Coordinated Multilevel Buffer Cache Management with Consistent Access Locality Quantification

Song Jiang, Kei Davis, and Xiaodong Zhang, *Senior Member*, *IEEE*

Abstract—This paper proposes a protocol for effective coordinated buffer cache management in a multilevel cache hierarchy typical of a client/server system. Currently, such cache hierarchies are managed suboptimally—decisions about block placement and replacement are made locally at each level of the hierarchy without coordination between levels. Though straightforward, this approach has several weaknesses: 1) Blocks may be redundantly cached, reducing the effective total cache size, 2) weakened locality at lower-level caches makes recency-based replacement algorithms such as LRU less effective, and 3) high-level caches cannot effectively identify blocks with strong locality and may place them in low-level caches. The fundamental reason for these weaknesses is that the locality information embedded in the streams of access requests from clients is not consistently analyzed and exploited, resulting in globally nonsystematic, and therefore suboptimal, placement and replacement of cached blocks across the hierarchy. To address this problem, we propose a coordinated multilevel cache management protocol based on consistent access-locality quantification. In this protocol, locality is dynamically quantified at the client level to direct servers to place or replace blocks appropriately at each level of the cache hierarchy. The result is that the block layout in the entirely hierarchy dynamically matches the locality of block accesses. Our simulation experiments on both synthetic and real-life traces show that the protocol effectively ameliorates these caching problems. As anecdotal evidence, our protocol achieves a reduction of block accesses of 11 percent to 71 percent, with an average of 35 percent, over *uniLRU*, a unified multilevel cache scheme.

Index Terms—Replacement algorithm, locality, multilevel caching, networked file system.

1 Introduction

With the ever-growing performance gap between memory and disk and rapidly improving CPU performance, efficient cache management is becoming an increasingly important consideration in system design. In a client/server system using distributed caches, good buffer cache management is critical to overall system performance. Researchers and practitioners seek to make the best use of the available buffer cache (henceforth just *cache*) along the data paths between disk and processor to satisfy requests before they reach disk surfaces. In addition to caches in clients, blocks may be cached in servers and disk built-in caches. Together these comprise a multilevel cache hierarchy.

1.1 Challenges in Hierarchical Caching

Though hierarchies of cache resources are, in aggregate size, increasingly large, the problem of making them work together to deliver performance commensurate with their aggregate size has not been satisfactorily solved. There are two challenges in achieving this goal.

The first challenge comes from weakened locality in low-level caches.[1] Caching works because of the existence of (temporal) locality, which is an intrinsic property of application workloads. Only a first-level cache is exposed to the original locality and, so, has the highest potential to exploit it. Lower-level caches service misses in higher levels. Thus, a stream of access requests from a client, as seen by a lower-level cache, has been filtered by the higher-level caches. Assuming that caching at higher levels is, to some degree, effective, the result is that the access stream seen by lower-level caches has weaker locality than is seen at higher levels. Williamson provides a more in-depth explication of this filtering effect and some of its implications [30].

The performance of widely used recency-based replacement policies, such as *least recently used* (LRU), can be significantly degraded when they are used below the first level. Muntz and Honeyman [22] and Zhou et al. [34] observed this performance loss. The conclusion is that employing an independent replacement policy below the first level loses opportunities to exploit the original locality information, i.e., the block access history at the client. Thus, the first challenge is making replacement decisions at all levels based on the original access stream which is only available at the first level.

The second challenge is to eliminate redundancy—caching of the same block at more than one level on its retrieval route. Because of redundancy, for some access patterns, the effective aggregate size of a multilevel cache may be no

- S. Jiang is with the Department of Electrical and Computer Engineering, Wayne State University, Detroit, MI 48202. E-mail: sjiang@eng.wayne.edu.
- K. Davis is with the Performance and Architecture Laboratory, Computer and Computational Sciences Division, Los Alamos National Laboratory, Los Alamos, NM 87545. E-mail: kei.davis@lanl.gov.
- X. Zhang is with the Department of Computer Science and Engineering, Ohio State University, Columbus, OH 43210. E-mail: zhang@cse.ohio-state.edu.

Manuscript received 3 Mar. 2005; revised 11 Nov. 2005; accepted 18 July 2006; published online 22 Nov. 2006.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0067-0305.

1. One level of cache is lower than another if it is further from the client. A first-level cache is at the highest level, residing in a client.

0018-9340/07/$20.00 © 2007 IEEE Published by the IEEE Computer Society

larger than that of the single level of cache with largest size. For a multilevel cache hierarchy, we propose using a unified replacement protocol that can determine the appropriate place for each block to be cached (if it should be cached), thus eliminating redundancy. The result is a cache hierarchy with an effective aggregate size equal to the sum of the cache sizes at each level.[2]

There may be multiple clients at the first level in the hierarchical cache system. In such a system, each client, along with its lower levels of caches, exhibits the aforementioned weakened locality and redundant caching problems. The possibility of coordinating the caches among the peer clients is beyond the scope of this paper.

1.2 Possible Solutions: Custom Second-Level Replacement and Unified LRU

Recent work addresses each of the two challenges, but most of that work addresses only one or the other. Multi-Queue [34] and unified LRU [31] are representative.

Multi-Queue (MQ) is a custom second-level replacement algorithm. To overcome LRU's poor performance in the presence of weak locality, MQ uses access frequency as the measure of block locality. It maintains multiple queues corresponding to disjoint ranges of access frequencies and uses the access frequencies of blocks for assignment to queues. Blocks migrate between queues, or are removed, according to their dynamically changing access frequencies. By maintaining a deep access history, MQ can achieve a higher hit ratio than LRU at the second level. Willick et al. also found that frequency-based replacement policies do better than recency-based ones for network file servers [28]. However, frequency-based replacement schemes have two weaknesses in the context of multilevel caching. First, like the prototypical frequency-based algorithm *least frequently used* (LFU), they respond slowly to changes in access patterns and incur a high bookkeeping overhead. Second, because only clients have the original locality information, the effectiveness at lower levels is limited by the attenuation of locality information.

Wong and Wilkes proposed eliminating redundancy by simply applying a unified LRU protocol across a two-level cache [31]. In their scheme, the top of the LRU stack resides in the client and the remainder in the second level (built-in disk-array cache in their case). Though this protocol has a significant advantage over independent replacement decisions at each level by eliminating redundancy, it has two critical weaknesses. First, a unified protocol built on LRU may not work well because LRU only predicts that a block will be reaccessed during its stay in the stack, but does not predict where in the stack the block will be when reaccessed. Thus, blocks that would be better stored in the second level of the cache start their journeys in the first level and move to the second step by step. Because they are not quickly evicted from the first-level cache after use, they use space better allocated to blocks with stronger locality. Second, it may generate undesirable demotions because any access that is not a hit in the first level results in a demotion from the first level, which forces a demotion from the second level, and so on to an eviction from the bottom. It has been shown that the benefits of coordinated cache management can be nullified by the cost of demotions when I/O bandwidth is below a certain threshold [11].

Schemes addressing the concerns of multilevel cache hierarchies must be evaluated in a systematic fashion. Chen et al. provide a comprehensive framework in which a multidimensional space of multilevel schemes may be evaluated [12]. Their framework consists of various combinations of three scheme elements, namely, cache collaboration, local cache replacement algorithms, and local optimization. While their work represents a milestone in the evaluation and understanding of multilevel cache management, we evaluate MQ, unified LRU, and the protocol we propose in the context of a framework described in this paper.

1.3 Our Approach

We address the challenges in two steps. First, we propose a new measure for quantifying locality. We then develop an efficient mechanism for distributing the cached blocks in the cache hierarchy according to this locality measure.

We characterize two attributes of a locality measure, *accuracy* and *stability*. Intuitively, accuracy characterizes how well a locality measure predicts actual locality, while stability characterizes the locality measure's sensitivity to changes in its input. Because the positioning of blocks in the cache hierarchy is based on the locality measure, blocks must be rearranged when their locality values change. Movement between levels of the cache incurs a communication cost, hence the stability of the measure is critical.

After developing a locality measure that demonstrates high accuracy and stability, we design a client-directed file block placement and replacement protocol. The effectiveness of our proposed protocol is evaluated relative to its satisfaction of three criteria: 1) The multilevel cache retains the same hit ratio as that of a single level cache with size equal to the aggregate size of the multi-level cache,[3] 2) localities of blocks are accurately quantified, and 3) communication overheads between caches are kept low.

2 Quantifying Locality for Hierarchical Buffer Caching

While there is a shared intuitive notion of locality, it appears that there is no single quantitative measure that would be ideal in all circumstances. In practice, every replacement algorithm either implicitly defines a locality measure (such as LRU) or seeks to optimize against an explicit measure.

2.1 Locality Measures

A block reference stream is represented as a sequence $\{R_t | t = 0, 1, 2, \ldots\}$. Time t is virtual, defined as the block index in the reference stream. In the stream, the block accessed at time t is R_t. The *distance* between two references R_i and R_j is the number of other *distinct* blocks accessed between time i and time j. If, for $i \leq j$, $R_i = R_j$ and, for all k

2. This is only achievable when there is one client cache at the first level of the hierarchy. In the multiclient case, redundancy is allowed so long as there is no duplicated block on any path between a client and the server.

