On Multi-level Exclusive Caching: Offline Optimality and Why promotions are better than demotions.

Binny S. Gill
IBM Almaden Research Center
binnyg@us.ibm.com

Abstract— Multi-level cache hierarchies have become very common; however, most cache management policies result in duplicating the same data redundantly on multiple levels. The state-of-the-art exclusive caching techniques used to mitigate this wastage in multi-level cache hierarchies are the DEMOTE technique and its variants. While these achieve good hit ratios, they suffer from significant I/O and computational overheads, making them unsuitable for deployment in real-life systems.

We propose a dramatically better performing alternative called PROMOTE, which provides exclusive caching in multi-level cache hierarchies without demotions or any of the overheads inherent in DEMOTE. PROMOTE uses an adaptive probabilistic filtering technique to decide which pages to "promote" to caches closer to the application. While both DEMOTE and PROMOTE provide the same aggregate hit ratios, PROMOTE achieves more hits in the highest cache levels leading to better response times. When inter-cache bandwidth is limited, PROMOTE convincingly outperforms DEMOTE by being 2x more efficient in bandwidth usage. For example, in a trace from a real-life scenario, PROMOTE provided an average response time of 3.42ms as compared to 5.05ms for DEMOTE on a two-level hierarchy of LRU caches, and 5.93ms as compared to 7.57ms on a three-level cache hierarchy.

We also discover theoretical bounds for optimal multi-level cache performance. We devise two offline policies, called OPT-UB and OPT-LB, that provably serve as upper and lower bounds on the theoretically optimal performance of multi-level cache hierarchies. For a series of experiments on a wide gamut of traces and cache sizes OPT-UB and OPT-LB ran within 2.18% and 2.83% of each other for two-cache and three-cache hierarchies, respectively. These close bounds will help evaluate algorithms and guide future improvements in the field of multi-level caching.

I. INTRODUCTION

Very rarely does data reach its consumer without traveling through a cache. The performance benefit of a cache is significant, and even though caches are much more expensive than mass storage media like disks, nearly all data servers, web servers, databases, and in fact most computing devices are equipped with a cache. In the last several decades numerous read caching algorithms have been devised (for example, LRU[10], CLOCK[8], FBR[29], 2Q[20], LRFU[21], LIRS[18], MQ[36], ARC[25] and SARC[14]). Most of the work, however, has focused on the case when a single significant layer of cache separates the data producer and the data consumer. In practice, however, data travels through multiple layers of cache before reaching an application. It has been observed that single-level cache replacement policies perform very poorly when used in multi-level caches [26].

We propose a simple and universal probabilistic technique, called PROMOTE, that adapts any single-level read caching algorithm into an algorithm that allows a multi-level read cache hierarchy to perform as effectively as a single cache of the aggregate size. Unlike previous algorithms, this novel algorithm imposes negligible computational and I/O overheads. As another key contribution to the field of multi-level caching, we provide, for the first time, techniques to compute very close bounds for the notoriously elusive offline optimal performance of multi-level caches.

A. The Problem with Inclusion

One of the earliest examples of multi-level caches arose in the context of a processor [30], [28]. The multiple layers of cache were named L1, L2, L3, etc., with L1 (highest cache) being the closest to the processor, the smallest in size, and the fastest in response time. For efficient cache coherency, systems enforced the inclusion property [1], which mandated that the higher caches be a subset of the lower caches. Other than in cases where L2 was only marginally larger than L1 [22], the performance penalty of this redundancy of content between the layers was not of much concern.

In stark contrast, in the hierarchy of caches formed by multiple computing systems connected to each other, inclusion, sometimes partial, is not by design and has been found to be detrimental to performance [26], [13]. The redundancy of data between the cache levels is most severe when the caches are of comparable sizes. A request is always serviced from the closest cache to the client that has the data, while further copies of the data in lower caches are not useful.

B. Working towards Exclusion

In cache hierarchies through which pages traverse fixed paths from the data source to the application, exclusivity of all caches is highly desirable. In *multi-path* cache hierarchies, where pages can get accessed via

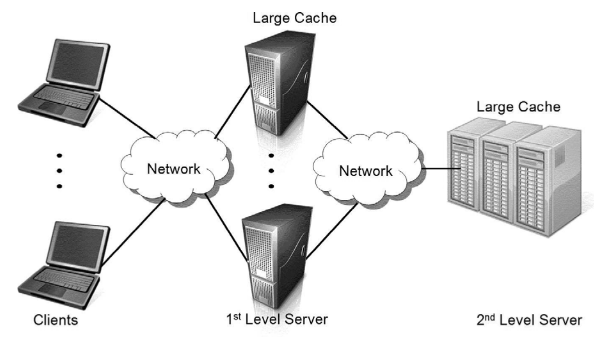


Fig. 1. Cache hierarchies that benefit from exclusive caching.

multiple paths, exclusive caching should be applied to those portions of the cache hierarchy which do not have a large workload overlap between the various paths.

Figure 1 shows a common scenario for which various exclusive caching algorithms have been proposed ([36], [33], [19], [12]). It is fairly common to find cache hierarchies formed by a first level of application servers (database servers, web proxy servers, web content delivery servers, storage virtualization servers) which act as clients for backend storage servers, while both are equipped with significant and comparable caches [33]. In some cases, it may also be possible to include the end clients to form an exclusive cache hierarchy of three or more levels of cache.

A naïve way of enforcing the exclusion property would be to associate different caches with different logical addresses. Obviously, this will not be able to gather frequently hit pages in the topmost caches and the average response time for some workloads might end up worse than the case when the caches are not exclusive at all. Succinctly, the challenge of multi-level exclusive caching is: *"Exclusivity achieved efficiently with most hits at the highest cache levels"*.

C. Exclusivity via Smart Lower Caches

It has been known that the Least Frequently Used ([13], [32]) algorithm performs better at second-level caches that the traditional LRU algorithm. A more sophisticated second-level cache management algorithm is the MQ algorithm [36] which maintains multiple lists geared to capture frequently accessed pages with long reuse intervals. However, it is not studied for more than two levels of cache and also cannot achieve complete exclusivity of the caches where desirable.

A more recent algorithm, X-RAY [2], constructs in the RAID controller cache an approximate image of the contents of the filesystem cache by monitoring the metadata updates, which allows for better cache replacement decisions and exclusivity. Such gray-box approaches, however, are domain-specific and not easily applicable to more than two levels of cache.

A similar approach is to use a Client Cache Tracking (CCT) table [6] in the lower cache to simulate the contents of the higher cache. This allows the lower cache to proactively reload the evicted pages from the storage media. The extra cost of these requests, however, may overburden the storage media resulting in high read response times.

D. Exclusivity via Smart Higher Caches

Chen et al. [7] propose an algorithm called AC_{CA} in which a client (higher cache) simulates the contents of the server (lower cache), and uses that knowledge to preferably evict those pages which are also present in the server. This is difficult to do in multi-client scenarios or where the server behavior is complex or proprietary.

E. Exclusivity via Multi-level Collaboration

When multiple layers of cache can be modified to communicate with each other, most of the guesswork of simulating cache contents can be avoided. Even though extensions are required to the communication protocol, this class has proven to be the most promising in terms of performance and ease of implementation.

1) Application controlled: The Unified and Level-aware Caching (ULC) algorithm [19] controls a cache hierarchy from the highest level by issuing RETRIEVE and DEMOTE commands to lower caches causing them to move blocks up and down the hierarchy, respectively. The highest cache level (application client) has to keep track of the contents of all the caches below, which entails complexity in the client's implementation.

2) Application hints: KARMA [12] is aimed at applications such as databases that can provide hints for placement decisions in all cache levels. Such solutions are application-specific and do not lend themselves to general applicability in multi-level caches. This brings us to DEMOTE, which is the most popular general-purpose collaborative multi-level caching technique.

F. The DEMOTE Technique

The DEMOTE technique [33], or equivalently the *Global* technique [35], shown in Figure 2, can be applied to many general purpose single-level caching policies (like, LRU, MQ, ARC, etc) to create multi-level versions that achieve exclusivity of cache contents. As with any exclusive caching scheme, DEMOTE should only be used in scenarios that benefit from exclusive caching [33], [27].

G. The Problems with DEMOTE

While the DEMOTE technique strives to improve the aggregate hit ratio over the non-exclusive variant, the overall performance might in fact suffer because

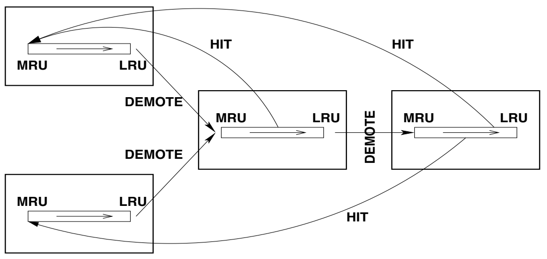


Fig. 2. The DEMOTE technique. When a client is about to eject a clean block from its cache, it sends the clean block to the lower cache using the DEMOTE operation. The lower cache puts the demoted block into its cache, ejecting another block if necessary to make space. Hits in any list are sent to the MRU end of the appropriate highest cache.

of the cost of the DEMOTE operation, including: (i) network traffic to send evicted pages to lower caches, and (ii) processor cycles consumed to prepare, send and receive demoted pages. This has thwarted the practical deployment of DEMOTE in real systems.

In cases where inter-cache bandwidth is tight, or the workload throughput is high, the average response time suffers in spite of a higher hit ratio. This happens as reads stall until demotions (sometimes in multiple levels) can create space for the new page. Further, an eviction, which used to be a trivial operation, now becomes an expensive operation almost equivalent to a write request to the lower cache. This leads to further degradation of performance. Add to this the concern that for workloads that do not exhibit temporal locality, like purely sequential (or purely random), all demotions are futile and we end up wasting critical network resources when most needed.

Eviction-based cache replacement [6] was proposed to alleviate the network bandwidth wastage. A list of evicted pages is sent periodically to the lower cache which reloads them from the disks into its cache. We have observed for many real systems and benchmarks these extra reads increase the average miss penalty, diminishing or defeating the benefits of any increase in hit ratio that such exclusive caching can provide. There have been other attempts to partially mitigate the cost of demotions [34] or the cost of the extra I/Os on the disks [6] by grouping them and using 'idle time'. In our opinion, idle time should never be considered free as it can be used by other algorithms like prefetching to improve read performance. Sophisticated enterprise-class systems which run round the clock strive hard to minimize idle-time. ULC and KARMA do reduce the number of low reuse pages that enter the higher caches and thereby minimizing the bandwidth wastage. However, ULC's complexity and KARMA's dependence on application hints make them less generally applicable.

