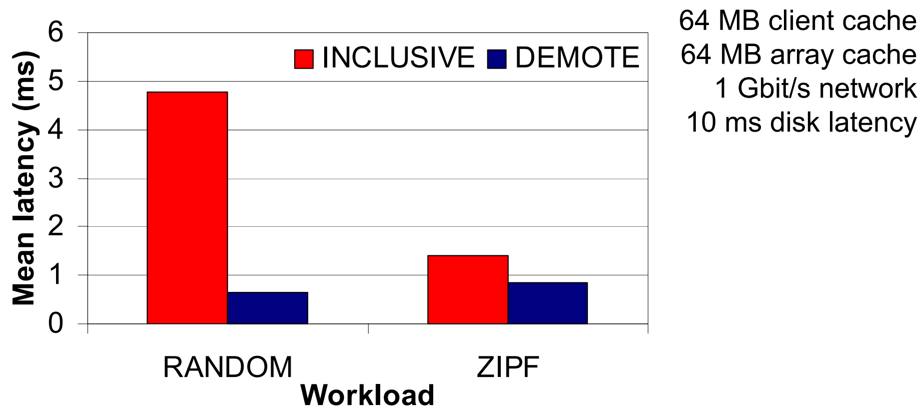
We used two synthetic workloads that roughly model real workloads: RANDOM, which is similar to transaction-processing workloads, and ZIPF, which models Unix file server and web server access patterns.

We simulated our caching schemes in Pantheon, a detailed array simulator from HP that models just about everything about in a disk array, including accurate disk timing and network controller overheads.

Single-client schemes

- INCLUSIVE
  - Baseline “typical” scheme
- DEMOTE
  - Exclusive caching scheme

Single-client results: Synthetic



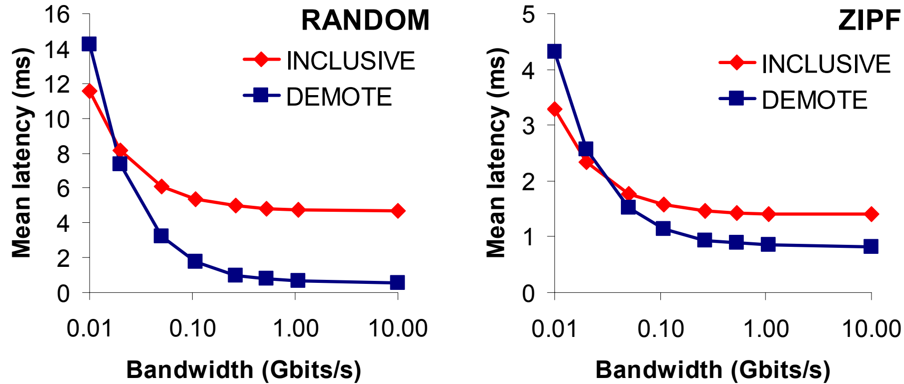
- Exclusive caching works

Our results show that our expectations about the DEMOTE caching scheme were correct. We obtained impressive speedups for both synthetic workloads.

We report mean request latency as seen by the client. Lower is better. Blue bars are us.

ZIPF shows an interesting result: with ZIPF, we would expect the client to hold the most popular blocks, and request other blocks relatively infrequently. Even so, we still obtain a noticeable speedup with the DEMOTE scheme.

Sensitivity evaluation results: Network



- Resilient to bandwidth variation

Having confirmed that our synthetic workloads obtained speedups from exclusive caching, we go on to see how our exclusive scheme stacks up when we reduce the available network bandwidth.

And we see that we do pretty well. Again, these graphs show mean request latency. Lower is better. Blue is us.

Our scheme holds up well except for very low bandwidths. At the crossover point, the extra network transfer cost of demotions outweighs any benefits obtained.

Better yet, the performance degradation is graceful. There are no cliffs.