3. Again, an adjustment must be made in the multiclient case.

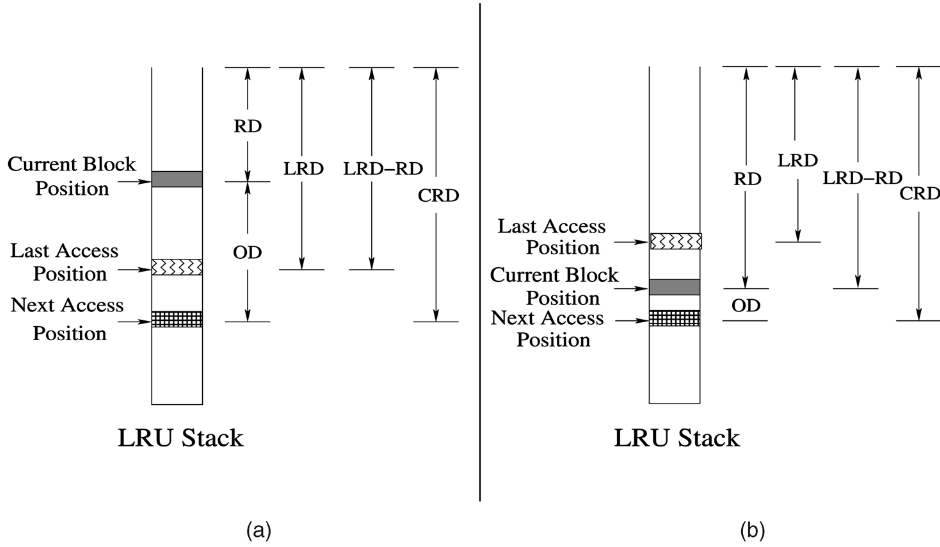

Fig. 1. In the LRU stack, for a given block, the position of the last access to the block corresponds to its LRD, its current position in the stack corresponds to its RD, and the position of its next access corresponds to its CRD. Before its current position exceeds its last position, (a) LRD-RD is LRD; after that, (b) LRD-RD is RD. This allows LRD-RD to accurately predict CRD. The illustration also shows that RD and OD change with every timestep.

such that $i < k < j$, $R_i \neq R_k$, the distance is called the *reuse distance* of R_i. For example, in a stream segment denoted by $\{a, b, c, b, a\}$, the reuse distance of block a is two.

As an offline optimal replacement strategy, OPT uses the distance between the current time and the next reference to a block as the measure of locality of the block [3]. We refer to this as the *OPT distance* (OD). Because OPT replacement maximizes the hit ratio for a given cache by selecting a block with the largest OD for replacement, OD is the most accurate characterization of locality.

The *recency distance* (RD), or just *recency*, of a block is the distance between the current time and the last reference to the block. (Note that the recency of a block *increases* with time as other blocks are accessed.) *Least recently used* (LRU) replacement is based on the assumption that a block accessed recently will be accessed again soon. The LRU stack, originally conceived for the implementation of LRU, stores blocks in the order of their references with the most recently accessed at the top of the stack. Thus, blocks are ordered according to their RD. When a block is accessed, it is placed at the top of the stack; if that block was not in the stack and adding it would exceed the stack size limit, the bottommost block is *demoted*—removed from the stack. Because RD is such a useful measure in the context of replacement algorithms, the LRU stack is widely used to study and describe other replacement algorithms even though they may not actually use RD and their natural implementations may not employ an LRU stack [17], [18], [25]. We note that both OD and RD are measured with respect to the current time, so they may change at every timestep. Because they are dynamic, their stability is of potential concern.

In unified LRU replacement, a block is demoted when its recency exceeds the local LRU stack size [31]. Were it known at the time of access of a block what its recency would be at the time of the next access, it would be cached directly at the level of cache corresponding to that recency (or not cached at all if the recency exceeds the aggregate cache size), thus avoiding demotions. This motivates the use of the distance between the last reference and the next reference to a block, *the current reuse distance* (CRD) to quantify locality. After a block is accessed, its CRD will not change until its next reference, which should give good stability. Like OD, CRD depends on a future access time and, so, is not determinable online.

To predict CRD online we could use the distance between the last reference to a block and the one before the last, the *last reuse distance* (LRD). (The LRD is deemed infinite if there is no penultimate reference.) However, when there is an abrupt change in the access pattern, LRD may not accurately predict CRD. We use RD, which increases with time toward CRD, to predict CRD once RD exceeds LRD. That is, we use the larger of LRD and RD, which we refer to as LRD-RD, to predict CRD. All of these locality measures are illustrated in an LRU stack shown in Fig. 1. We will develop our caching protocol based on a data structure derived from the LRU stack.

2.2 Comparisons of Locality Measures

A replacement algorithm works by ranking blocks according to some locality measure; the block with the lowest rank is the first to be demoted. Each of the four measures, OD, RD, CRD, and LRD-RD, has an associated replacement algorithm. For example, the measure used by the OPT replacement algorithm is OD and the measure used by LRU is RD. How well a replacement algorithm performs as a unified replacement algorithm for a multilevel cache hierarchy is determined by the accuracy and stability of the corresponding locality measure. To evaluate and compare the behaviors of these measures, we use six small-scale workload traces, *cs*, *glimpse*, *zipf*, *random*, *cpp*, and *multi*. We briefly describe these traces and give references to complete descriptions.

1. **cs** is a trace that was collected by running an interactive query answering tool called *cscope* on a set of indexed C source programs [9]. The trace contains 6,781 references to 1,409 distinct blocks.
2. **glimpse** is a text information retrieval utility trace [9]. The search is facilitated by prebuilt indexes on the text files. The trace has 6,015 references to 2,529 distinct blocks.
3. **zipf** is a synthetic trace in which only a few blocks are frequently accessed. Formally, the probability of a reference to the *i*th block is proportional to $1/i$. The trace has 115,467 references to 10,500 distinct blocks.
4. **random** is a synthetic trace with a temporally uniform distribution of references across all the accessed blocks. The trace has 100,000 references to 10,000 distinct blocks.
5. **cpp** is a trace of the GNU C preprocessor *cpp* run against the FreeBSD operating system kernel source of about 11 MB [9]. The trace has 9,047 references to 1,223 distinct blocks.
6. **multi** is obtained by executing the four workloads, *cpp*, *gnuplot*, *glimpse*, and *postgres*, concurrently, which covers a 29 MB data set. *gnuplot* is an interactive plotting program and *postgres* executes join queries on four relations in a postgres relational database. Details of the trace can be found elsewhere under the name of *multi3* [19]. The trace has 30,241 references to 7,454 distinct blocks.

The effectiveness of each of the four measures depends on the access patterns of the workloads. There are three reasons for having chosen these traces. First, each of these traces shows a distinct data access pattern or a combination of distinct patterns. Traces *cs* and *glimpse* have a repeating access pattern, where all blocks are repeatedly accessed in a regular pattern. Trace *cpp* has a so-called LRU-friendly access pattern where blocks accessed more recently are the ones more likely to be accessed again soon. Trace *multi* provides a mix of sequential, repeating, and temporally clustered references. Second, these traces provide representative access patterns. Repeating and sequential access patterns are common in database applications. Random access patterns are observed in the index scan and hash-join database operations. *Zipf*-like access patterns are typical of file access in Web servers. Mixed access patterns are expected in compute servers shared by multiple users. Third, these traces have frequently been used in recent studies of replacement algorithms and their characteristics and their interactions with the replacement algorithms are well understood.

For a given measure, when there is a reference to a block, the locality value of the block, and possibly those of other blocks, changes. For each measure, we maintain a list of blocks ordered by their locality values. The list is updated at each block reference. Each list is divided into 10 segments of equal size, corresponding to a hypothetical level in a cache hierarchy.[4] The number of references to each segment is recorded to determine the accuracy of the measure. Block movements across each segment boundary are counted to determine the stability of the measure. For example, if the given measure is RD, the list is actually an LRU stack of unbounded size. We assume a cold cache (or empty list) in the evaluations, so we take cold access misses into account.

4. The improbably large depth of 10 was chosen to give a fine-grained measure of stability.

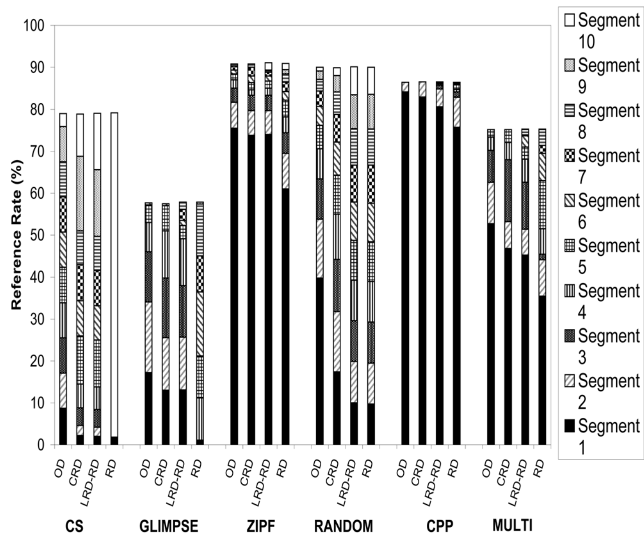

Fig. 2. Reference ratios to each of the segments (the ratios between the number of references to a segment and the number of all references in a workload). It also shows the cumulative reference ratios for the first N segments in each workload, where N is 1 through 10.