H. Our Contributions

We present two major results:

Bounds for Optimal Performance: In the study of caching algorithms, it is invaluable to know the offline optimal performance that a multi-level cache can deliver. While Belady's MIN [4] (an offline algorithm) is considered optimal for single-level caches, it cannot be applied to multi-level cache scenarios, where, apart from the hit ratio, the location of hits is also extremely important. Our first contribution provides insight into the optimal offline performance of multi-level caches.

We provide policies called OPT-UB and OPT-LB that provably serve as upper and lower bounds for the optimal offline performance for multi-level caches along both average response time and inter-cache bandwidth usage metrics.

Through a series of experiments on a wide gamut of traces, cache sizes and configurations, we demonstrate that OPT-UB and OPT-LB are very close bounds on the optimal average response time, running on an average, within 2.18% and 2.83% of each other for all the tested two-cache and three-cache hierarchies, respectively. Even for more complex hierarchies, the bounds remain close at about 10% of each other. This novel result enables us to estimate for the first time, the performance gap between the current state-of-the-art algorithms and the offline optimal for multi-level caches.

PROMOTE Technique: As another fundamental contribution to the field of caching, we propose a simple and significantly better alternative to the DEMOTE technique, called PROMOTE, which provides exclusive caching without demotions, application hints, or any of the overheads inherent in DEMOTE. PROMOTE uses a probabilistic filtering technique to "promote" pages to higher caches on a read. Not only do we show that PROMOTE is applicable to a wider range of algorithms and cache hierarchies, it is on an average, 2x more efficient than DEMOTE requiring only half the inter-cache bandwidth between the various cache levels.

In a wide variety of experiments, while both techniques achieved the same aggregate hit ratio, PROMOTE provided 13.0% and 37.5% more hits in the highest cache than DEMOTE when the techniques were applied to LRU and ARC [25] algorithms, respectively, leading to better average response times even when we allow DEMOTE unlimited inter-cache bandwidth and free demotions. In limited bandwidth scenarios, PROMOTE convincingly outperforms DEMOTE. For example, in a trace from a real-life scenario, PROMOTE provided an average response time of 3.21ms as compared to 5.43ms for DEMOTE on a two-level hierarchy of ARC caches, and 5.61ms as compared to 8.04ms on a three-level cache hierarchy.

II. Offline Optimal

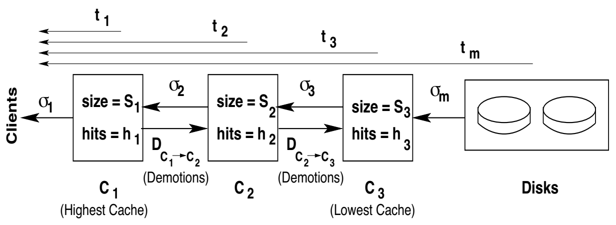


Fig. 3. A multi-level single-path cache hierarchy.

A. Quest for Offline Optimality

The offline optimal performance, which is the best possible performance given full knowledge of the future requests, is a critical guide for cache algorithm development. The offline optimal algorithm for single-level caches is the classic Belady's MIN or OPT [4] which simply evicts the page with the furthest re-reference time. For four decades, the quest for a similar algorithm for multi-level caches has remained unfulfilled. Hit ratio, the performance metric that served us so well in single-level caches, loses its fidelity to performance in multi-level caches, where the benefit of a hit depends on which level it occurred in (e.g. two hits on a higher cache could be better than three hits on a lower cache.)

The average read response time is a much better performance metric, but is complicated in the presence of demotions and/or eviction-based reloading from disks. The inter-cache bandwidth usage is another important metric since many scenarios are bottle-necked by network resources [33]. It does appear that different policies would be optimal depending on which performance metric we use.

We now prove universal bounds for the single-path scenario as depicted in Figure 3 that simultaneously apply to both the average response time and inter-cache bandwidth metrics. The bounds apply to all algorithms irrespective of whether demotions, inclusion, or hints are used. Later, we explain extensions to compute the bounds for the multi-path scenarios as well.

C_i	Cache at level i, $i = 1, 2, ..., n$
S_i	Size of the cache C_i
h_i	Number of hits at C_i
t_i	Avg. round-trip response time from C_i
t_m	Avg. round-trip response time from disks
σ_i	Sequence of reads seen by C_i
σ_m	Sequence of reads (misses) seen by disks
$D_{c_i \to c_{i+1}}$	Number of demotions from C_i to C_{i+1}

From Figure 3 and definitions above, the total response time is

$$totalRespTime = \sum_{i=1}^{n} h_i \cdot t_i + |\sigma_m| \cdot t_m \qquad (1)$$

and the inter-cache traffic between caches C_i and C_{i+1} is

$$interCacheTraf_{c_i \to c_{i+1}} = |\sigma_{i+1}| + |D_{c_i \to c_{i+1}}| \qquad (2)$$

We define $hitOPT(\sigma, S)$ to be the number of hits produced by Belady's offline OPT policy on the request sequence σ and a single-level cache of size S.

B. Optimal Upper Bound: OPT-UB

We define a conceptual policy OPT-UB that serves as an *upper bound on performance*, implying that no policy can perform better than this bound in terms of either average response time or inter-cache bandwidth usage by achieving better (lower) values for either metric. OPT-UB is a bound, not on the performance of a particular cache, but on the aggregate performance of the cache hierarchy.

Let OPT-UB be a policy that for a request sequence σ, exhibits for each cache C_i, h_i number of hits, while requiring no demotions or reloads from disks, where,

$$h_i = hitOPT(\sigma, \sum_{j=1}^{i} S_j) - hitOPT(\sigma, \sum_{j=1}^{i-1} S_j) \qquad (3)$$

Note that we intend to compute a theoretical upper bound which is necessarily achievable.

Lemma II.1 *No policy can have more hits, up to any cache level, than* OPT-UB. *More precisely,* OPT-UB *maximizes* $\sum_{i=1}^{k} h_k, \forall k \leq n$.

Proof: By Eqn. 3 the aggregate hits for the set of caches $C_1, .., C_k$ is

$$aggrHits_k = \sum_{i=1}^{k} h_i = hitOPT(\sigma, \sum_{i=1}^{k} S_i) \qquad (4)$$

By the definition of $hitOPT$, this is the same as that obtained by Belady's OPT on a cache of the aggregate size. Since Belady's OPT is known to deliver the maximum possible hit ratio, $aggrHits_k$ is maximized for all $k \leq n$. ■

Theorem II.1 *No policy can have a lower total inter-cache traffic than* OPT-UB.

Proof: Inter-cache traffic is the sum of misses and demotions between two adjacent caches (by Eqn. 2). Since, OPT-UB is defined to have no demotions and maximizes the aggregate hits at all levels of the cache (Lemma II.1), no other policy can have lower inter-cache traffic than OPT-UB. ■

Theorem II.2 *No policy can have a lower average response time than* OPT-UB.

Proof: We prove by contradiction. Let a better performing policy achieve a lower average response time (equivalently, lower total response time) for a workload than OPT-UB, by providing h'_i hits at each corresponding cache C_i, and m' overall misses, as compared to h_i hits and m misses for OPT-UB.

$$\begin{aligned}
\therefore \quad & \sum_{i=1}^{n} h'_i \cdot t_i + m' \cdot t_m \; < \; \sum_{i=1}^{n} h_i \cdot t_i + m \cdot t_m \\
\Rightarrow \quad & \sum_{i=1}^{n} h'_i \cdot (t_i - t_m) + (m' + \sum_{i=1}^{n} h'_i) \cdot t_m \\
& \quad < \; \sum_{i=1}^{n} h_i \cdot (t_i - t_m) + (m + \sum_{i=1}^{n} h_i) \cdot t_m \\
\Rightarrow^{(a)} \quad & \sum_{i=1}^{n} h'_i \cdot (t_m - t_i) \; > \; \sum_{i=1}^{n} h_i \cdot (t_m - t_i) \\
\Rightarrow \quad & \sum_{i=1}^{n} h'_i \cdot (t_n - t_i + t_m - t_n) \; > \\
& \quad \sum_{i=1}^{n} h_i \cdot (t_n - t_i + t_m - t_n) \\
\Rightarrow^{(b)} \quad & \sum_{i=1}^{n} h'_i \cdot (t_n - t_i) \; > \; \sum_{i=1}^{n} h_i \cdot (t_n - t_i) \\
& \quad + \; (\sum_{i=1}^{n} h_i - \sum_{i=1}^{n} h'_i) \cdot (t_m - t_n) \\
\Rightarrow^{(c)} \quad & \sum_{i=1}^{n} h'_i \cdot (t_n - t_i) \; > \; \sum_{i=1}^{n} h_i \cdot (t_n - t_i) \\
\Rightarrow^{(d)} \quad & \sum_{i=1}^{n-1} h'_i \cdot (t_n - t_i) \; > \; \sum_{i=1}^{n-1} h_i \cdot (t_n - t_i)
\end{aligned}$$

(a) follows as the sum of all hits and misses is the same for both policies ($|\sigma_1|$) and the second term on both sides of the inequality can be removed. The second term on the right hand side of (b) is non-negative because $t_m > t_n$ and by Lemma II.1, no policy can have more aggregate hits (up to any cache level) than OPT-UB. (c) follows by removing the non-negative second term in inequality (b). (d) follows as the n^{th} term in the summation is zero. Note that between step (a) and step (d), the superscript of the summation has dropped from n to $n-1$. Steps (a) through (d) can be repeated until $n = 2$ (as, for all i, $t_i > t_{(i-1)}$). We will then arrive at $h'_1 \cdot (t_2 - t_1) > h_1 \cdot (t_2 - t_1)$. As $t_2 > t_1$, it implies that $h'_1 > h_1$, which contradicts Lemma II.1, which states that OPT-UB maximizes $\sum_{i=1}^{k} h_k, \forall k \leq n$ (including $k = 1$).

■

C. Optimal Lower Bound: OPT-LB

We now introduce a very simple and practical offline algorithm called OPT-LB, which provides a very close lower bound on optimal multi-level caching performance. A better performing policy will have to demonstrate either lower average response time or lower inter-cache bandwidth usage. OPT-LB is the best known offline multi-level caching policy that we are aware of.

The basic idea is to apply Belady's OPT algorithm in a cascaded fashion. We start with the highest cache level, C_1, and apply OPT to the request sequence σ_1, assuming a single cache scenario. We note the misses from C_1 with their timestamps and prepare another request sequence, σ_2, for the lower cache C_2. We repeat the same process at C_2, in turn generating σ_3. This is performed for each level and we keep a count of hits obtained at every level.