Single-client workloads: Real

- CELLO99: File server
- TPC-H: Database server benchmark
- DB2: Multi-client database workload
- HTTPD: Web server farm

- Simulated in fscachesim

Since our synthetic workloads obtained impressive speedups from exclusive caching, we went on to see how more realistic workloads would fare. For each workload, we used client and array cache sizes that were of the same magnitude as the working set of the load.

CELLO99 is a month-long trace of a file server.

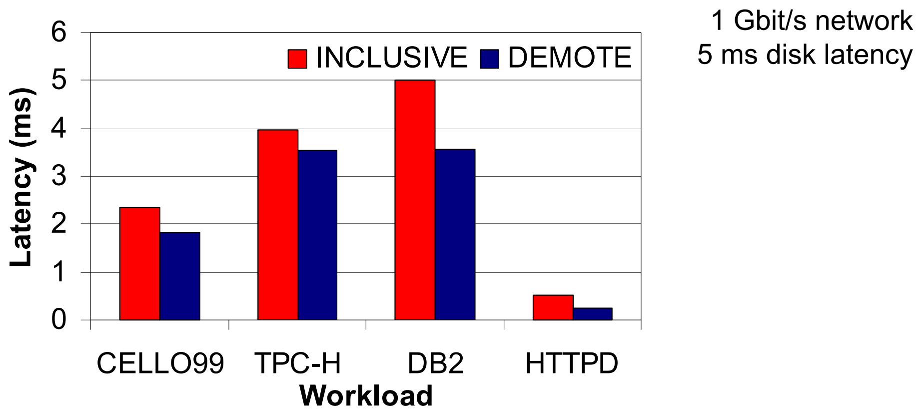
TPC-H is a portion of an audited run of the TPC-H transaction processing benchmark.

DB2 is a trace of an eight-node parallel database system performing join, set, and aggregation operations.

HTTPD is a one-hour trace of a seven-node parallel web server.

Since we were primarily concerned with cache behavior only, and since Pantheon couldn't simulate the larger caches required by these workloads, we used a simpler cache simulator called Pantheon that omitted the detailed disk and network models.

Single-client results: Real

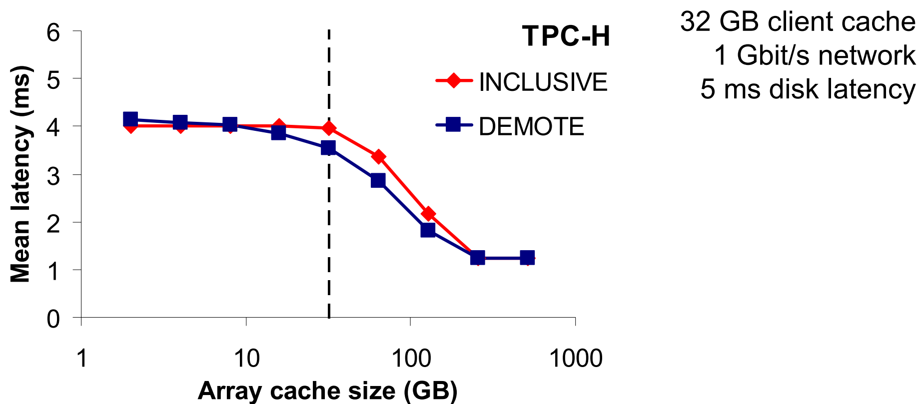


- Exclusive caching works for real loads

Our results demonstrate that our DEMOTE caching scheme provides benefits to real-life workloads. We used faster disk models than those used in the synthetic experiments, commensurate with those that existed in the original traced systems.

HTTPD shows an interesting result: even though the client contains most of the working set, as evidenced by the low mean latency with the INCLUSIVE scheme, switching to the DEMOTE scheme still yields a speedup.

Sensitivity evaluation results: Cache size



- Resilient to array cache size variation

16 Theodore Wong USENIX 2002

By now, you’re probably wondering what happens if our assumption about equal client and array cache sizes is bogus.