We first measure accuracy. An accurate locality measure should generate a distribution of hits that decreases with the distance from the head of the list. This is desirable because access to higher levels is faster. Fig. 2 shows, for each measure and each workload, the relative number of references to each segment of the list. We make the following observations:

1. OD provides the optimal distribution. The higher a segment is (closer to the list head and with a *smaller* segment number in Fig. 2), the higher the reference ratio the segment achieves with OD. In contrast, RD provides the worst distribution, though it attempts to predict OD. This is especially apparent for the workloads *cs* and *glimpse* with repeating access patterns. Most of their references go to the low segments (after segment 9 in *cs* and after segment 3 for *glimpse*). This indicates that unified LRU replacement cannot achieve high hit ratios unless the aggregate cache size can hold all the accessed blocks. RD only performs well on the workloads with an LRU-friendly access pattern, such as *cpp*.
2. CRD performs well for all of the workloads, a foregone outcome. Except for trace *random*, LRD-RD has performance very close to CRD, though it does not depend on future knowledge. Random replacement, wherein blocks are randomly selected for replacement, has a hit ratio proportional to the cache size. All of the online algorithms perform no better than random replacement for trace *random*, also a foregone outcome.
3. LRD-RD exhibits better accuracy than RD for workloads *cs*, *glimpse*, *zipf*, and *multi*. For the LRU-friendly workload *cpp*, both RD and LRD-RD perform very well, with RD performing only slightly better.

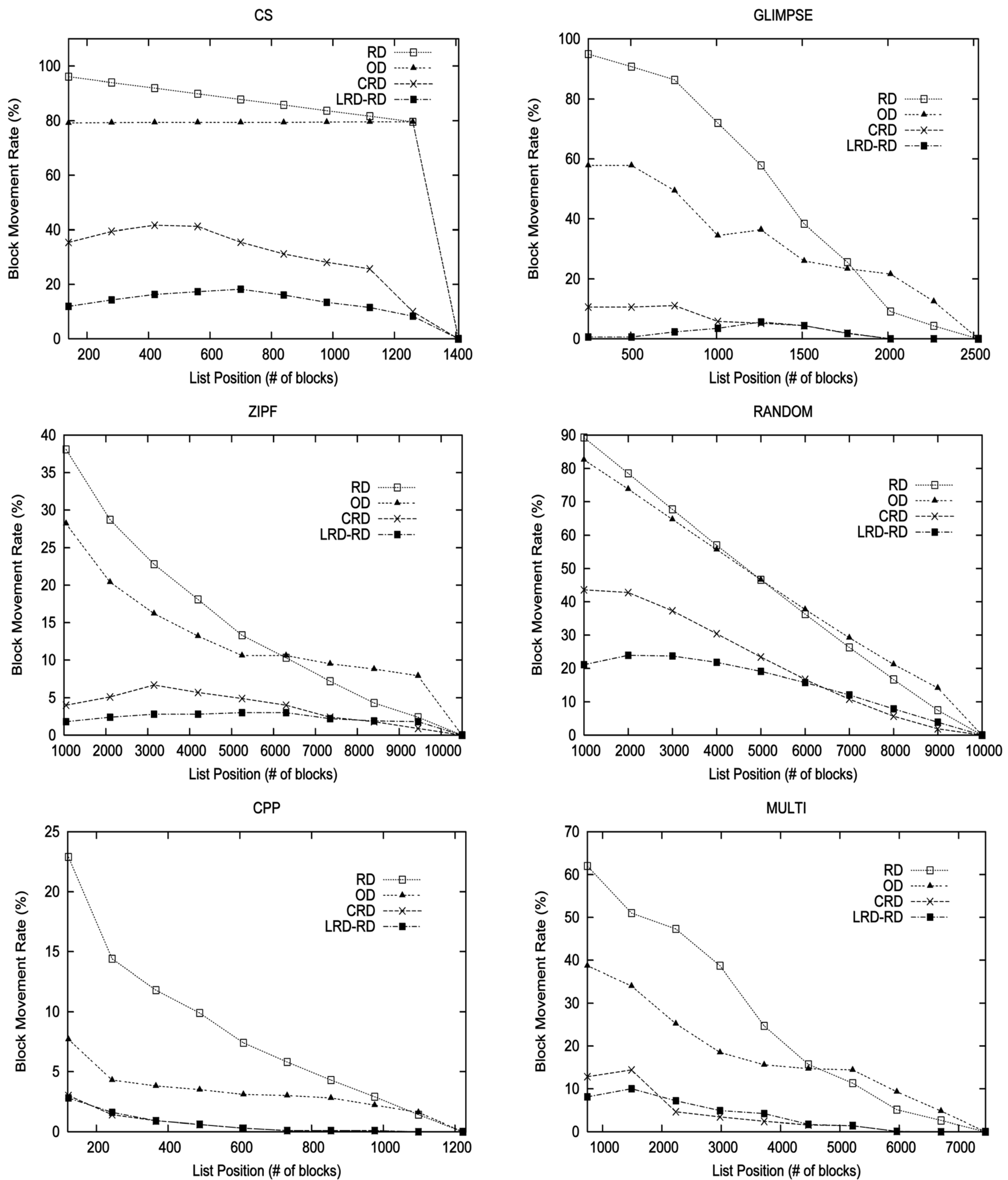

Fig. 3. Comparisons of the ratios of the number of block movements across a segment boundary of the ordered lists and the number of total references for the four measures RD, OD, CRD, and LRD-RD on various workloads. OD and RD consistently exhibit high movement ratios, while CRD and LRD-RD have low movement ratios.

Next we consider stability. Fig. 3 shows the ratios of block movements across each of the segment boundaries and the number of all references for each of the four measures. For example, the first point from the left on a curve shows the fraction of all references that blocks cross the boundary between the first and second segments. A small ratio indicates high stability at that boundary. We make the following observations:

1. As expected because they are dynamic, OD and RD have the highest movement ratios. Comparatively, CRD and LRD-RD have much lower movement ratios.
2. The difference in the ratios for CRD (respectively, LRD-RD) and OD (respectively, RD) are especially pronounced with the repeating pattern workload *glimpse*. However, even for the LRU-friendly workloads like *cpp* and *zipf*, the differences are still large. This demonstrates that an online unified caching protocol based on LRD-RD promises a much smaller additional communication cost than one based on RD.
3. The ratios for LRD-RD are smaller than those for CRD in most cases. Thus, while the accuracy of CRD is optimal, its stability is not.

TABLE 1
Comparisons of the Accuracy of Four Measures of Locality, Their Stability, and Their Implementability

	OD	RD	CRD	LRD-RD
Accuracy	strong	weak	strong	strong
Stability	weak	weak	strong	strong
On-line measure	no	yes	no	yes
Bookkeeping cost	large	small	large	small

The time cost of bookkeeping for each of these measures differs greatly. On average, the costs for OD and CRD are more than a hundred times larger than those of RD and LRD-RD. Table 1 summarizes the relative attributes of the four measures, showing that LRD-RD is a desirable measure for designing a unified caching protocol.

3 The Unified Level-Aware Caching Protocol

We have shown that the ordering defined by LRD-RD provides an accurate indicator for where a block should be cached in a cache hierarchy or that it should not be cached at all.[5] Based on this ordering, we propose a multilevel cache placement and replacement protocol, called *Unified Level-Aware Caching* (ULC), to exploit hierarchical locality.

Based on access patterns and the sizes of the cache at each level, ULC dynamically ranks accessed blocks into levels $L_1, L_2, \dots, L_n$, corresponding to each level of cache, and L_{out} (signifying not to be cached) according to their LRD-RD positions. The size of the first-level cache determines the number of L_1 blocks and similarly for the other levels. To separate the blocks that reside in different levels of cache, we define the block with the least locality at each level to be a *yardstick* block.

ULC runs on the client and lower level caches are not responsible for extracting locality information; they merely respond to directives from the client. Every block request from the client-level cache carries a level tag and, if the attached tag matches a cache's level, it will cache the retrieved block. Otherwise, the block is discarded after the block is sent to the next higher level cache. When the block positions need adjusting, the client sends block demotion instructions down the hierarchy, to which the caches respond appropriately.

Our client-directed protocol attempts to answer the following questions in designing hierarchical caching protocols: 1) How do we effectively and *consistently* exploit locality in the entire cache hierarchy; 2) how do we make the exploited locality usable by all caches in the hierarchy; and 3) how do we minimize the overhead of the protocol.

3.1 A Detailed Description

In Section 2.2, we showed that the LRD-RD measure is a promising basis on which to build a multilevel caching protocol. However, an implementation of an algorithm directly based on LRD-RD ranking will take at least $\Theta(\log n)$ time, where n is the number of distinct accessed blocks, to insert a newly accessed block into the ordered list. To develop an efficient algorithm with $O(1)$ time for block insertion, we transform the process of determining the position of a block in an LRD-RD ordered list into two steps: 1) When a block is accessed, its recency is 0, so its LRD-RD is LRD, which is the recency at which it was just accessed. We use the LRD to determine in which segment the block will be cached at the time of retrieval. 2) Once a block is assigned to a specific segment, we use RD to determine its position in the segment.

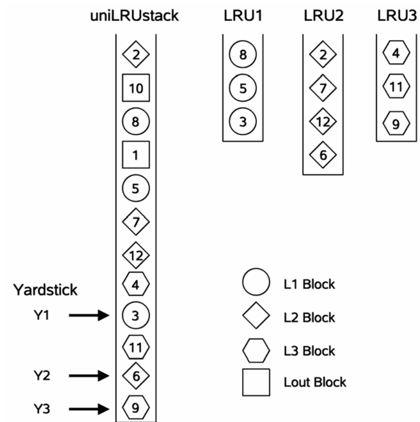

Fig. 4. An example showing the data structure of ULC for a three-level hierarchy. The blocks with their recencies less than that of yardstick Y_3 are kept in *uniLRU-stack*. The level status (L_1, L_2, or L_3) of a block is determined by its position between two yardsticks, where it was accessed last time. Its recency status (R_1, R_2, or R_3) is usually determined by its position between two yardsticks, where it stays currently. To decide which block should be replaced in each level, the blocks at the same level can be viewed as being organized in a separate LRU stack (LRU_1, LRU_2, or LRU_3) and the bottom block is for replacement.