In other words, $h_i = hitOPT(\sigma_i, S_i)$ and $\sigma_{i+1} = traceOPT(\sigma_i, S_i)$, where $traceOPT$ is a trace of the misses when OPT is applied to the given request stream and cache size.

Once each level has learned its σ_i, all cache levels can operate together replicating the result in real-time. Since this can be done practically, this policy by definition serves as the lower bound for the offline optimal along any performance metric. Note that OPT-LB does not require any demotions or reloads from disks. Even though OPT-LB does not guarantee exclusivity of caches, we experimentally confirm that OPT-LB is indeed a very close lower bound for offline optimals for both average response time and inter-cache bandwidth usage in multi-level caches.

D. Bounds for Multi-path Hierarchies

It is simple to extend OPT-UB and OPT-LB for any complex cache hierarchy. While it is fairly common to accept the use of traces for multi-client caching experiments ([33], [19]), the results are accurate only in cases where the relative order of requests from various clients can be assumed fixed. The same holds true for OPT-UB and OPT-LB, which are valid bounds for multi-path hierarchies if we can assume that the relative order of requests from various clients is fixed. Note that there is no such caveat in single client scenarios, where trace-based analysis is accurate.

We extend OPT-UB as follows: we start with determining the maximum hit ratio obtainable at each cache at the highest level by applying Belady's OPT. Similarly, we determine the maximum aggregate hit ratio obtainable in each two-level subtree starting at the highest levels. We subtract the hit ratio of the highest level caches to obtain the hit ratio for the second-level caches. We do this until we have hit ratio values for all cache levels, using which, we arrive at the OPT-UB average response time value. This is a simple generalization of the single-path approach.

```
ADAPTING PROBPROMOTE:

float $hintFreq = 0.05$;
float $sizeRatio = \sum_{i=1}^{k-1} S_i / \sum_{i=1}^{k} S_i$; (at level $k$)
float $probPromote = sizeRatio$;
struct adaptHint {
    time_t $life$;        // life of the cache
    float $occupancy$;// fraction of cache occupied by $T_2$
                          // -only required by PROMOTE-ARC
} $higherHint$;           // hint from higher cache

Every $cache.life * hintFreq$
1:  Prepare and send adaptHint to lower cache

On receiving adaptHint $h$
2:  $higherHint = h$;
3:  Every alternate time (to observe the impact
    of the previous adaptation): adjustIfNeeded();

adjustIfNeeded()
4:  static float $prev = 0$;
5:  float $curr$ = adaptRatio(); /* algo-specific */
6:  float $f$ = $(2 * curr - 1)$; /* $f = 0$ is the target */
7:  if $((f > 0$ &&
8:       $prev - curr < hintFreq * (prev - 0.5))$ ||
9:      $(f < 0$ &&
10:      $curr - prev < hintFreq * (0.5 - prev)))$
11:     $probPromote +=$
12:             $(1 - probPromote) * probPromote * f$;
13:     if $(probPromote > sizeRatio)$
14:             $probPromote = sizeRatio$;
15:     endif
16: endif
17: $prev = curr$;

shouldPromoteUpwards()
18: if $(HighestCache$ ||
19:     $probPromote <$ randomBetween0and1())
20:     return $false$;
21: endif
22: return $true$;
```

Fig. 4. The common methods used by PROMOTE to adapt *probPromote* within every cache by leveraging the hint protocol.

We extend OPT-LB for multi-path hierarchies by merging traces (according to timestamps) emerging from higher caches before applying them to the lower cache. We present a couple of illustrative examples towards the end of the paper (Figures 14 and 15).

III. THE PROMOTE TECHNIQUE

The goal of the PROMOTE technique is to provide exclusive caching that performs better than DEMOTE, while at the same time, requires no demotions, reloads from disks, or any computationally intense operation. Each cache independently and probabilistically decides whether to keep the data exclusively as it is passed along to the application. The probability of holding the data or promoting the data upwards is adaptively determined.

```
PROMOTE-LRU:

adaptRatio() /* return a value between 0 and 1 */
23: return $higherHint.life$ /($cache.life$ + $higherHint.life$)

On receiving from higher cache (readReq $addr$)
24: page $p$ = lookupCache($addr$);
25: if ($p$) /* hit */
26:     $promoteHint$ = shouldPromoteUpwards();
27:     if ($promoteHint$)
28:         remove page $p$ from cache
29:     endif
30:     send to higher cache ($p$, $promoteHint$)
31: else /* miss */
32:     send to lower cache ($addr$)
33: endif

On receiving from lower cache (page $p$, bool $promoteHint$)
34: if ($promoteHint$)
35:     $promoteHint$ = shouldPromoteUpwards();
36:     if (!$promoteHint$)
37:         create page $p$ in cache
38:     endif
39: endif
40: send to higher cache ($p$, $promoteHint$)

PROMOTE-ARC:

adaptRatio() /* returns a value between 0 and 1 */
41: float higher = $higherHint.occupancy/higherHint.life$;
42: float self = $T_2.occupancy/T_2.life$;
43: return $self/(self + higher)$;

On receiving from higher cache (readReq $addr$, bool $T_2hint$)
44: page $p$ = lookupCache($addr$);
45: if ($p$ || $addr$ found in history)
46:     $T_2hint = true$;
47:     remove $addr$ from history (if present)
48: endif
49: if ($p$) /* hit */
50:     $promoteHint$ = shouldPromoteUpwards();
51:     if ($promoteHint$)
52:         remove page $p$ from cache
53:     endif
54:     send to higher cache ($p$, $promoteHint$, $T_2hint$)
55: else /* miss */
56:     send to lower cache ($addr$, $T_2hint$)
57: endif

On receiving from lower cache (page $p$,
                            bool $promoteHint$, bool $T_2hint$)
58: if ($promoteHint$)
59:     $promoteHint$ = shouldPromoteUpwards();
60:     if (!$promoteHint$)
61:         if ($T_2hint$)
62:             create page $p$ in $T_2$
63:         else
64:             create page $p$ in $T_1$
65:         endif
66:     endif
67: endif
68: send to higher cache ($p$, $promoteHint$, $T_2hint$)
```

Fig. 5. The PROMOTE augmentations to ARC and LRU algorithms. These enhancements are apart from the regular mechanisms (not shown) inherent in these algorithms.

We present the PROMOTE algorithm in Figures 4 and 5.

A. Achieving Exclusivity

The PROMOTE technique ensures that only one cache claims ownership of a page on its way to the client. On subsequent accesses to the page, the page can either stay in the same cache or be promoted upwards. As in DEMOTE, only when a page is accessed via different paths in a multi-path hierarchy can a page appear in multiple locations (which is better than enforcing absolute exclusivity).

B. Using $promoteHint$ with a READ Reply

While DEMOTE uses an additional bandwidth-intensive operation (DEMOTE: similar to a page WRITE), PROMOTE uses a boolean bit, called $promoteHint$, that is passed along with every READ reply, and helps to decide which cache should own the page. The $promoteHint$ is set to true when a READ reply is first formed (cache hit or disk read) and set to false in the READ reply when some cache decides to keep (own) it. A READ reply with a false $promoteHint$ implies that a lower cache has already decided to own the page and the page should not be cached at the higher levels. When a cache receives a READ reply with a true $promoteHint$, the cache gets to decide whether to keep (own) the page locally or "promote" it upwards. At each cache level PROMOTE maintains a separate $probPromote$ value, which is the probability with which that cache will promote a page upwards. If a cache decides to own the page (Lines 18-22), it changes the $promoteHint$ to false in the reply and maintains a copy of the page locally. In all other cases, a READ reply is simply forwarded to the higher cache with the $promoteHint$ value unchanged, and any local copy of the data is deleted from the cache. The highest cache has $probPromote = 0$, implying that it always owns a page that it receives with a true $promoteHint$. A page received with a false $promoteHint$, implying that the page is already owned by a lower cache, is merely discarded upon returning it to the application.

C. Promoting Hits Upwards

Pages that incur repeated hits in PROMOTE are highly likely to migrate upwards in the hierarchy as they are subjected repeatedly to the $probPromote$ probability of migrating upwards. The more the number of hits incurred by a page the more it can climb in the hierarchy. The most hit pages soon begin to accumulate in the topmost level.

Note that while DEMOTE uses inter-cache bandwidth for pages on their way up towards the application and also their way down, PROMOTE saves bandwidth by moving pages only in one direction.

D. Adapting $probPromote$

Each cache maintains and adapts a separate $probPromote$ value. The reader will appreciate that the lower the $probPromote$ value at a cache, the lesser is the rate at which new pages will enter the caches above it. Thus, by changing the value of $probPromote$, a lower cache can influence the influx rate of new pages at the higher caches. The fundamental idea in the PROMOTE technique is to use this leverage to create a situation where the "usefulness" of pages evicted from the various caches are roughly the same (if possible). This is different from DEMOTE, where pages evicted from the higher cache are more useful (and hence need to be demoted) than the pages evicted from the lower cache.

To facilitate the adaptation, PROMOTE requires a very simple interface to periodically receive adaptation information like the *cache life* (the timestamp difference between the MRU and LRU pages in the cache) of the next higher cache(s). At regular intervals, each cache, other than the lowest, sends the hint to the next lower cache (Lines 1-3), using which, the $adaptRatio$ is computed (Lines 23, 41-43). The goal is to adapt $probPromote$ in such a way that the $adaptRatio$ approaches 0.5 (a value that implies that the usefulness of pages evicted from the higher cache is the same as of those from the lower cache). If the higher cache has a larger life ($adaptRatio > 0.5$), $probPromote$ is increased, else it is decreased. Since there is a lag between the adaptation in $probPromote$ and its impact on the $adaptRatio$, the recomputation of $probPromote$ (Lines 4-17) is done only on alternate times the hint is received (Line 3). Further, if the previous adaptation of $probPromote$ is found to have reduced the separation of $adaptRatio$ and 0.5 by a reasonable fraction (Lines 7-10), then no further adaptation is done. To avoid any runaway adaptation, $probPromote$ needs to be carefully adapted so that the adaptation is proportional to the difference between $adaptRatio$ and 0.5 and also is slower when close to extreme values of 0 and 1 (Lines 11-12). To start off in a fair fashion, $probPromote$ is initialized according to the sizes of the caches. Since the higher caches usually demonstrate higher hit rates than the lower caches, we forbid $probPromote$ to go beyond the ratio thus determined (Lines 13-14).