Here, we show the effect of varying array cache size for TPC-H while keeping the client cache size constant. Lower is better. Blue is us. The dotted line shows the point where the client and array caches sizes are the same.

The answer is that our exclusive scheme still performs OK. We only start to get into trouble when the array cache is an order of magnitude smaller than the client cache, and of course we get no improvement if the working set fits in the client cache alone.

Single-client summary

Workload	Speedup
RANDOM	7.5
ZIPF	1.7

Workload	Speedup
CELLO99	1.3
TPC-H	1.1
DB2	1.4
HTTPD	2.2

- Exclusive caching yields significant speedups
- Resilient to bandwidth, cache size variation

Overall, we get up to a 7.5 times speedup with our DEMOTE exclusive caching scheme for all of the single-client workloads.

Also, the benefits of exclusive caching are resilient to bandwidth and array cache size variations.

Multi-client evaluation

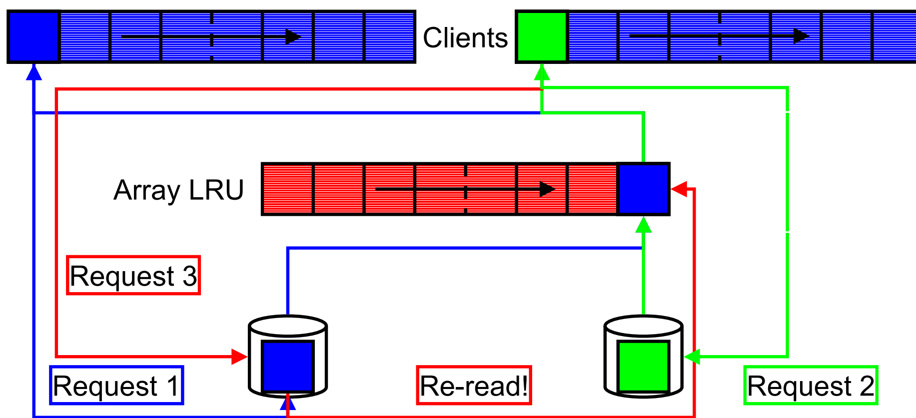
- Predict benefits for real systems
  - Large array cache, smaller client caches
- Consider effects of inter-client sharing
- Define two types of workload:
  - Disjoint workloads: No block sharing
  - Shared workloads: Some block sharing

We now move on to multi-client evaluation. Having seen that realistic workloads obtain speedups in the single-client case, we decided to evaluate the performance of exclusive caching schemes in realistic system environments with multiple request streams.

Multi-client systems have a new variable: the degree to which clients request the same blocks from storage.

It helps to divide multi-client workloads into two families: disjoint workloads in which client do not access any of the same blocks, and shared workloads in which clients access some of the same blocks in the working set.

Shared workloads and DEMOTE

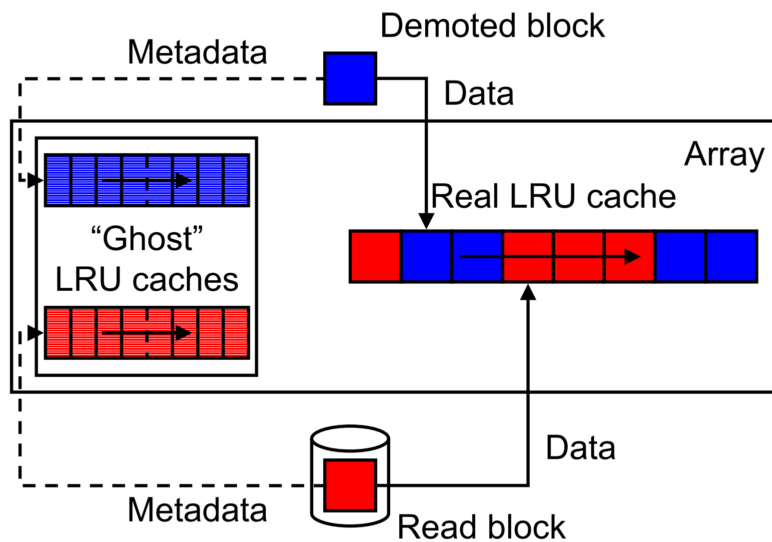


- Need to save "disk-read" blocks at array

For shared workloads, our DEMOTE protocol may actually hurt performance, by overly aggressively discarding blocks read from the disk.