As shown in Fig. 4, recently accessed blocks are maintained in a LRU stack representing the entire cache hierarchy, to which we refer as the *uniLRU-stack*. These blocks could be cached in any level or even not cached.[6] For each level of cache, there is a yardstick block Y_i in *uniLRU-stack*, which is the block in cache level i that has the largest recency value in L_i. The size of *uniLRU-stack* is actually determined by the position of Y_n, the last yardstick, which always sits at the bottom. Any blocks with recencies larger than that of Y_n will be removed from *uniLRU-stack* and become L_{out} blocks, which are not cached at any level. Usually, when a block gets accessed with a recency between the recencies of Y_{i-1} and Y_i, the block becomes an L_i block. All of the blocks cached at the same level can be viewed as a local LRU stack, called LRU_i, where the order of blocks is determined by their recencies in *uniLRU-stack* and its size does not exceed the size of that level of cache. The block to be replaced on level L_i is the bottom block of stack LRU_i. For the requested blocks that are neither cached in LRU_1 nor going to be cached there because their LRDs are larger than the recency of Y_1, we set up a small LRU stack called *tempLRU* to temporarily store these blocks so that they can be quickly discarded from the L_1 cache.

5. Those requested blocks that should not be cached in the first-level cache are still brought into the client for use, but will not be cached there, i.e., these blocks will be quickly replaced from the client after the reference.

6. In an implementation, only some metadata, such as a block identifier and two statuses used in the ULC protocol, are stored in the stack.

There are two possible structures for the cache hierarchy. One is the single-client structure in which there is only one client connected to one server and another is the multiclient structure in which more than one client shares the same server and blocks requested by different clients are shared in the server.[7] There are two additional challenges for the multiclient ULC protocol: 1) how to cache shared blocks in server caches, which could carry different level tags set by different clients, and 2) how to allocate the server cache to different clients.

3.1.1 The Single-Client ULC Protocol

The single-client ULC algorithm runs on the client which holds the first-level cache. It has the knowledge of the size of the cache on each level. For each block in the *uniLRU-stack*, there are two associated statuses: level status and recency status. Level status indicates at which level the block is cached. When a block is accessed, we need to know its recency to determine its level status. The recency is actually its LRD. It takes at least $\Theta(N)$ time to maintain the exact recency information for all blocks, where N is the aggregate size of the caches. In fact, we only need to know the recencies of the two yardsticks the recency lies between. Thus, we maintain a recency status R_i for each block, which usually indicates its recency is between the recencies of yardsticks Y_{i-1} and Y_i (or just less than Y_1 if i is 1). The average cost to maintain recency statuses is $O(1)$, which will be explained.

Initially, if L_i is not full and the levels that are higher are full, any requested L_{out} blocks get level status L_i and reside in level L_i. If all the caches are full, any blocks accessed when they are not in the *uniLRU-stack* are given level status L_{out}. There are two circumstances for a block to be outside of the *uniLRU-stack*. One is that the block is accessed for the first time, another is that block has not been accessed for a long period of time and has left the *uniLRU-stack* from the bottom. For these blocks, their level status is L_{out} and their recency status is R_{out}.

We define an operation for yardsticks in the *uniLRU-stack* called *yardstickadjustment*, which moves a yardstick from the current yardstick block with level status L_i in the direction toward the stack top to the next block with level status L_i. All the blocks it passes, including the current yardstick block, change their recency status from R_i to R_{i+1}. While any newly accessed blocks have recency status R_1, their recency statuses are changed by *yardstickadjustment* after that. We note that the operation can cause *yardstick inversion*, that is, yardstick Y_j can be closer to the stack top than Y_i, where $i < j$. Though yardstick inversion is allowed, it happened only occasionally in our experiments. This is because a reversed yardstick will cause prompt block promotions. These promotions force LRU bottom block demotions, which cause the yardsticks to recover their normal orientation. For the *yardstickadjustment* operation, all yardsticks move in only one direction, toward the stack top. All of the accessed blocks are placed at the stack top. So, on average, a yardstick moves by at most one block with a block access, giving $O(1)$ cost.

When a yardstick block changes its position in the *uniLRU-stack*, we need to perform yardstick adjustment to ensure that the yardstick is on the block with the correct recency status and with the largest recency among the blocks at that level. Demoting a block into a low-level cache is equivalent to moving the bottom block of local stack LRU_i into LRU_{i+1}, which is sorted on their recencies in the *uniLRU-stack*.

There are two types of requests in ULC, both are sent from the client to the lower-level caches to coordinate various levels of caches to work under a unified caching protocol.

1. $retrieve(b, i, j)$, $(i \geq j)$: Retrieve block b from level L_i and cache it on level L_j when it passes level L_j on its way to level L_1.
2. $demote(j, i)$, $(i > j)$: Demote the yardstick for level j to level $j+1$, (i.e., from the bottom of local stack LRU_j to the top of local stack LRU_{j+1}). Repeat the operation at the next cache level until $i = j+1$.

If there is a reference to block b with level status L_i and recency status R_j, there are only two cases to consider: $i = j$ and $i > j$. That $i \geq j$ is invariant: When block b has level status L_i, it stays above yardstick Y_i; any block that stays above yardstick Y_i must have recency status R_j, where $j \leq i$, otherwise yardstick Y_i must have passed the block and, during *yardstickadjustment*, any block passed by Y_i must change its recency status to R_{i+1}. When block b is referenced, it is moved to the top of the *uniLRU-stack* and its recency status becomes R_1. This also makes it stay at the top of stack LRU_i. If $j > 1$, block b goes to stack *tempLRU* in the client and will be replaced soon from the client cache. Then, for each of the two cases, we act as follows: For $i = j$, block b remains in its current level of cache with the same level status ($retrieve(b, i, i)$). For $i > j$, because block b will be moved from level L_i and cached at level L_j ($retrieve(b, i, j)$), a space needs to be freed at level L_j. We demote the yardstick block Y_j to the next lower-level cache, whose yardstick block may have to be demoted in turn if its status level is greater than L_i ($demote(j, i)$). The protocol is described using pseudocode in Fig. 5.

3.1.2 The Multiclient ULC Protocol

When there are multiple clients sharing one server, the caches in the server are no longer solely used by one client. In the single-client ULC protocol, the number of blocks with level status L_i, or the size of stack LRU_i, is determined by the size of the level L_i cache. If the cache at level L_i is shared by multiple clients, an allocation policy is needed on level L_i for good performance of the entire system. To obtain the best performance, it is known that allocation should follow the dynamic partition principle, where each client is allocated a number of cache blocks that varies dynamically in accordance with its working set size. Experience has shown that global LRU performs well by approximating the dynamic partition principle [8]. Thus, we use a global LRU stack called *gLRU* in the server to facilitate the allocation operation. The block order in *gLRU* is determined by the block recencies, which are determined by the timings of requests from clients requiring a block be cached in the server. The bottom block of *gLRU* is the one to be replaced when a free buffer is needed. For each block in *gLRU*, we record its owner—the client most recently requesting the block to be cached in this server. A block is cached at the highest possible level under the instructions from the clients. If there is only one client, the bottom block of *gLRU* is always the yardstick block Y_i in *uniLRU-stack* and is

7. Here we call the high-level cache the client and the low-level cache the server when discussing two adjacent levels.

```
/* Procedure at the first level to be invoked upon a
   reference to block b with Level Li and Recency Rj */

   Send retrieve(b, i, j) to the low level caches;

   if (i > j) {
     demote(j, i);
     /* Yj is the yardstick for level j */
     yardstickadjustment(Yj);
   }

   move b to the top of stack uniLRUstack;
   recency status of b <-- R1;
   level status of b <-- Lj;

   if (j > 1)
     b is placed at the top at stack tempLRU for quick eviction;
```

Fig. 5. The single-client ULC Protocol.

also the bottom block of stack LRU_i in the client. Because the server cache is shared among the clients, the bottom block of LRU_i could have been replaced in the server. If this is the case, it is equivalent to shrinking the cache size of the server dedicated to the client. So, when a block is replaced from *gLRU*, a message is sent to its owner client so that a yardstick adjustment can occur there. Correspondingly, the size of LRU_i is reduced by one. The owner notifications of block replacements can be delayed until the next requested block is sent to its owner client without affecting its correctness. At that time, they are piggybacked on the next retrieved block, thus saving extra messages. Fig. 6 shows an example to illustrate the multiclient case. By dynamically adjusting the yardsticks of affected clients based on the information provided by the allocation policy, we have a ULC protocol running in clients allowing their low level caches to change their sizes dynamically. The changing sizes are the result of the allocation policy with the goal of high performance for the entire system.

4 Performance Evaluation

This section presents our trace-driven simulation results. We compare ULC with two other multiclient caching protocols: independent LRU, or *indLRU*, which is a commonly used protocol, and unified LRU, or *uniLRU*, an LRU-based unified caching protocol [31].

4.1 Performance Metric

We use block hit ratio and average block access time, T_{ave}, to evaluate the performance of various protocols. T_{ave} measures the average time required to access a block perceived by applications. The access time is determined by the hit ratios and miss penalties at different levels of the caching hierarchy, as well as other communication costs. In general, we calculate T_{ave} for an n-level cache hierarchy as follows: $T_{ave} = \sum_{i=1}^{n} h_i T_i + h_{miss} T_m + T_{demotion}$, where h_i is the hit ratio at level L_i cache, T_i is the time it takes to access the cache at level L_i, h_{miss} is the miss ratio for the cache hierarchy (equivalent to $1 - \sum_{i=1}^{n} h_i$), T_m is the cost for a miss, and $T_{demotion}$ is the demotion cost for block placements required by a unified replacement protocol. If we assume the demotion cost for a block from level L_i to L_{i+1} is T_{di} and the demotion rate between level L_i and L_{i+1} is h_{di}, then $T_{demotion} = \sum_{i=1}^{n-1} T_{di} h_{di}$. We do not consider the situation where demotions are delayed so that their costs could be hidden from applications for two reasons. First, demotions are highly likely to occur in a bursty fashion, especially for an LRU-based unified replacement, where up to 100 percent of the references incur demotions. A small number of dedicated buffers makes it difficult to buffer the delayed blocks, thus its performance gain is very limited. Second, reserving a large number of buffers for delayed demotions effectively reduces the cache size and would reduce the hit ratios.