Let us examine some examples of PROMOTE in existing cache management policies:

1) PROMOTE-LRU *:* As shown in Figure 6, LRU is implemented at each cache level, augmented by the PROMOTE protocol. The dynamic adaptation of $probPromote$ at each level, results in equalizing the cache lives and it can be shown that the cache hierarchy achieves a hit ratio equal to that of a single cache of the aggregate size. The same is true, if instead of the

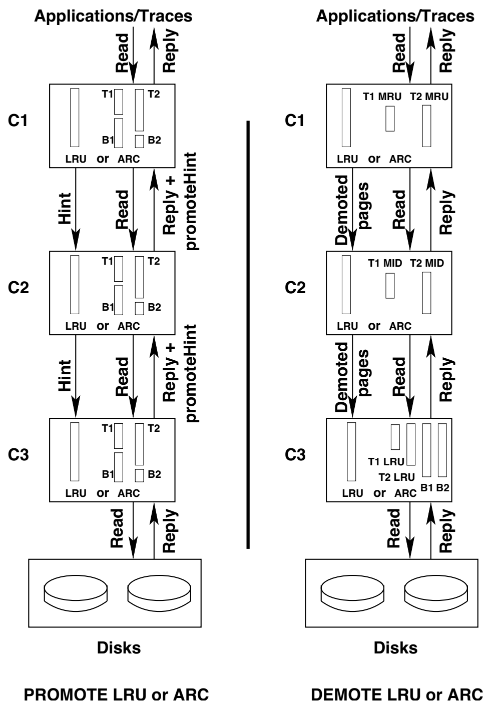


Fig. 6. Communication protocols for the PROMOTE technique (left panel) and the DEMOTE technique (right panel). Although shown in the same diagram, either LRU or ARC data-structures are used consistently across all levels.

cache lives we choose to equalize the marginal utility of the caches. The marginal utility can be computed by measuring the hit rate on a fixed number of pages in the LRU ends of the caches [14]. To avoid the extra complexity, we do not use the marginal utility approach in this paper.

2) PROMOTE-ARC *:* First, we summarize the ARC algorithm [25]: ARC maintains c pages in the cache and c history pages. LRU list T_1 contains pages that have been seen only once recently, and T_2 contains the pages that have been seen at least twice recently. The addresses of the pages recently evicted from T_1 and T_2 are stored in B_1 and B_2, respectively. $|T_1| + |T_2| \leq c$ is enforced, while the relative sizes of the lists are adapted according to the relative rates of hits in the corresponding history lists B_1 and B_2.

For simplicity, consider a single-path hierarchy of ARC caches (Figure 6). Theoretically, the caches could pass the marginal utility of the T_1 and T_2 lists to lower caches which could dynamically adapt $probPromote$ at each cache level to equalize the utility across the hierarchy. However, it can be challenging to adapt $probPromote$ in a stable way for both T_1 and T_2 lists at each level. We found a simple policy that works very well for ARC. For traffic destined for T_1 lists, $probPromote$ at each cache C_k is set to a fixed value $\sum_{i=1}^{k-1} |C_i| / \sum_{i=1}^{k} |C_i|$. Instead of the cache life, the $life/occupancy$ of the T_2 list is passed to lower caches, where $occupancy$ is the fraction of the cache occupied by T_2. Merely using the cache life for T_2 list did not fare well, compared to DEMOTE-ARC, in unlimited bandwidth cases. The $probPromote$ for the T_2 lists is dynamically adapted at each level so as to equalize $life/occupancy$ across the hierarchy (Lines 41-43).

Another hint called the T_2hint is used along with read requests and replies to indicate that the page should be stored in a T_2 list as it has been seen earlier (Line 46). If any cache decides to keep the page (Line 59), it creates the page in T_2 if T_2hint is true; else it creates it in T_1.

E. Handling Multi-path Hierarchies

Since PROMOTE does not significantly alter the local caching policy, extensions to multi-path hierarchies is as simple as requiring that the caches with multiple directly connected higher caches maintain a separate $probPromote$ value corresponding to each such higher cache, and that hints be sent to all directly connected lower caches. It may not always be possible to equalize the cache lives or marginal utilities, however, merely allowing the adaptation to attempt the equalization results in better performance.

On the other hand, it is difficult to conceive a multi-path version of DEMOTE in many cases (e.g. ARC hierarchies in Figures 14 and 15). Hence, PROMOTE is not only easier to apply to multi-level caches than DEMOTE, it is also more broadly applicable.

IV. EXPERIMENTAL SET-UP

A. Traces

We use both synthetic and real-life scenario traces that have been widely used for evaluating caching algorithms.

P1-P14: These traces [25], [17] were collected over several months from workstations running Windows NT by using VTrace [23].

Financial1 and Financial2: These traces [16] were collected by monitoring requests to disks of OLTP applications at two large financial institutions.

SPC1: We use a trace (as seen by a subset of disks) when servicing the SPC-1 benchmark [24]. It combines three workloads that follow purely random, sequential, and hierarchical reuse access models. This synthetic workload has been widely used for evaluating cache algorithms [25], [14], [16], [3].

Zipf Like: We use a synthetic trace that follows a Zipf-like [37] distribution, where the probability of the i^{th} page being referenced is proportional to $1/i^{\alpha}$ ($\alpha = 0.75$, over 400K blocks). This approximates common access patterns, such as in web traffic [9], [5]. Multi-level caching algorithms [33], [12] have employed this trace for evaluations.

Since write cache management policies need to leverage both temporal and spatial locality (see [15]), the write cache is typically managed using a policy distinct from the read cache. Following previous work [33], [12], we focus on the read component of caches and choose to ignore the writes for simplicity. Including the writes would only turn the comparisons more in favor of PROMOTE as they would increase contention for the disk and network resources, a scenario in which PROMOTE easily outshines DEMOTE. Each trace is limited to the first two million reads to shorten the experiment durations.

B. Software Setup

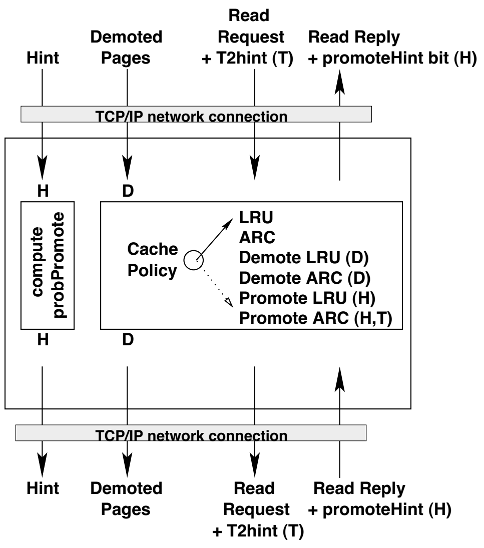

Hint
Demoted Pages
Read Request + T2hint (T)
Read Reply + promoteHint bit (H)
TCP/IP network connection
H
D
compute probPromote
Cache Policy
LRU
ARC
Demote LRU (D)
Demote ARC (D)
Promote LRU (H)
Promote ARC (H,T)
H
D
TCP/IP network connection
Hint
Demoted Pages
Read Request + T2hint (T)
Read Reply + promoteHint (H)

Fig. 7. CacheSim block diagram: Multiple instances form a hierarchy of caches. Algorithm-specific interfaces are marked: D, H, and T (T_2hint).

We implemented *CacheSim* (Figure 7), a framework which allows us to benchmark various algorithms for a wide range of cache sizes, real-life traces, and any single and multi-path hierarchy. *CacheSim* is instantiated with a given cache size and one of the following policies: LRU, ARC, DEMOTE-LRU, DEMOTE-ARC, PROMOTE-LRU, and PROMOTE-ARC. One instance of *CacheSim* simulates one level of cache in a multi-level cache hierarchy, while communicating to the higher and lower caches over a TCP/IP network. The highest level reads requests from a trace file one I/O (size 512 bytes) at a time, with no thinktime, while the lowest level simulates disk responses with a fixed response time.

Apart from the traditional read interfaces, *CacheSim* also implements two special interfaces:

- DEMOTE interface: Used only by the DEMOTE technique to transfer evicted pages to the next lower cache.
- *Hint* interface: Used by the PROMOTE technique to periodically transfer life and occupancy information to the next lower cache.

CacheSim simulates the following realistic roundtrip response times for storage system hierarchies (based on our knowledge): For two-level scenarios: t_1 = 0.5 ms, t_2 = 1.0 ms, t_m = 5.0. For three-level scenarios: t_1 = 0.5 ms, t_2 = 1.0 ms, t_3 = 2.0 ms, t_m = 10.0 ms. The results in this paper are applicable for any set of values where $t_i < t_{i+1}$.

C. The Competitors: DEMOTE *vs.* PROMOTE

To compare the two techniques, we apply both the DEMOTE and the PROMOTE techniques to two popular single-level caching policies. While LRU is the most fundamental caching technique (variants of which are widely used [26], [13], [11], [31]), ARC is arguably the most powerful single-level caching algorithm [25]. A multi-level cache performing the same as a single-level ARC cache represents the most powerful application-independent, multi-level caching policy.

DEMOTE-ARC is implemented by maintaining the T_1 and T_2 lists as global lists, divided among all caches in proportion to the size of the caches [12]. The aggregate hit ratio is precisely equal to that obtained by a single cache of the aggregate size implementing ARC. This is the strongest multi-level caching contender we can devise.

DEMOTE-LRU is implemented as suggested in earlier work [33], also depicted earlier in Figure 2. PROMOTE-LRU and PROMOTE-ARC are implemented as explained in Section III.

For completeness, we also compare the performance of LRU, which is defined as the simple Least Recently Used (LRU) policy implemented in each cache within the hierarchy. There are no inclusion or exclusion guarantees since each cache behaving as if it were the only cache.

D. Measuring Success: The Treachery of Hits

Hit ratio has been extensively used in the study of single-level caches, where higher hit ratios generally imply better performance. In multi-level scenarios, however, the benefit of a hit varies significantly depending on the cache level at which it occurs, making hit ratio a misleading performance metric.