Consider this simple example.

[…]



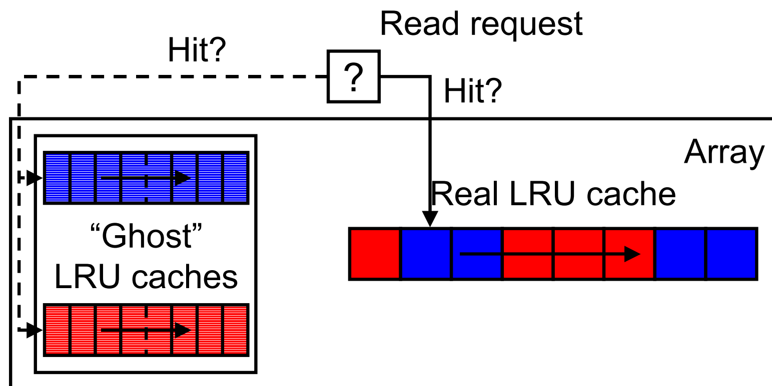
To determine how long to save read blocks in the array cache, we developed an adaptive extension to our exclusive caching scheme. Our extension simulates the effect of caching only read or demoted blocks at the array, by using ghost caches.

Here's how ghost caches work. Suppose that we have read a block from disk. We first take the block metadata and cache it in an LRU-style ghost cache that simulates what would happen if we only cached read blocks. Then, we insert the actual block data into the real array cache at a position determined by a score value I'll introduce soon.

We treat demoted blocks in a similar manner.

Once we insert a block into the real cache, it is moved around just as in a standard LRU cache: if a subsequent block request hits in the cache, we move the block to the tail. When the block reaches the head, we discard it.

Adaptive caching: Receiving requests



Now, when we receive a read request at the array, we first check to see if the request hits in our ghost caches, and update the LRU queues and hit counts as appropriate: in other words, we see if we would have gotten a hit if we had cached only blocks of one type.

We then see if the request hits in the real cache.

Insertions with scores

- Read block insertion:
  - Score = read ghost hits / sum of ghost hits
  - 0 → real cache head, 1 → real cache tail
- Demoted block insertion: similar to read

➢ Need to make insertions fast

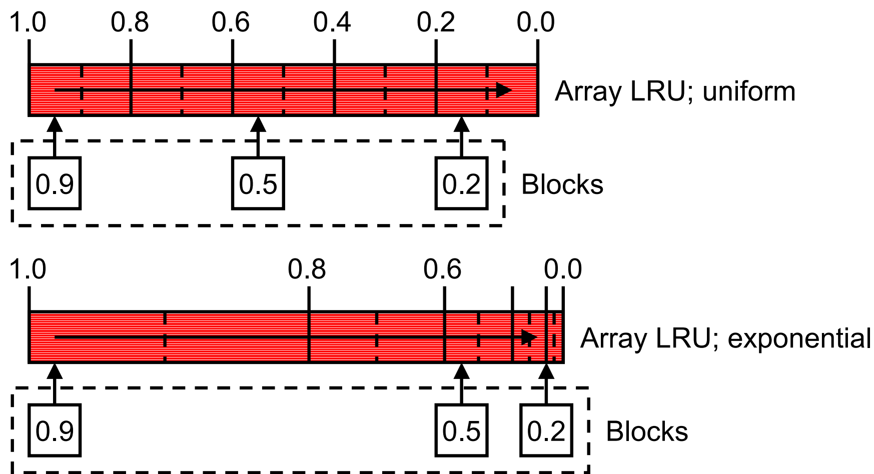
We use the relative hit counts of the ghost caches to compute a score. The score is then used to determine where to insert a block into the real cache.