Specifically, for a two-level client-server cache hierarchy, the average access time is as follows:

$$T_{ave} = h_c T_c + h_s T_s + (1 - h_c - h_s) T_m + h_{c-s} T_{c-s},$$

where h_c and h_s are the hit ratios for the client and server, respectively, T_c and T_s are the costs for a hit in the client and server, respectively, and T_m is the cost for a miss in the server. If the disk access time for a block is T_d, T_m can be regarded as $T_s + T_d$, h_{c-s} is the demotion rate between the client and the server. T_{c-s} is the cost for a demotion. We assume $T_c \approx 0$, the demotion cost T_{c-s} is approximated as the server hit time T_s. Then,

$$T_{ave} \approx h_s T_s + (1 - h_c - h_s) T_d + h_{c-s} T_s.$$

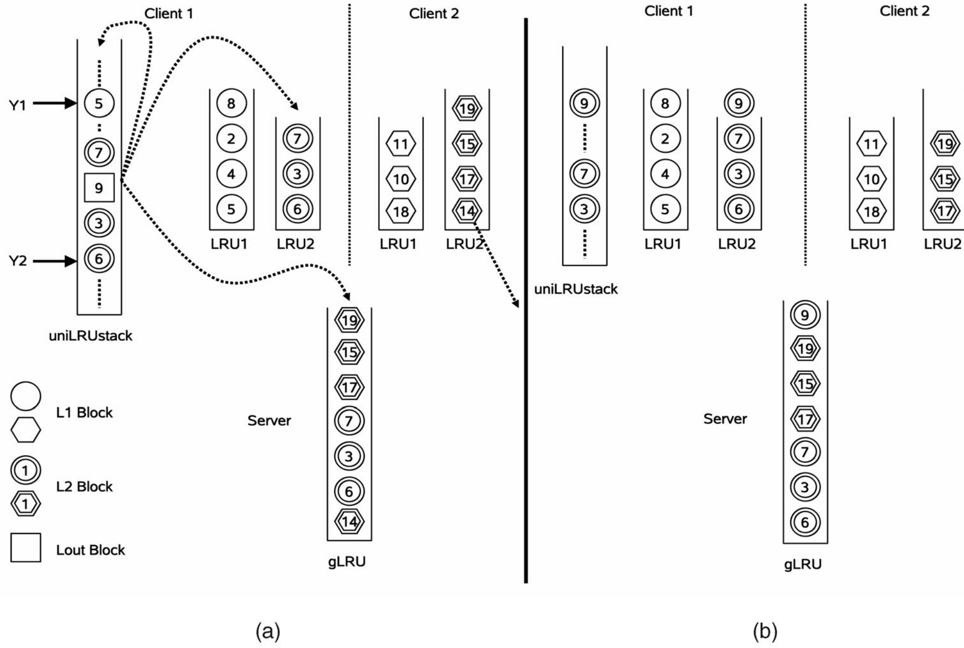

Fig. 6. An example explaining how a requested block is cached in the server cache and how the allocation policy adjusts the size of the server cache used by various clients in a multiclient two-level caching structure. Originally, server stack *gLRU* holds all the L_2 blocks from clients 1 and 2, which are also in their LRU_2 stacks, respectively (see (a)). Then, block 9 is accessed in client 1. Because block 9 is between yardstick Y_1 and Y_2 in its *uniLRU-stack*, it turns into an L_2 block and needs to be cached in the server. Because the server cache is full, the bottom block of *gLRU*, block 14, is replaced, which will be notified to its owner, client 2, through a piggyback on the next retrieved block going to client 2 (delayed notification). After the server cache reallocation, the size of server cache for client 1 is increased by 1 and that for client 2 is reduced by 1 (see (b)).

4.2 Simulation Environment

We use trace-driven simulation for the evaluation. Our simulator tracks the statuses of all accessed blocks and monitors the requests and hits seen at each cache level and the demotions at each level boundary. We assume the size of a cache block is 8 KB. We use seven large-scale traces to drive the simulator: two synthetic traces, *random* and *zipf*, and five real-life workload traces. We have described the two synthetic traces in Section 2. Here, we significantly increase the scale of the two traces: *random* accesses 65,536 unique blocks with a 512 MB data set. It has about 65 M block references. *zipf* accesses 98,304 unique blocks with a 768 MB data set. It has about 98 M block references. The three real-life traces used for the single-client simulation are described as follows:

1. **httpd** was collected on a 7-node parallel Web server for 24 hours [26]. The size of the data set served was 524 MB which is stored in 13,457 files. A total of about 1.5 M HTTP requests are served, delivering over 36 GB of data. We aggregate the seven request streams into a single stream in the order of the request times for the single client structure study.
2. **dev1** is an I/O trace collected over 15 consecutive days on a Red Hat Linux 6.2 desktop [6]. It includes text editor, compiler, IDE, browser, e-mail, and desktop environment usage. It has around 100 K references. The size of the data set is approximately 600 MB.
3. **tpcc1** is also an I/O trace collected while running the TPC-C database benchmark with 20 warehouses on Postgres 7.1.2 with Red Hat Linux 7.1 [6]. It has approximately 3.9 M references. The data set size is approximately 256 MB.

We also select three traces for multiclient simulation. One of them is the original *httpd* trace with seven access streams, each for one client. The other two multiclient traces are as follows:

1. **openmail** was collected on a production e-mail system running the HP OpenMail application for 25,700 users, 9,800 of whom were active during the hour-long trace [31]. The system has six HP 9000 K580 servers running HP-UX 10.20. The size of the data set accessed by all six clients is 18.6 GB.
2. **db2** was collected by an 8-node IBM SP2 system running an IBM DB2 database that performed join, set, and aggregation operations for 7,688 seconds [26]. The total data set size is 5.2GB and it is stored in 831 files.

For all the simulation experiments, we use the first one-tenth of block references in the traces to warm the system before the measurements were collected. The size of the *uniLRUStack* is two times the number of blocks that the aggregate caches can hold.

4.3 Comparisons of Multilevel Protocols in a Three-Level Structure

To demonstrate the ability of multilevel caching protocols (ULC, indLRU, and uniLRU) to accurately quantify locality with stability, we test them in a three-level caching hierarchy for the five single client traces, simulating a scenario where the block retrieval route consists of a disk array containing a large RAM cache, a server, and a client. For a common local network environment, we assume the cost to transfer an 8 KB block between the client and the server through LAN is 0.4ms, the cost between the server and the RAM cache in the disk array through SAN is 0.2ms, and the cost of a block from a disk into its cache is 10ms

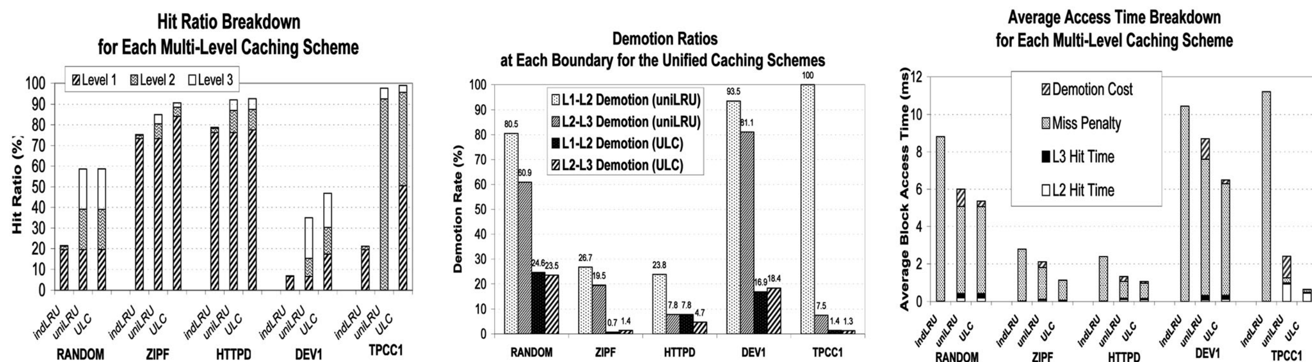

Fig. 7. Hit ratios in each of the three levels, demotion rates at each of two boundaries (between L1 and L2, and between L2 and L3 cache), and average access time for each workload with the multilevel caching protocols indLRU, uniLRU, and ULC.

[31]. We assume the cache sizes of the client, the server, and the disk array are 100 MB each for traces *random*, *zipf*, *httpd*, and *dev1*, and the cache sizes are 50 MB each for trace *tpcc1* due to its comparatively small data set. We report the hit ratios at each of the three levels, demotion rates at each boundary, and average access times for each workload with the three multilevel caching protocols in Fig. 7.