In this paper we use the average response time as the key performance metric. In practice, different algorithms result in different amounts of inter-cache traffic, and in limited bandwidth scenarios, the observed average response time depends more on the inter-cache

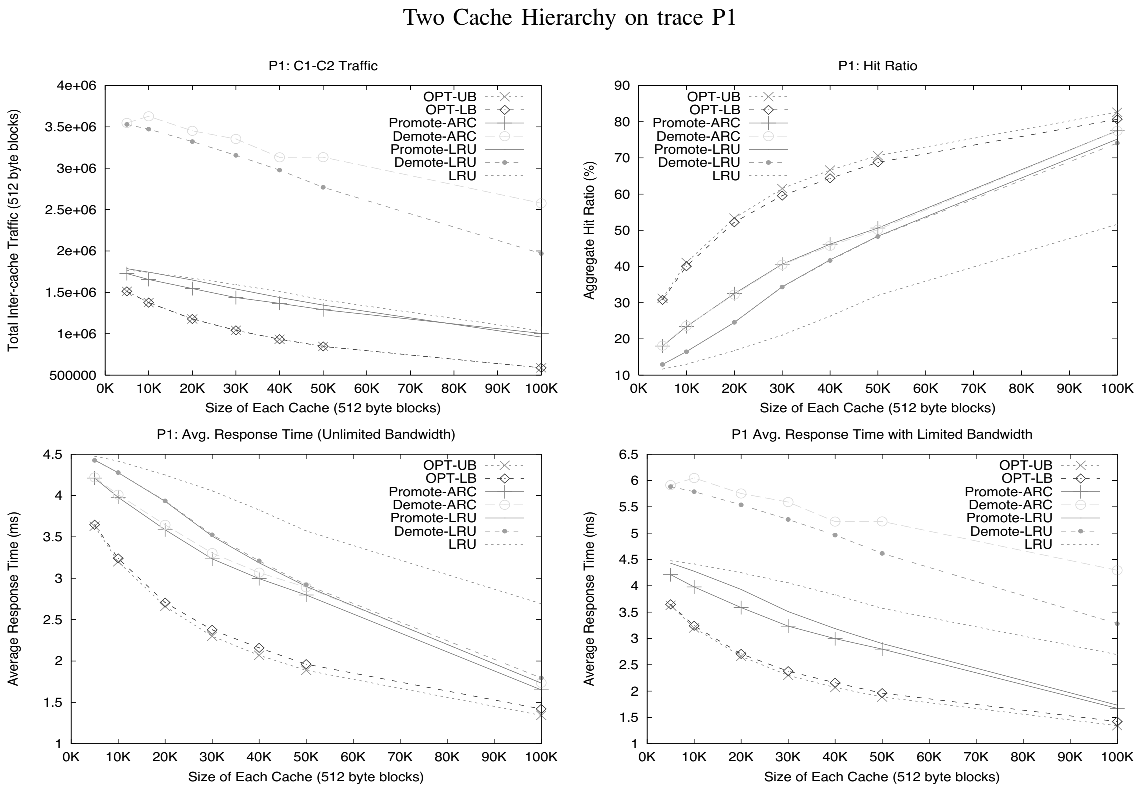


Fig. 8. On the x-axis is the size of the C_1 and C_2 caches. We plot the inter-cache traffic (top-left), the aggregate hit ratio (top-right), and the average response time allowing unlimited bandwidth and free demotions (bottom-left), as a function of the caching algorithm and cache size. In a limited bandwidth scenario (300 blocks per second: 1.5 times that required if there were no hits or demotions), PROMOTE outperforms DEMOTE significantly (bottom-right).

bandwidth usage than on the number or location of hits. Hence, we measure both the inter-cache bandwidth usage and the average response time (assuming unlimited bandwidth). The actual bandwidth limit depends on the hardware and the number and intensity of other applications using the same network fabric. We provide measurements for some limited bandwidth cases as well.

In our experimental results, error bars are not shown since the average response times over separate runs was within 1%, even for the adaptive algorithms.

V. RESULTS

A. *Very Close Bounds on Offline Optimal Performance:* OPT-UB *and* OPT-LB

We computed the average response times for OPT-UB and OPT-LB for a wide range of traces (Section IV-A) and cache sizes (Figures 8 through 13), and found that on an average the bounds ran within 2.18% and 2.83% of each other for two and three level single-path hierarchies, respectively. The maximum separation between bounds for any trace and cache combination was only 8.6% and 10.0% for the two and three level caches, respectively. In terms of inter-cache bandwidth usage, OPT-LB is optimal and coincides with OPT-UB for the C_1-C_2 traffic. This is because OPT-LB does not use any demotions and achieves the maximum possible hits in the C_1 cache (as given by Belady's OPT). For C_2-C_3 traffic, the bounds run, on an average, within 3.4% of each other. OPT-LB is not optimal for the C_2-C_3 traffic because it does not use demotions between the C_1 and C_2 which could have potentially reduced the number of misses flowing out of C_2.

In a more complex multi-path scenario shown in Figure 14 (and Figure 15), the bounds ran about 8.5% (and 10.8%) of each other in terms of the average response time, and coincided in terms of inter-cache traffic.

We believe that the closeness of these bounds in practice and the fact that they are significantly superior to the current state-of-the-art multi-level caching algorithms (Figures 8 through 13) constitute an extremely significant result, and provide an important missing perspective to the field of multi-level caching.

B. *Two Cache Hierarchy*

In Figure 8, we present detailed results for a first trace, P1, in a two cache (same size) hierarchy. We observed similar results with all traces given in Section IV-

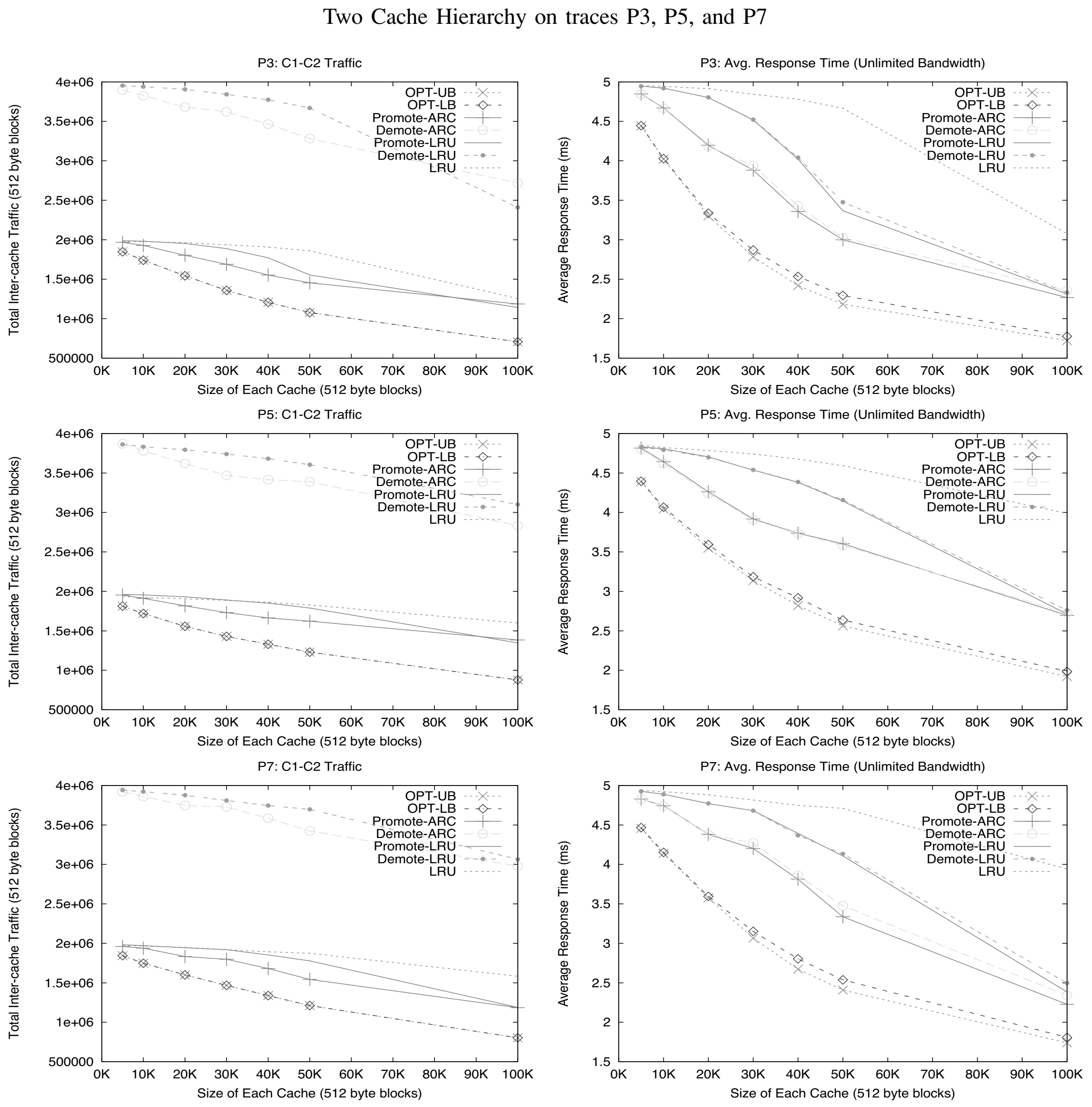


Fig. 9. On the x-axis is the size of the C_1 and C_2 caches. We plot the inter-cache traffic (left), and the average response time allowing unlimited bandwidth and free demotions (right), as a function of the caching algorithm and cache size.

A, the results for some of which are shown in Figures 9 and 10.

Inter-cache Bandwidth Usage: PROMOTE 2x more efficient than DEMOTE. In Figure 8-10, we plot the total traffic between the C_1 and C_2 caches (demotions + C_1 misses). Observe that as the cache sizes grow, the inter-cache traffic decreases as C_1 produces more hits. For both LRU and ARC variants, DEMOTE generates more than double the traffic generated by PROMOTE. This is because DEMOTE causes a demotion for every C_1 miss (after C_1 is full), and also incurs more misses in C_1 than PROMOTE. This is true for all traces and cache sizes, where, on an average, DEMOTE requires 101% more inter-cache bandwidth than PROMOTE for the LRU variant, and about 121% more for the ARC variant.

Aggregate Hit Ratio: PROMOTE same as DEMOTE. In Figure 8, we observe that both PROMOTE-LRU and PROMOTE-ARC achieve almost the same aggregate hit ratio as their DEMOTE counterparts. This was observed for all traces and cache sizes. We also confirm that plain LRU achieves the lowest aggregate hit ratio as the inclusive nature of the lower cache results in very few hits. Please note, however, that the aggregate hit ratio is not a reliable performance metric.

Hits in the Highest Cache: PROMOTE beats DEMOTE.