As an example, consider the insertion of a block read from the disk. We compute the score by dividing the read ghost hit count by the sum of all ghost hit counts.

We then insert the read block into the real cache. A score of 0 means that read blocks are not useful, and we insert the block at the head of the real cache. A score of 1 means that read blocks are incredibly useful, and we insert the block at the tail.

The insertion of a demoted block is similar to a read, except that we use the demoted ghost hit count instead.

Locating the exact insertion position for a block can be time-consuming. We need a mechanism to make insertions fast.



To make insertions fast, we divide the cache into segments, and assign a maximum score to each segment.

Then, when computing the score for a block to insert, we round up to the nearest maximum segment score.Thus, a block with a 0.9 score goes to the tail of the 1.0 segment. A block with a 0.5 score goes into tail of the 0.6 segment, and a block with a 0.2 score goes into, well, the 0.2 segment.

We may decide to weight segment sizes to reward high scores and penalize low scores. We experimented with an exponential weighting, in which each segment was twice the size of the previous segment, with the largest segment at the tail.

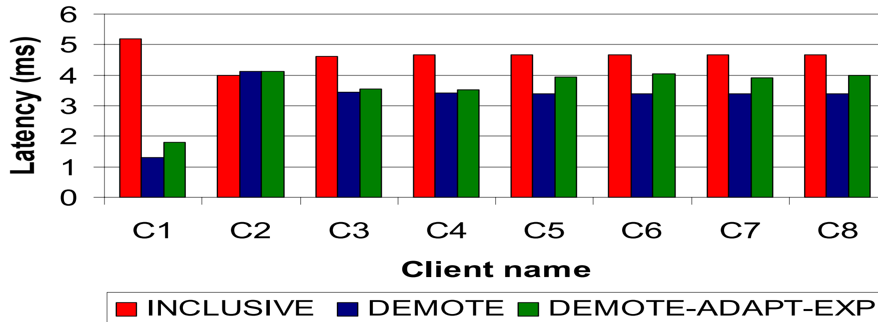
Multi-client workloads

- Disjoint:
  - DB2 (8 hosts): As before
  - OPENMAIL (6 hosts): Mail server farm
- Shared:
  - HTTPD (7 hosts): As before
- Simulated in fscachesim

Multi-client schemes

- INCLUSIVE
  - Baseline
- DEMOTE
- DEMOTE-ADAPT-EXP
  - Adaptive caching, exponential segments

Multi-client results: DB2



- DEMOTE for disjoint workloads:
  - Keep demoted blocks
  - Eject disk-read blocks

On this graph, we show the per-client request latency. Lower is better. Red is the baseline, blue and green are DEMOTE and DEMOTE-ADAPT-EXP respectively.

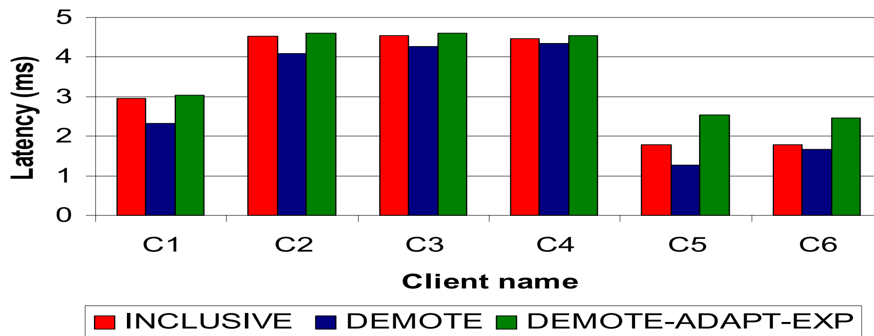
Since DB2 is disjoint, read blocks are never reused, so the array should discard them immediately.

Thus, for the most part, all of the clients get speedups out of DEMOTE, which discards read blocks immediately.