Confirming the experimental results in [31], we observe that there are significant performance improvements of uniLRU over indLRU for all the five traces, from 17 percent to 80 percent reduction on average access time (see the third graph). Actually, these are the results of two combined effects of uniLRU: 1) increasing the cache hit ratios and 2) generating additional demotion cost. UniLRU eliminates the redundancy in the hierarchy, making the low levels of caches contribute to the hit ratio just as if they stayed in the first level. For example, in a random access pattern, the contribution of a cache to the hit ratio should be proportional to its size. However, the second and third levels of caches attain much lower hit ratios (1.7 percent and 0.3 percent, respectively) than that of first-level cache (19.5 percent) for trace *random* in indLRU (first graph). The unified replacement protocol uniLRU makes makes much better use of the lower-level caches. Their hit ratios (19.6 percent and 19.5 percent, respectively) are almost the same as that of first-level cache. However, this improvement comes at a high price: high demotion rates. For example, in trace *random*, uniLRU has a first boundary demotion rate of 80.5 percent, which means 80.5 percent of block references accompany *write-backs* to the server. Furthermore, it has a 60.9 percent demotion rate at the second boundary (second graph). The worst case for the demotion rates of uniLRU is trace *tpcc1*, which has a looping access pattern. Its first boundary demotion rate is 100 percent! This is because uniLRU has little ability to predict the level where an accessed block will be accessed. For a looping access pattern, blocks are accessed at a large recency equal to the loop distance, which implies almost all the blocks of *tpcc1* are accessed after they are demoted into the second level of cache. So, the hit ratio of the second level cache is very high (92.5 percent) and 44.7 percent of the average access time is spent on the demotion. According to the requirement on the ability to accurately quantify locality for a multilevel caching protocol, the distribution of the level L_1 hit ratio (0.03 percent) is much less than the L_2 hit ratio (92.5 percent) under uniLRU shows a bad case.

Compared with uniLRU, the ULC protocol has an access-time-aware hit ratio distribution along the levels of caches: more hits appearing on high levels. For example, the hit ratios of the levels L_1, L_2, and L_3 are 50.3 percent, 45.1 percent, and 3.4 percent, respectively, for trace *tpcc1*. And, such a distribution is achieved without paying the high costs of demotions. For example, the two boundary demotion rates of *tpcc1* are 1.4 percent and 1.3 percent, respectively (second graph). It is also shown that ULC has significant demotion rate reductions over uniLRU for all five traces. This explains why the proportion of demotion cost in the average access time for ULC is much smaller (from 1 percent to 8.3 percent with an average of 4.1 percent) than that for uniLRU (from 12.6 percent to 44.7 percent with an average of 21.5 percent) (third graph).

The access time breakdowns also suggest that ULC will keep performing significantly better than uniLRU except for trace *random*, even if we assume the demotions could be moved off the critical path of response time. In fact, this is an unrealistic assumption: Experiments running a TPC-C benchmark on a client-server system have shown that demotions can significantly delay the network and lower the system throughput [11]. In summary, ULC achieves from 11 percent to 71 percent reduction on average access time with an average of 35 percent over that of uniLRU.

4.4 The Performance Implication of System Parameters

To be widely applicable, a caching protocol should consistently deliver improved performance over existing ones with a large range of system parameters such as cache size and network bandwidth. For the convenience of observing and comparing performance differences of the protocols in this study, we choose a two-level cache hierarchy to present our results. For the two-level hierarchy evaluation, we include Multi-Queue (MQ). In a client-server caching hierarchy, the environment that MQ is designed for, we use MQ in the server and use LRU in the client independently. There is a parameter in the MQ replacement, called *lifeTime*, which determines the speed to decay the frequency of a nonaccessed block. Because this parameter is workload dependent, we run each trace for multiple sample *lifeTime* values in the range suggested in [33] and report the best results of these runs. To see how advanced local replacement algorithms and local optimizations suggested in the multicache evaluation framework [12] could change the relative performance advantages of ULC, we introduce a new protocol, called *indARC*, in which both client and server use the ARC replacement algorithm locally and independently. ARC represents the state-of-art advanced local replacement algorithm [23], which has been

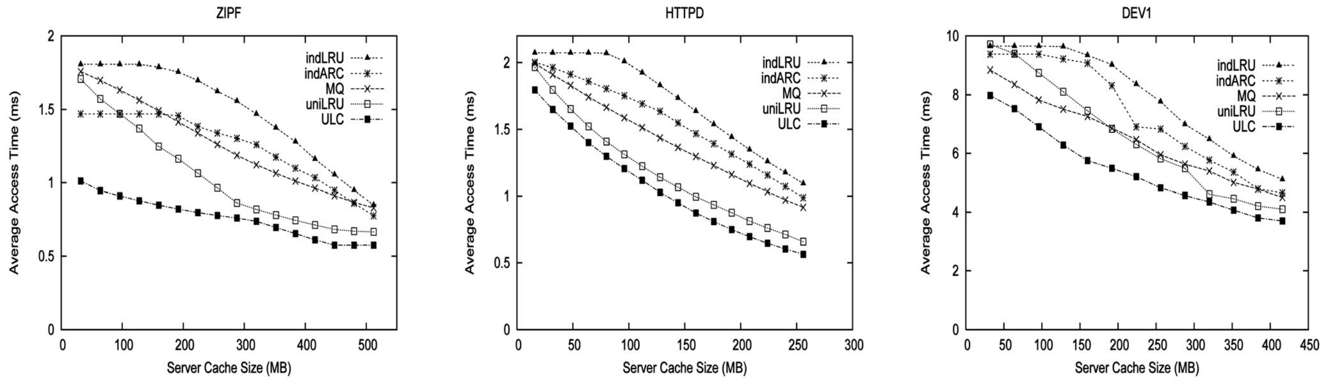

Fig. 8. The average access times for protocols ULC, uniLRU, MQ, and indLRU with various server cache sizes. The client cache size is fixed. It is 256 MB for *zipf*, and 128 MB for *httpd* and *dev1*.

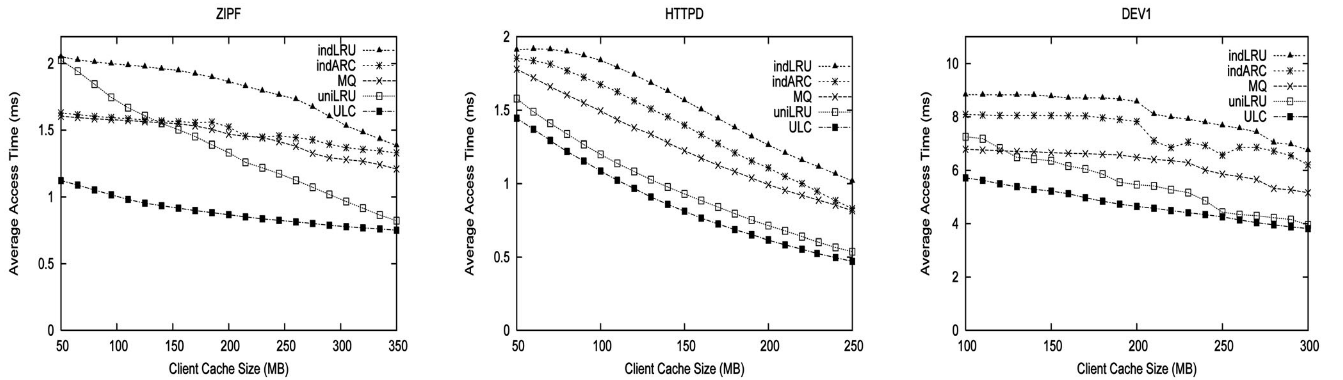

Fig. 9. The average access times for protocols ULC, uniLRU, MQ, and indLRU with various client cache sizes. The server cache size is fixed. It is 200 MB for *zipf* and *dev1* and 150 MB for *httpd*.

deployed in IBM TotalStorage DS8000. Because of space constraints, we report only the results for one synthetic trace, *zipf*, and two real-life traces, *httpd* and *dev1*. The results for other traces are consistent with these.

4.4.1 The Impact of Server Cache Size

Fig. 8 shows the average access times for each workload as the server cache size changes for all five caching protocols: ULC, uniLRU, MQ, indARC, and indLRU. An observation for the indLRU hit ratio curves is that there is a segment of flat curve for each workload with small server cache sizes. These curves start to drop when the server cache sizes approach the client cache size. This demonstrates the serious underutilization of the server cache under indLRU due to the redundancy and locality filtering effect. That is, under indLRU, a relatively small server cache unfortunately has little contribution to the reduction of the average access time in a system with a large client cache size. This is consistent with the results from another study [32], which suggests increasingly large built-in disk caches help little with a comparatively large file system cache under two independent LRU replacements. However, such an observation does not exist for all three of the other protocols, which achieve better performance than indLRU for all the workloads.

The results also show that indARC has a substantially smaller average block access times than indLRU, which conforms to the conclusion in another paper [12] that advanced local replacement algorithms and local optimizations can improve performance without explicit collaboration. However, the performance of indARC still apparently falls behind ULC, an aggressive collaboration scheme, which demonstrates the collaboration scheme holds its advantages over noncollaboration schemes for the tested workloads.

It is shown that, for most of the cases, the performance of uniLRU is better than that of MQ, though MQ does not have demotion costs. This reflects the merit of the unified caching protocol, the elimination of data redundancy. It is also shown that the performance gains of uniLRU over MQ are increased with the increase of server cache size. Our study shows that this is because MQ relies more on the reference frequencies to make a replacement decision when the cache size becomes large. Thus, MQ becomes less responsive to the changing access patterns and less effective than LRU-based protocols with large server cache sizes. For all the traces, ULC achieves the best performance, steadily decreasing the access time with the increase of server cache sizes. Its high hit ratios and low demotion rates are the two major contribution factors.