For the same aggregate hit ratio, a higher number

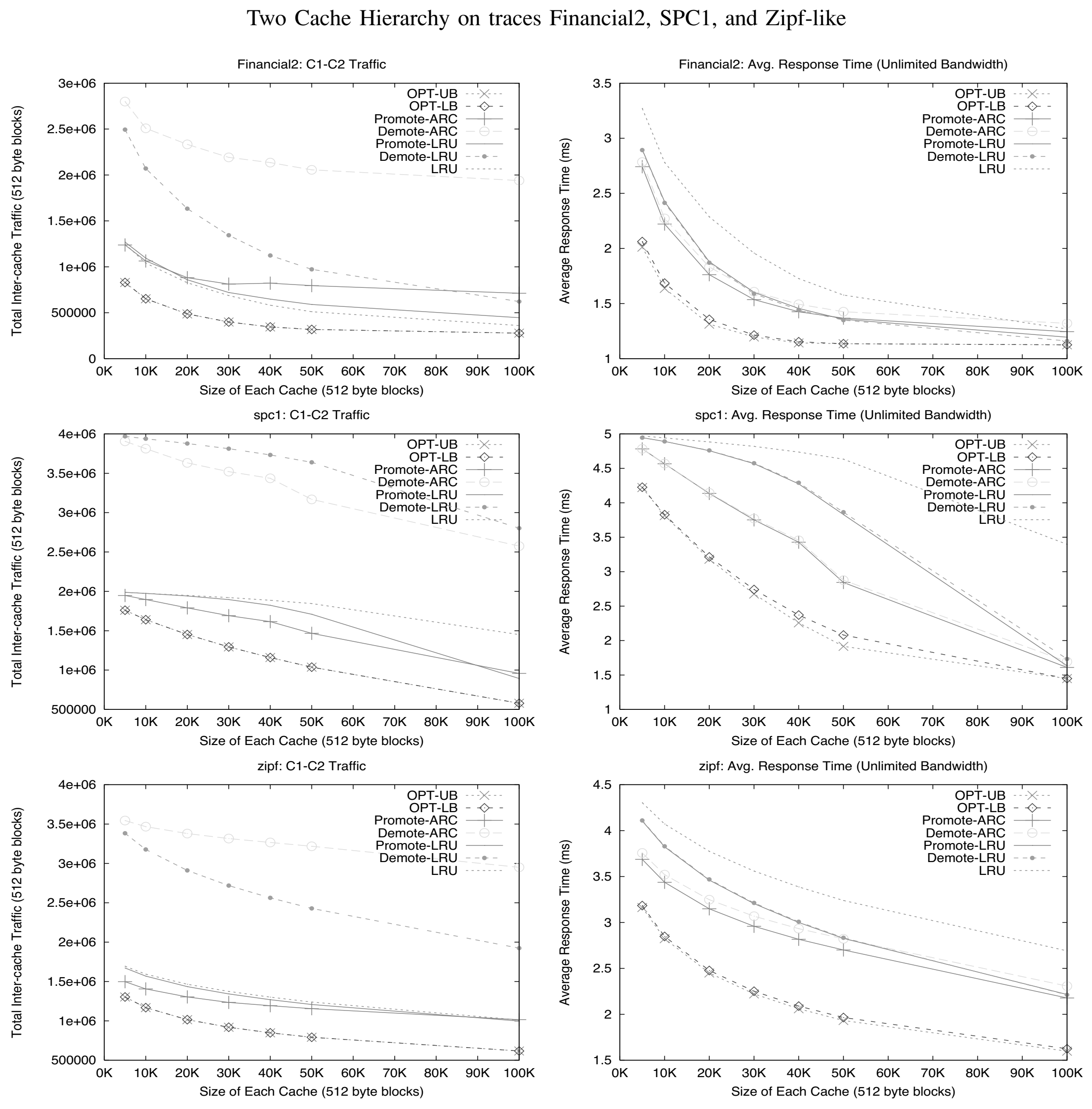


Fig. 10. On the x-axis is the size of the C_1 and C_2 caches. We plot the inter-cache traffic (left), and the average response time allowing unlimited bandwidth and free demotions (right), as a function of the caching algorithm and cache size.

	P1	P3	P5	P7	P9	P11	Financial1	Financial2	spc1	zipf
PROMOTE-ARC	709476	546440	377332	456983	570250	692042	532270	1204243	533414	845104
DEMOTE-ARC	408174	333751	280481	263244	469771	540978	506503	947009	390810	365909
PROMOTE-LRU	653880	446803	210473	222156	511271	639430	644893	1410101	291183	791667
DEMOTE-LRU	590276	140384	172625	125960	396135	667633	688946	1488752	155197	760877

TABLE I. Number of C_1 hits (out of 2000000 requests), at a cache size of $50K$ blocks. While aggregate hits were almost the same for both PROMOTE and DEMOTE, we observed that PROMOTE accumulates more hits in C_1.

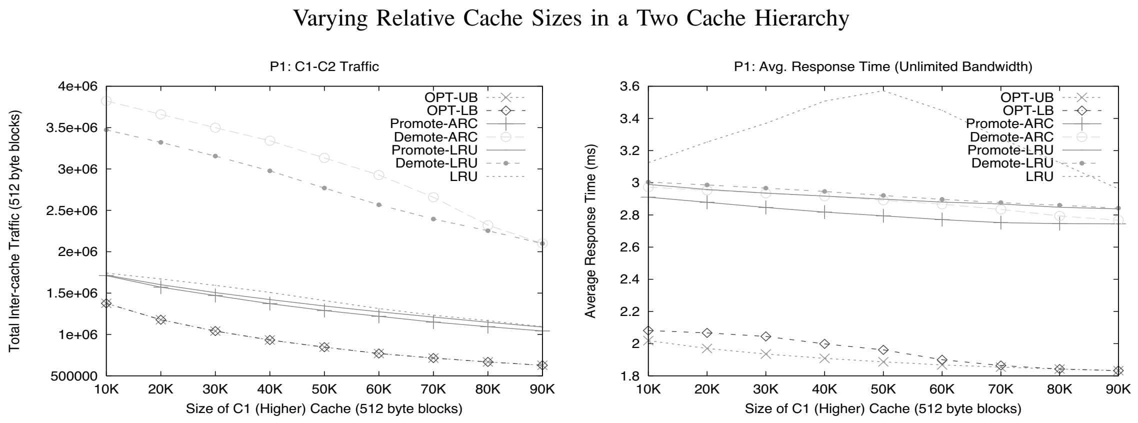


Fig. 11. On the x-axis is the size of the C_1 (higher) cache. $|C_1| + |C_2| = 100K$ blocks.

of hits in the highest cache leads to better average response times. In Table I, we compare the number of hits in C_1 for a wide range of traces with two levels of cache of $50K$ blocks each. While all the traces exhibited similar behavior, we skip some traces in the table to keep it small. We observe that PROMOTE-LRU beats DEMOTE-LRU by 13.0% on an average, and PROMOTE-ARC beats its DEMOTE contender by 37.5%. This is primarily because PROMOTE probabilistically promotes high reuse pages to the higher cache, while DEMOTE forces all pages to flow through the higher cache, pushing out a great number of hits to the lower cache levels.

Average Response Time: PROMOTE beats DEMOTE. In Figure 8 we plot the average response time for the trace P1 at various cache sizes. When we assume an unlimited inter-cache bandwidth and free demotions (although DEMOTE has a rough model to attribute for demotion costs, we create the case most favorable to DEMOTE), PROMOTE-LRU still beats DEMOTE-LRU by up to 4% and PROMOTE-ARC beats DEMOTE-ARC by up to 5% across all cache sizes. For all other traces (some shown in Figures 9 and 10) PROMOTE achieves 0.3%(LRU) and 1.5%(ARC) better response times on an average.

In the lower right panel of Figure 8, we examine a limited bandwidth case, which is more realistic. We allow 300 blocks per second, which is 1.5x that required if there were no hits or demotions ($1/t_m = 200$). When we average the response time across all cache sizes, we observe that PROMOTE substantially outperforms DEMOTE by achieving lower response times, 3.21ms (for ARC) and 3.42ms (for LRU), as compared to DEMOTE with 5.43ms (for ARC) and 5.05ms (for LRU), respectively. In fact, both DEMOTE-LRU and DEMOTE-ARC consistently perform worse than even plain LRU. Surprisingly, for smaller cache sizes, DEMOTE variants perform even worse than no caching at all (i.e. worse than $t_m = 5ms$)! This happens because, when bandwidth is the bottleneck and we use DEMOTE, the bandwidth saved due to one hit in cache C_1 is consumed by one demotion due to a miss. When the number of misses is greater than the number of hits in the cache C_1 ($< 50\%$ hit ratio), a no-caching policy will actually perform better.

Since DEMOTE is clearly worse in the limited bandwidth case, we consistently assume unlimited inter-cache bandwidth and free demotions for the remaining traces shown in Figures 9 and 10.

C. Differing Cache Sizes

In Figure 11, we vary the relative size of the C_1 and C_2 caches from $1 : 9$ to $9 : 1$ while keeping the aggregate cache size equal to $100K$ blocks. We present average response time and inter-cache traffic (assuming unlimited bandwidth and free demotions) for the trace P1 (other traces have similar results). We observe that PROMOTE variants have consistently better response times than the DEMOTE variants across the entire spectrum of relative sizes. The average response time for plain LRU peaks (implying that the hit ratio is the lowest) when the two caches are of the same size. This confirms that the most duplication of data happens when the caches are of comparable sizes. For all the other algorithms, the average response time decreases as the size of the C_1 cache increases as more hits occur at the highest cache.

As before, we observe that the DEMOTE variants invariably consume 2x bandwidth when compared to the PROMOTE variants.

D. Varying Inter-cache Bandwidth

We consider a client using 50K blocks each in two levels of cache, and having a network impact of up to 500KBps. In a typical enterprise SAN environment, there are thousands of such concurrent application threads, scaling the need for both cache and

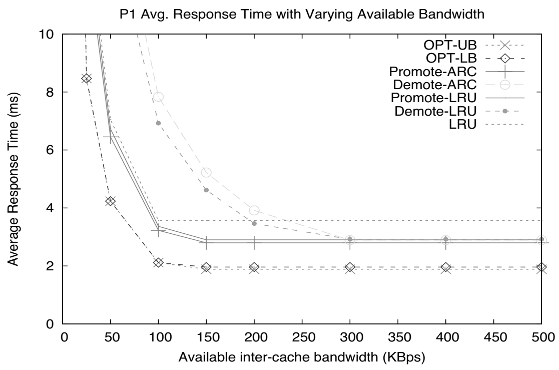


Fig. 12. In a two-cache hierarchy with $50K$ blocks each, we vary the inter-cache bandwidth available to a client on the x-axis. On the y-axis we plot the average response time in milliseconds.

network resources by many orders of magnitude. In fact, even without demotions, many SAN environments are regularly found bandwidth-bound (particularly with sequential reads), implying, as we observe below, that implementing DEMOTE might be detrimental to performance.

In Figure 12, we plot the average response time as a function of the inter-cache bandwidth available to the client. For each algorithm, we notice two regimes, a bandwidth-sensitive regime on the left side where decreasing the bandwidth available increases the average response time, and a bandwidth-insensitive, flat regime, on the right. As expected, OPT-UB, closely followed by OPT-LB, performs the best, with the lowest average response time and the least inter-cache bandwidth requirement (as indicated by the long flat portion on the right). Note that, the DEMOTE variants perform even worse than plain LRU when bandwidth is not abundant. SAN environments which cannot accommodate the 2x network cost of DEMOTE over LRU will see this behavior. This fundamental concern has limited the deployment of DEMOTE algorithms in commercial systems.