Clients get smaller speedups out of DEMOTE-ADAPT-EXP, which wastes array cache space by keeping read blocks around for a little while. Perhaps, with a longer trace, the array would learn to discard read blocks immediately.

A blip: client 2 has a small working set that mostly fits in the client cache alone. Thus, demoting blocks is pointless.

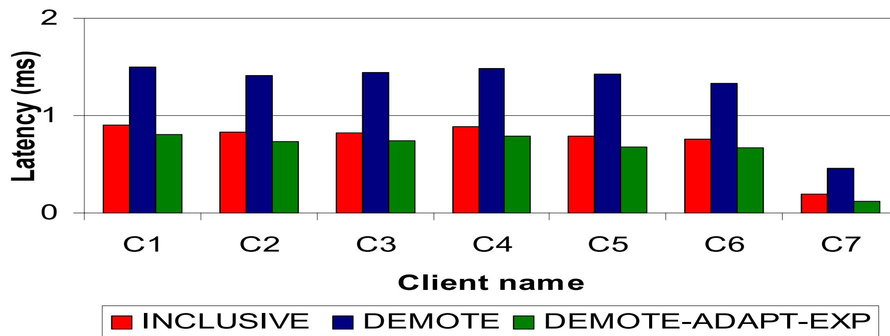
Multi-client results: OPENMAIL



➢ Again, DEMOTE for disjoint workloads:
- Keep demoted blocks
- Eject disk-read blocks

OPENMAIL shows the similar results to DB2, again confirming that the array should immediately discard read blocks for disjoint workloads.

Multi-client results: HTTPD



- DEMOTE-ADAPT-EXP for shared workloads:
  - Keep both demoted and disk-read blocks

HTTPD shows the opposite behavior from DB2 and OPENMAIL. DEMOTE discards shared blocks before other clients have the chance to request them, which causes those request to go to disk.

DEMOTE-ADAPT-EXP keeps those read blocks around, thus letting other clients get at them.

A blip: client 7 had a smaller working set than the others.

Multi-client summary

Workload	Mean per-client speedup	
	DEMOTE	DEM-ADAPT-EXP
DB2	**1.5**	1.3
OPENMAIL	**1.2**	0.9
HTTPD	0.6	**1.2**

- Eject read blocks for disjoint workloads (DB2, OPENMAIL)
- Keep some read blocks for shared workloads (HTTPD)

Related work

- Inclusive caching
  [Muntz1992, Froese1996]
- Global memory management:
  - Database system cache management
    [Franklin1992]
  - Peer-to-peer cooperative caching
    [Dahlin1994, Feeley1995]
- Per-workload cache management policies

Long history of cache research going back over thirty years. These are the highlights that are most relevant to our work.

Muntz1992 identified the problem of inclusive caching in his “trash” paper. Froese1996 et al also looked at inclusive caching, and suggested different cache management policies to cope with this. No demotions, though.

Several research groups have looked at cooperative caching for databases and file systems. Franklin et al built a system for global memory management where a central server keeps a directory of blocks in the global (client and array) memory space, and redirected requests to whoever held the block. Peer-to-peer cooperative caching work has looked at removing the central server, and maintaining the directory in a distributed fashion. Our solution requires no central directory, and in fact makes fewer assumptions about the capabilities of the clients.

And of course, several research groups (too many to enumerate here) have looked changing cache management policies at the clients and array on to match the expected workload – but with no additional communication between clients and array. Our protocol is another type of policy, but with a simple additional operation to exchange some explicit information between them.

Conclusions

"My cache OR your cache?"
–Wong and Wilkes

- Exclusive caching beats inclusive
- Simple demote op yields big speedups

In conclusion, storage systems should never be afraid to ask, "my cache or yours?".

More information

- My research page:
  - http://www.cs.cmu.edu/~tmwong/research/
- HPL Storage Systems Program:
  - http://www.hpl.hp.com/SSP/