4.4.2 The Impact of Client Cache Size

Fig. 9 shows the average access times for each workload as the client cache size changes. It is shown that uniLRU benefits much more from the added client cache size than indLRU and MQ. This is because increasing client size has negative effects for indLRU and MQ: more data redundancies in indLRU and weaker locality available for MQ in the server. A unified caching protocol is immune to these effects. However, the performance of uniLRU is worse than that of MQ with small client cache sizes for *zipf* and *dev1*. The explanation is that the smaller the client cache size is, the more requested blocks are retrieved from outside of the client. In uniLRU, every block brought from outside of the

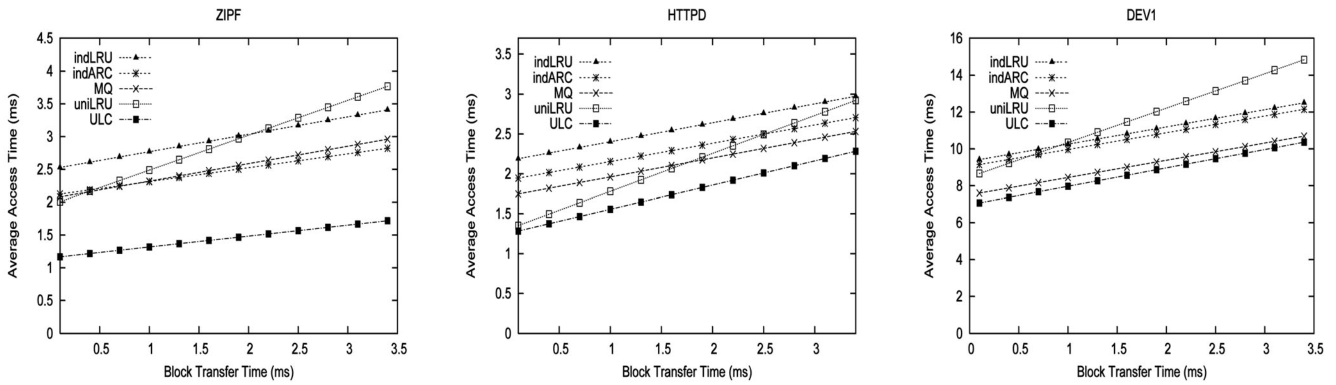

Fig. 10. The average access times for protocols ULC, uniLRU, MQ, and indLRU with various block transfer times. The client and server cache sizes are fixed and are 100 MB each for all the workloads.

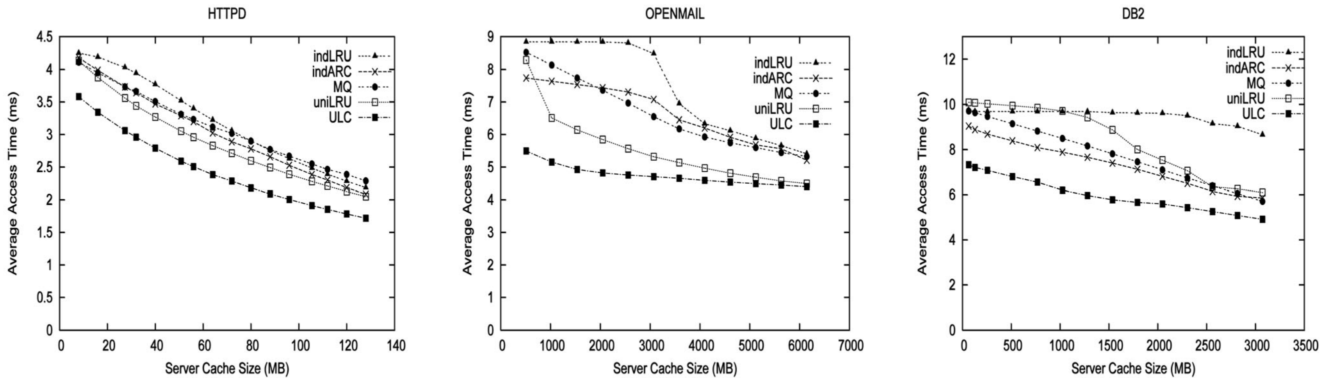

Fig. 11. The average access times of multiclient traces *httpd*, *openmail*, and *db2* with various server cache sizes. Among them *httpd* is with seven clients, *openmail* is with six clients, and *db2* is with eight clients. Each client contains 8 MB, 1 GB, or 256 MB, respectively.

client incurs a demotion. Small client caches cause large demotion costs, which increase the access time in uniLRU. Though ULC is also a unified caching protocol, it maintains its best performance in the whole range of client cache sizes because of its accurate block placement decisions.

4.4.3 The Impact of Network Speed

Fig. 10 shows the average access time for each workload as we change the 8 KB block transfer time. It is expected that the increase of transfer time has a more seriously negative effect for unified protocols than for independent protocols because the former has the additional demotion costs determined by the transfer time. We see that the average access time of uniLRU does increase more quickly than those of indLRU and MQ. However, with low demotion rates, ULC has a similar impact from the increase of transfer time as indLRU and MQ do, even less impact for trace *zipf* because of the contribution of transfer time to the miss penalty and its much reduced miss ratios.

4.5 Comparisons of Caching Protocols for Multiclient Workloads

Because the performance of uniLRU can significantly deteriorate because of cache competition and data sharing among clients for the multiclient structure, Wong and Wilkes also proposed two adaptive cache insertion policies to supplement their primitive protocol [31]. Among their three multiclient traces, *httpd*, *openmail*, and *db2*, *httpd* is the one with data sharing. While they did not state which version of their unified LRU protocols should be used for a specific workload, we ran all the versions and report the best results for comparisons.

Fig. 11 shows that, for all the workloads, ULC achieves the best performance and, in most cases, indLRU has the worst performance. However, there are two cases where indLRU beats uniLRU and MQ. One case is MQ with large server cache sizes for trace *httpd*. When server cache sizes become large enough, LRU's inability of dealing with weak locality becomes less destructive. However, as a frequency-based replacement, MQ's shortcoming of slowness to respond to pattern changes becomes obtrusive. Another case is uniLRU with small cache sizes for trace *db2*. This is because *db2* contains looping access patterns. LRU is not effective on a workload with this pattern until all blocks in the looping scopes can be held in the cache. Carefully examining detailed experiment reports indicates that both indLRU and uniLRU achieve very low hit ratios (6.9 percent and 7.9 percent, respectively), for the two levels of caches, compared with that of MQ (12.3 percent) and that of ULC (35.1 percent). Thus, it is the large demotion cost of uniLRU (with an average demotion rate 88.6 percent for the eight clients, compared with that of ULC (7.2 percent)) that makes the difference. With the increase of the cache size, some looping scopes are covered by the combined two-level caches, but not by a single level, which explains why the performance of uniLRU starts exceeding that of indLRU when the server cache size reaches 1GB. However, the performance of uniLRU is worse than that of MQ because of its looping access pattern. For workloads *httpd* and *openmail*, uniLRU beats MQ by eliminating data redundancy.

5 Other Related Work

In addition to the work described in Section 1, there is some other related work concerned with improving the performance of the client-server hierarchy without changing I/O interfaces or protocols. Chen et al. [11] proposed an eviction-based placement policy that avoids the need to push blocks down the hierarchy by reloading them, from disk, into their correct positions. X-RAY is a mechanism for achieving a high degree of cache exclusivity between client and server [1]. There the server attempts to track the state of the client's LRU stack by recording disk requests and tracking metadata writeback.

Jiang and Zhang propose the LIRS replacement algorithm to address the performance degradation of LRU on workloads with weak locality [18]. In the LIRS stack, blocks with small recencies are the ones that are retained. This single-level cache replacement motivated us to investigate whether last reuse distance (LRD) could be effectively used to predict hierarchical locality such that blocks with different locality values could be placed into correct cache levels. The adaptive caching scheme ACME maintains multiple object orderings simultaneously, each of which corresponds to a replacement policy [2]. There, caching decisions are made according to the current best-performing policy.

Work on cooperative caching [14], [24], [27] seeks to coordinate the caches of multiple clients on a LAN to create a fourth level in the network file system's cache hierarchy. Besides local cache, server cache, and server disk, data may also be cached in another client's cache. Associated issues include availability of idle cache and consistent sharing. While our ULC protocol is intended for the conventional file cache hierarchy, there is evidence that it would also enhance the effectiveness of data placement in cooperative caching [16].

Regarding the cache hierarchy between processor and memory, the interaction of replacements at various levels, inclusivity, and the performance implications are not an issue. This is because, moving down the hierarchy, each cache is typically an order of magnitude larger than its predecessor and these large ratios, together with their greatly differing access latencies, make redundancy inconsequential. In fact, multilevel inclusivity may be accepted as a principle to simplify cache coherence [5]. In contrast, the sizes of caches in the client-server hierarchy do not follow this regularity: A client cache may be larger than the second level cache.

We assume that ULC works in a trusted environment. Though it is a client-directed protocol, ULC does not increase the vulnerability of servers because, even with an independent caching protocol, a client is still able to abuse server caches by sending extra requests to servers to keep its blocks in the server.

Almost all existing file systems use LRU or variants thereof as their underlying replacement algorithms. ULC inherits their central data structure, the LRU stack. The operation cost associated with the stacks is $O(1)$ time with each reference request. Regarding space cost, our techniques needs 17 bytes (8 bytes for the file identifier and block offset, 8 bytes for two pointers in a doubly-linked list, and 1 byte for status) for a block in the client, which represents only 0.2 percent of an 8 Kbyte block. The metadata in the shared server cache needs an additional 1 or 2 bytes for recording the block owner. The stack sizes on other levels except the first one are determined by their cache sizes. Thus, a server with a 1 GB cache only uses 2.2 MB for its metadata. The first level cache has to hold *uniLRUstack*, whose actual size is determined by the working set size of the applications running on the client. If needed to save space, the relatively cold blocks (those with low-level status) could be removed from the stack without compromising the accuracy of ULC. For example, 8.5 MB of metadata in the client is sufficient for a working set of up to 4 GB. This is highly affordable in a system seeking to improve file I/O performance by using large caches.