The PROMOTE variants are significantly better than the DEMOTE variants when bandwidth is limited, while they outperform LRU handsomely when bandwidth is abundant. As bandwidth is reduced, LRU becomes only marginally worse than PROMOTE because the benefit of more hits in the lower cache, C_2, is no longer felt as the bandwidth between the caches becomes the bottleneck. Overall, PROMOTE performs the best in all bandwidth regimes.

E. Three Cache Hierarchy

Increasing the complexity of the hierarchies we study, we now turn to a three-level (three equal size caches) hierarchy.

As in the two-level case, we present detailed results for the first trace P1 in Figure 13. The other traces had similar results but we do not present plots for lack of space.

We observe that for the wide variety of traces and cache sizes, PROMOTE outperforms DEMOTE in three-level caches as well:

Inter-cache Bandwidth Usage: PROMOTE is 2x more efficient than DEMOTE which uses 105% (111% resp.) more bandwidth between C_1 and C_2 and 98% (113% resp.) more bandwidth between C_2 and C_3, when compared to PROMOTE-LRU (PROMOTE-ARC, respectively).

Aggregate Hit Ratio: PROMOTE same as DEMOTE.

Hits in the Highest Cache: PROMOTE achieves 1.5% and 10% more hits in the top two caches than DEMOTE for the LRU and ARC variants, respectively.

Average Response Time: When bandwidth is not limited and demotions are free, PROMOTE beats DEMOTE by 0.2% and 1.3% on the average response time for LRU and ARC variants, respectively. For a limited bandwidth case, where we allow 200 blocks per second (2x times $1/t_m = 100$), When we average the response time across all cache sizes, we observe that PROMOTE substantially outperforms DEMOTE by achieving lower response times, 5.61ms (for ARC) and 5.93ms (for LRU), as compared to DEMOTE with 8.04ms (for ARC) and 7.57ms (for LRU), respectively.

F. More Complex Cache Hierarchies

PROMOTE can be applied to complex hierarchies. As the possible configurations are endless, we pick two simple and yet interesting configurations for our experiments.

1) Tree-like Hierarchy: We use a hierarchy of three caches, C_{1a} ($40K$ blocks) and C_{1b} ($30K$ blocks) at the first level, and a shared cache C_2 ($50K$ blocks) at the second-level (see Figure 14). While C_{1a} serves one of P2, P3, P4 or P5 traces, C_{1b} serves the P1 trace. We impose no bandwidth restrictions and assume free demotions for the DEMOTE algorithm. We observe that for all four combinations, the PROMOTE-LRU has equal or better average response time while generating only half the inter-cache traffic than DEMOTE-LRU. DEMOTE cannot be applied to tree-like ARC hierarchies, allowing PROMOTE-ARC to win by default. This is because DEMOTE simulates a global ARC algorithm which adapts the ratio of the global ARC lists, T_1 and T_2. In single-path scenarios, the same ratio of the two ARC lists, $|T_1| : |T_2|$, can be enforced at all levels. However, in multi-path scenarios, this is not always possible, as the amount of T_2 pages in a cache depends on the workload it sees, and the ratio determined by ARC may not be enforceable.

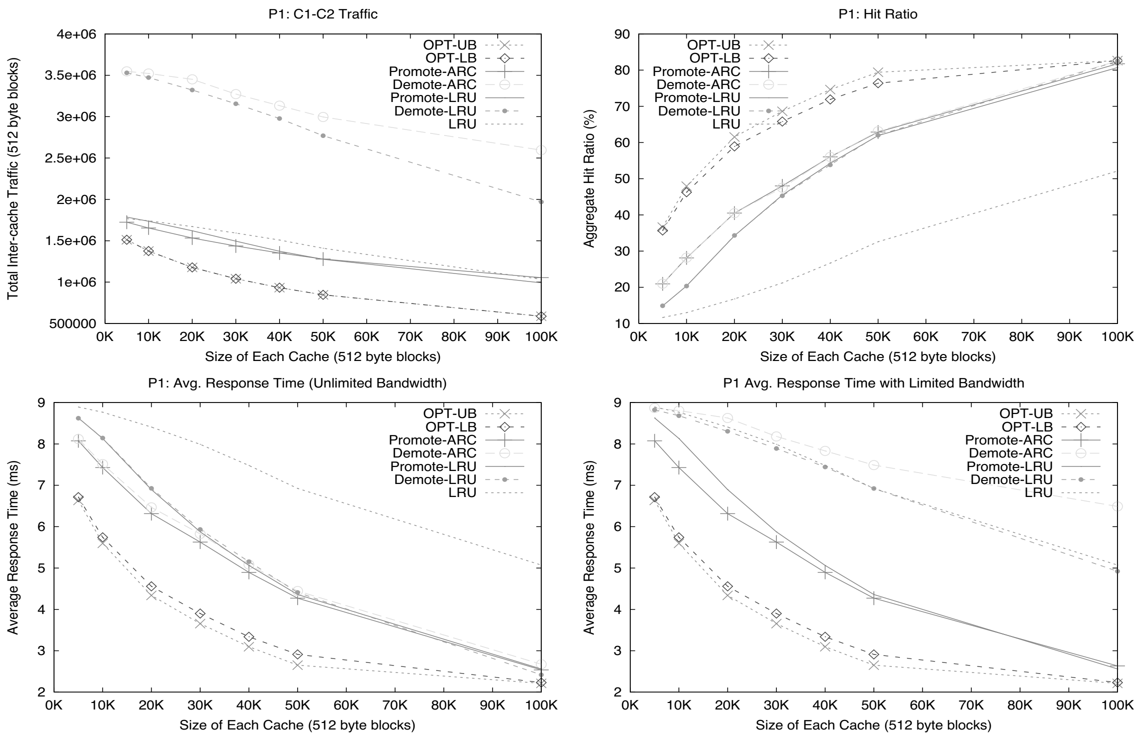
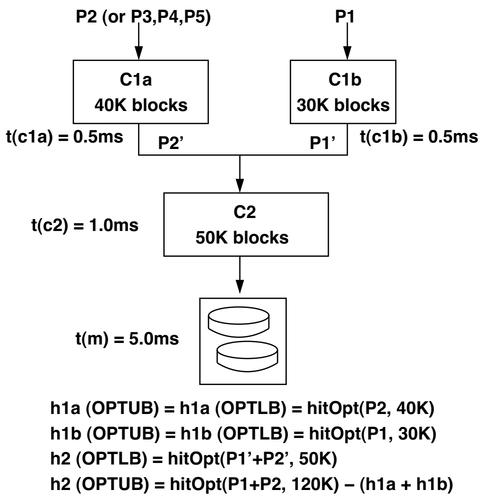
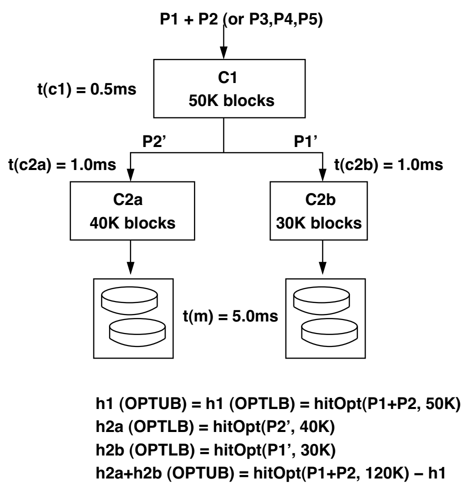


Fig. 13. On the x-axis is the size of each cache: C_1, C_2, and C_3. We present the total traffic between C_1 and C_2 (top-left), the aggregate hit ratio (top-right), and the average response time allowing unlimited bandwidth and free demotions (bottom-left) for a range of per-level cache sizes. In a limited bandwidth scenario (200 blocks per second: 2 times that required if there were no hits or demotions), Promote outperforms Demote significantly (bottom-right).


P2 (or P3,P4,P5)
P1
C1a
40K blocks
C1b
30K blocks
t(c1a) = 0.5ms
P2'
P1'
t(c1b) = 0.5ms
t(c2) = 1.0ms
C2
50K blocks
t(m) = 5.0ms
h1a (OPTUB) = h1a (OPTLB) = hitOpt(P2, 40K)
h1b (OPTUB) = h1b (OPTLB) = hitOpt(P1, 30K)
h2 (OPTLB) = hitOpt(P1'+P2', 50K)
h2 (OPTUB) = hitOpt(P1+P2, 120K) – (h1a + h1b)

		Hit Ratio Contribution = hits / 4000000				Traf. (MBlocks)		Avg.
		C1a	C1b	C2	Aggr	C1a-C2	C1b-C2	Resp. Time
P1+P2	Demote-LRU	0.14	0.10	0.12	0.36	2.83	3.16	3.45
	Promote-LRU	0.14	0.12	0.10	0.36	1.46	1.54	3.45
	Promote-ARC	0.16	0.14	0.13	0.43	1.36	1.43	3.13
	OPT-LB	0.27	0.24	0.10	0.61	0.90	1.04	2.29
	OPT-UB	0.27	0.24	0.12	0.63	0.90	1.04	2.12
P1+P3	Demote-LRU	0.02	0.10	0.10	0.23	3.77	3.16	4.01
	Promote-LRU	0.04	0.11	0.08	0.23	1.83	1.55	4.01
	Promote-ARC	0.10	0.14	0.10	0.34	1.59	1.45	3.51
	OPT-LB	0.20	0.24	0.12	0.56	1.21	1.04	2.53
	OPT-UB	0.20	0.24	0.14	0.59	1.21	1.04	2.29
P1+P4	Demote-LRU	0.02	0.10	0.06	0.18	3.82	3.16	4.23
	Promote-LRU	0.01	0.11	0.06	0.18	1.94	1.55	4.22
	Promote-ARC	0.03	0.14	0.10	0.27	1.87	1.45	3.85
	OPT-LB	0.09	0.24	0.08	0.41	1.66	1.04	3.20
	OPT-UB	0.09	0.24	0.11	0.44	1.66	1.04	2.98
P1+P5	Demote-LRU	0.03	0.10	0.08	0.21	3.68	3.16	4.08
	Promote-LRU	0.04	0.13	0.05	0.22	1.86	1.55	4.06
	Promote-ARC	0.09	0.14	0.09	0.32	1.66	1.43	3.60
	OPT-LB	0.17	0.24	0.10	0.51	1.33	1.04	2.74
	OPT-UB	0.17	0.24	0.12	0.53	1.33	1.04	2.53

Fig. 14. A tree-like multi-path cache hierarchy. Traces do not overlap (2 million reads each). For ease of comparison between caches, the individual hit ratio contributions are normalized based on the total number of reads in the cache hierarchy.



P1 + P2 (or P3,P4,P5)
t(c1) = 0.5ms
C1
50K blocks
P2'
P1'
t(c2a) = 1.0ms
t(c2b) = 1.0ms
C2a
40K blocks
C2b
30K blocks
t(m) = 5.0ms
h1 (OPTUB) = h1 (OPTLB) = hitOpt(P1+P2, 50K)
h2a (OPTLB) = hitOpt(P2', 40K)
h2b (OPTLB) = hitOpt(P1', 30K)
h2a+h2b (OPTUB) = hitOpt(P1+P2, 120K) – h1

		Hit Ratio Contribution = hits / 4000000				Traf. (MBlocks)		Avg.
		C1	C2a	C2b	Aggr	C1-C2a	C1-C2b	Resp. Time
P1+P2	DEMOTE-LRU	0.19	0.10	0.06	0.35	3.17	3.27	3.50
	PROMOTE-LRU	0.18	0.11	0.06	0.35	1.73	1.54	3.49
	PROMOTE-ARC	0.23	0.13	0.07	0.43	1.62	1.46	3.18
	OPT-LB	0.46	0.06	0.08	0.60	1.05	1.09	2.35
	OPT-UB	0.46	0.00	0.00	0.63	1.05	1.09	2.10
P1+P3	DEMOTE-LRU	0.11	0.08	0.07	0.25	3.83	3.25	3.93
	PROMOTE-LRU	0.15	0.04	0.05	0.24	1.87	1.55	3.97
	PROMOTE-ARC	0.20	0.08	0.06	0.35	1.78	1.40	3.52
	OPT-LB	0.40	0.06	0.10	0.56	1.33	1.08	2.57
	OPT-UB	0.40	0.00	0.00	0.59	1.33	1.08	2.27
P1+P4	DEMOTE-LRU	0.13	0.01	0.06	0.20	3.87	3.04	4.15
	PROMOTE-LRU	0.11	0.02	0.06	0.19	1.96	1.58	4.20
	PROMOTE-ARC	0.17	0.03	0.06	0.26	1.96	1.37	3.88
	OPT-LB	0.32	0.05	0.03	0.41	1.75	0.95	3.21
	OPT-UB	0.32	0.00	0.00	0.44	1.75	0.95	2.98
P1+P5	DEMOTE-LRU	0.12	0.04	0.07	0.22	3.75	3.27	4.07
	PROMOTE-LRU	0.13	0.03	0.06	0.22	1.91	1.58	4.07
	PROMOTE-ARC	0.18	0.07	0.07	0.31	1.86	1.44	3.66
	OPT-LB	0.36	0.06	0.09	0.51	1.49	1.06	2.77
	OPT-UB	0.36	0.00	0.00	0.53	1.49	1.06	2.51

Fig. 15. An inverted tree-like single-path cache hierarchy. Two traces are merged before presenting to cache C_1. For ease of comparison between caches, the individual hit ratio contributions are normalized based on the total number of reads in the cache hierarchy.

In Figure 14, we also show the steps used to compute OPT-UB and OPT-LB in accordance to Section II-D. As usual, we observe that OPT-LB and OPT-UB provide close bounds (8.5% apart) on the optimal performance for the given hierarchy.

2) Inverted Tree-like Hierarchy: We invert the cache hierarchy used above as shown in Figure 15. At the first level we have a single cache C_{1a} ($50K$ blocks) and at the second level we have C_{2a} ($40K$ blocks) and C_{2b} ($30K$ blocks). The trace P1 accesses data through the C_1, C_{2a} hierarchy, while the traces P2, P3, P4 or P5 are served through the C_1, C_{2b} hierarchy. We again notice that PROMOTE-LRU performs within 1% of the DEMOTE variant in terms of response time, and is twice as efficient in terms of bandwidth usage. PROMOTE-ARC performs much better than the LRU based algorithms as expected. DEMOTE cannot generalize ARC for this hierarchy and thus is not a contender. Again we observe that OPT-LB and OPT-UB provide close bounds (10.8% apart) on the optimal performance for the given hierarchy.

VI. CONCLUSIONS

As large caches are becoming ubiquitous, multi-level caching is emerging as an important field for innovation. In this paper we have made two major contributions.

We have demonstrated a simple and powerful technique, called PROMOTE, which is significantly superior to the popular DEMOTE technique. For half the bandwidth, PROMOTE provides similar aggregate hit ratios for a variety of workloads, with more hits in the topmost cache. The reduction in bandwidth usage provides huge improvements in average response times when network resources are not abundant. Even in constrained bandwidth cases, unlike DEMOTE, PROMOTE always performs better than a non-exclusive hierarchy of caches. This characteristic is essential for implementation in commercial systems where network usage behavior cannot be predicted. We anticipate the principles in the PROMOTE technique to engender more sophisticated multi-level caching policies in the future.

While improving caching algorithms is important, knowing the theoretical bounds on performance is extremely invaluable. We have provided this much needed knowledge in the form of two very close bounds on the optimal performance. OPT-UB delimits the best possible response time and bandwidth usage for any multi-level caching policy, while, OPT-LB serves as the best known off-line multi-level caching policy that we are aware of. We hope that these new bounds will spur and guide future research in this field.

VII. ACKNOWLEDGMENTS

We are indebted to the anonymous reviewers of the paper for their insightful comments. We are also grateful to Dr. Theodore Wong, our shepherd, for detailed comments and suggestions that greatly improved the readability of the paper.

REFERENCES

[1] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 73–80, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[2] Lakshmi N. Bairavasundaram, Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, Munich, Germany, June 2004.

[3] S. Bansal and D. Modha. Car: Clock with adaptive replacement, 2004.

[4] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Sys. J.*, 5(2):78–101, 1966.

[5] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *INFOCOM (1)*, pages 126–134, 1999.

[6] Z. Chen, Y. Zhou, and K. Li. Eviction-based cache placement for storage caches. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 269–282, 2003.

[7] Zhifeng Chen, Yan Zhang, Yuanyuan Zhou, Heidi Scott, and Berni Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 145–156, New York, NY, USA, 2005. ACM Press.

[8] F. J. Corbató. A paging experiment with the multics system. In *In Honor of P. M. Morse*, pages 217–228. MIT Press, 1969. Also as MIT Project MAC Report MAC-M-384, May 1968.

[9] Mark E. Crovella and Azer Bestavros. Self-similarity in World Wide Web traffic: evidence and possible causes. *IEEE /ACM Transactions on Networking*, 5(6):835–846, 1997.

[10] Peter J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, 1968.

[11] EMC. EMC symmetric dmx architecture guide. http://www.emc.com/products/systems/pdf/C1011_emc_symm_dmx_pdg_ldv.pdf, March 2004.

[12] Michael Factor, Assaf Schuster, and Gala Yadgar. Karma: Know-it-all replacement for a multilevel cache. In *Proc. of the USENIX Conference on File and Storage Technologies, 2007*, 2007.

[13] Kevin W. Froese and Richard B. Bunt. The effect of client caching on file server workloads. In *HICSS (1)*, pages 150–159, 1996.

[14] Binny S. Gill and Dharmendra S. Modha. SARC: Sequential prefetching in adaptive replacement cache. In *Proceedings of the USENIX 2005 Annual Technical Conference*, pages 293–308, 2005.

[15] Binny S. Gill and Dharmendra S. Modha. WOW: Wide ordering of writes - combining spatial and temporal locality in non-volatile caches. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, pages 129–142, 2005.

[16] Pawan Goyal, Divyesh Jadav, Dharmendra S. Modha, and Renu Tewari. CacheCOW: providing QoS for storage system caches. In *SIGMETRICS*, pages 306–307, 2003.

[17] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. The automatic improvement of locality in storage systems. *ACM Trans. Comput. Syst.*, 23(4):424–473, 2005.

[18] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proc. ACM SIGMETRICS Conf.*, 2002.

[19] Song Jiang and Xiaodong Zhang. ULC: A file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. *ICDCS*, 00:168–177, 2004.

[20] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proc. VLDB Conf.*, pages 297–306, 1994.

[21] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Computers*, 50(12):1352–1360, 2001.

[22] Lishing Liu. Issues in multi-level cache designs. In *ICCS '94: Proceedings of the1994 IEEE International Conference on Computer Design: VLSI in Computer & Processors*, pages 46–52, Washington, DC, USA, 1994. IEEE Computer Society.

[23] J. R. Lorch and A. J. Smith. The VTrace tool: Building a system traces for Windows NT and Windows 2000. *MSDN Magazine*, 15(10):86–102, Oct 2000.

[24] Brue McNutt and Steven Johnson. A standard test of I/O cache. In *Proc. Comput. Measurements Group's 2001 Int. Conf.*, 2001.

[25] Nimrod Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST*, pages 115–130, 2003.

[26] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems - or - your cache ain't nuthin' but trash. *Proceedings of the USENIX Winter Conference*, pages 305–313, January 1992.

[27] Li Ou, Xubin He, Martha J. Kosa, and Stephen L. Scott. A unified multiple-level cache for high performance storage systems. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 143–152, Washington, DC, USA, 2005. IEEE Computer Society.

[28] S. Przybylski, M. Horowitz, and J. Hennessy. Characteristics of performance-optimal multi-level cache hierarchies. In *ISCA '89: Proceedings of the 16th annual international symposium on Computer architecture*, pages 114–121, New York, NY, USA, 1989. ACM Press.

[29] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *Proc. ACM SIGMETRICS Conf.*, pages 134–142, 1990.

[30] R. T. Short and H. M. Levy. A simulation study of two-level caches. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 81–88, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[31] J. Z. Teng and R. A. Gumaer. Managing IBM database 2 buffers the effect of client caching on file server workloads. *IBM Systems Journal*, 23(2):211–218, 1984.

[32] Darryl L. Willick, Derek L. Eager, and Richard B. Bunt. Disk cache replacement policies for network fileservers. In *International Conference on Distributed Computing Systems*, pages 2–11, 1993.

[33] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *Proc. of the USENIX Annual Technical Conference*, pages 161–175, 2002.

[34] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. Demotion-based exclusive caching through demote buffering: Design and evaluations over different networks. In *Workshop on Storage Network Architecture and Parallel I/O (SNAPI)*, 2003.

[35] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):505–519, 2004.

[36] Yuanyuan Zhou, James Philbin, and Kai Li. The Multi-Queue replacement algorithm for second level buffer caches. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 91–104, 2001.

[37] G. K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison Wesley, Reading, MA, 1949.