6 Limitations of the Protocol

While the ULC protocol exhibits impressive performance improvements in both single-client and multiclient scenarios with a diverse collection of workloads, it does have limitations. First, ULC does not consider cache coordination among client caches that are distributed at the same level. As the aggregate client cache size scales with the number of clients, there are ample opportunities to improve system performance by coordinating the client caches. This issue is addressed in another cooperative cache management scheme called *LAC* [16]. Second, we chose LRD-RD as the locality measure in ULC by comparing several measures associated with some popular replacement algorithms. It remains as future work to evaluate other locality measures that are associated with the advanced replacement algorithms that might improve ULC's performance. Third, ULC would require I/O interface protocol enhancements and software changes for deployment. This poses a challenge to its acceptance in industry. As a proof of concept, however, using simulation experiments we have shown that ULC is an attractive candidate for industry adoption.

7 Conclusions

Effective management of a multilevel cache hierarchy is important for good performance of applications for the following reasons: 1) Increasingly more applications rely on the hierarchy for their file accesses, 2) the miss penalties are expensive with the persistent preformance differential between the cache systems and disk storage systems, and 3) a management protocol without efficient coordination can seriously underutilize the cache system and limit the overall system performance. After demonstrating the relatively poor accuracy and stability of extant locality measures with representative file access patterns, we propose the ULC caching protocol. Compared with the commonly used independent LRU protocol and the other recently proposed protocols, the ULC protocol demonstrates distinct performance advantages. Our experimental results show that ULC is able to consistently and significantly reduce average block access times as seen by applications. In addition, we also show that ULC can be implemented efficiently with $O(1)$ time complexity with only a few stack operations associated with each reference.

References

[1] L.N. Bairavasundaram, M. Sivathanu, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, "X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs," *Proc. 31st Ann. Int'l Symp. Computer Architecture,* June 2004.

[2] I. Ari, A. Amer, R. Gramacy, E.L. Miller, S.A. Brandt, and D.D.E. Long, "ACME: Adaptive Caching Using Multiple Experts," *Proc. 2002 Workshop Distributed Data and Structures,* Mar. 2002.

[3] L.A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems J.*, vol. 5, no. 2, pp. 78-101, 1966.

[4] M.G. Baker, J.H. Hartman, M.D. Kupfer, K.W. Shirriff, and J.K. Ousterhout, "Measurements of a Distributed File System," *Proc. 13th ACM Symp. Operating Systems Principles*, pp. 198-212, 1991.

[5] J.-L. Baer and W.-H. Wang, "On the Inclusion Properties for Multi-Level Cache Hierarchies," *Proc. 15th Ann. Int'l Symp. Computer Architecture*, pp. 73-80, 1988.
[6] Trace Distribution Center, Performance Evaluation Laboratory, Brigham Young Univ., http://tds.cs.byu.edu/, 2001.
[7] E.G. Coffman and P.J. Denning, *Operating Systems Theory.* Prentice-Hall, 1973.
[8] P. Cao, E.W. Felten, and K. Li, "Application-Controlled File Caching Policies," *Proc. USENIX Summer 1994 Technical Conf.*, pp. 171-182, 1994.
[9] J. Choi, S. Noh, S. Min, and Y. Cho, "Towards Application/File-Level Characterization of Block References: A Case for Fine-Grained Buffer Management," *Proc. 2000 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* June 2000.
[10] J. Choi, S. Noh, S. Min, and Y. Cho, "An Implementation Study of a Detection-Based Adaptive Block Replacement Scheme," *Proc. 1999 Ann. USENIX Technical Conf.*, pp. 239-252, 1999.
[11] Z. Chen, Y. Zhou, and K. Li, "Eviction-Based Placement for Storage Caches," *Proc. 2003 Ann. USENIX Technical Conf.*, June 2003.
[12] Z. Chen, Y. Zhang, Y. Zhou, H. Scott, and B. Schiefer, "Empirical Evaluation of Multi-Level Buffer Cache Collaboration for Storage System," *Proc. ACM Int'l Conf. Measurement and Modeling of Computing Systems (SIGMETRICS '05)*, June 2005.
[13] P.J. Denning, "The Working Set Model for Program Behavior," *Comm. ACM,* vol. 11, no. 5, May 1968.
[14] M.D. Dahlin, R.Y. Wang, T.E. Anderson, and D.A. Patterson, "Cooperative Caching: Using Remote Client Memory to Improve File System Performance," *Proc. First Symp. Operating Systems Design and Implementation,* pp. 267-280, Nov. 1994.
[15] W. Effelsberg and T. Haerder, "Principles of Database Buffer Management," *ACM Trans. Database Systems,* pp. 560-595, Dec. 1984.
[16] S. Jiang, K. Davis, F. Petrini, X. Ding, and X. Zhang, "A Locality-Aware Cooperative Cache Management Protocol to Improve Network File System Performance," *Proc. 26th Int'l Conf. Distributed Computing Systems,* July 2006.
[17] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," *Proc. 20th Int'l Conf. VLDB,* pp. 439-450, 1994.
[18] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance," *Proc. 2002 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* June 2002.
[19] J. Kim, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim, "A Low-Overhead, High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References," *Proc. Fourth Symp. Operating Systems Design and Implementation,* Oct. 2000.
[20] D. Lee, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim, "On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies," *Proc. 1999 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems*, May 1999.
[21] D. Lee, S. Noh, S. Min, and Y. Cho, "Efficient Caching Algorithms for Two-Level Disk Cache Hierarchies," *Proc. Eighth Ann. Symp. Combinatorial Pattern Matching,* 1997.
[22] D. Muntz and P. Honeyman, "Multi-Level Caching in Distributed File System—or—Your Caching Ain't Nuthin' but Trash," *Proc. USENIX Winter Technical Conf.*, 1992.
[23] N. Megiddo and D. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," *Proc. Second USENIX Symp. File and Storage Technologies (FAST '03)*, Mar. 2003.
[24] P. Sarkar and J. Hartman, "Efficient Cooperative Caching Using Hints," *Proc. Second Symp. Operating Systems Design and Implementation,* 1996.
[25] Y. Smaragdakis, S. Kaplan, and P. Wilson, "EELRU: Simple and Effective Adaptive Page Replacement," *Proc. 1999 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* pp. 122-133, May 1999.
[26] M. Uysal, A. Acharya, and J. Salts, "Requirements of I/O Systems for Parallel Machines: An Application-Driven Study," Technical Report CS-TR-3802, Dept. of Computer Science, Univ. of Maryland, College Park, May 1997.
[27] G. Voelker, E. Anderson, T. Kimbrel, M. Feeley, J. Chase, A. Karlin, and H. Levy, "Implementing Cooperative Prefetching and Caching in a Globally Managed Memory System," *Proc. 1998 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* June 1998.
[28] D.L. Willick, D.L. Eager, and R.B. Bunt, "Disk Cache Replacement Policies for Network Fileservers," *Proc. 13th Int'l Conf. Distributed Computing Systems,* pp. 2-11, 1993.
[29] J. Wilkes, "The Pantheon Storage-System Simulator," Technical Report, HPL-SSP-95-14 rev. 1, HP Lab, May 1996.
[30] C. Williamson, "On Filter Effects in Web Caching Hierarchies," *ACM Trans. Internet Technology,* vol. 2, no. 1, pp. 47-77, Feb. 2002.
[31] T.M. Wong and J. Wilkes, "My Cache or Yours? Making Storage More Exclusive," *Proc. 2002 Ann. USENIX Technical Conf.*, June 2002.
[32] Y. Zhu and Y. Hu, "Can Large Disk Built-In Caches Really Improve System Performance," *Proc. 2002 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* June 2002.
[33] Y. Zhou, "Memory Management for Networked Servers," PhD dissertation, Computer Science Dept., Princeton Univ., Nov. 2000.
[34] Y. Zhou, J.F. Philbin, and K. Li, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches," *Proc. 2001 Ann. USENIX Technical Conf.*, pp. 91-104, June 2001.

Song Jiang received the BS and MS degrees in computer science from the University of Science and Technology of China in 1993 and 1996, respectively, and received the PhD degree in computer science from the College of William and Mary in 2004. He is currently an assistant professor in the Department of Electrical and Computer Engineering at Wayne State University. He was a postdoctoral research associate at Los Alamos National Laboratory from 2004 to 2006, where he developed next-generation operating system capabilities for large-scale parallel computing systems. He received the S. Park Graduate Research Award at the College of William and Mary in 2003. His research interests are in the areas of operating systems, computer architecture, and distributed systems.

Kei Davis received the MS degree in computation from the University of Oxford in 1988 and the PhD degree in computing science from the University of Glasgow in 1994. He is currently a technical staff member and team leader of quantum and classical information sciences at the Los Alamos National Laboratory. His research interests are in the areas of programming languages, operating systems, and parallel computing.

Xiaodong Zhang received the BS degree in electrical engineering from Beijing Polytechnic University in 1982 and the PhD degree in computer science from the University of Colorado at Boulder in 1989. He is the Robert M. Critchfield Professor of Engineering, and chair of the Department of Computer Science and Engineering, at the Ohio State University. He served as the program director of advanced computational research at the US National Science Foundation from 2001 to 2004. He is the associate editor-in-chief of the *IEEE Transactions on Parallel and Distributed Systems* and currently serves on the editorial boards of the *IEEE Transactions on Computers* and *IEEE Micro*. His research interests are in the areas of parallel and distributed computing and systems, computer architecture, and Internet computing. He is a senior member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.