ULC: A File Block Placement and Replacement Protocol to Effectively Exploit Hierarchical Locality in Multi-level Buffer Caches [*]

Song Jiang and Xiaodong Zhang
Dept. of Computer Science, College of William and Mary, Williamsburg, VA 23187, USA

Abstract

In a large client/server cluster system, file blocks are cached in a multi-level storage hierarchy. Existing file block placement and replacement are either conducted on each level of the hierarchy independently, or by applying an LRU policy on more than one levels. One major limitation of these schemes is that hierarchical locality of file blocks with non-uniform strengths is ignored, resulting in many unnecessary block misses, or additional communication overhead. To address this issue, we propose a client-directed, coordinated file block placement and replacement protocol, where the non-uniform strengths of locality are dynamically identified on the client level to direct servers on placing or replacing file blocks accordingly on different levels of the buffer caches. In other words, the caching layout of the blocks in the hierarchy dynamically matches the locality of block accesses. The effectiveness of our proposed protocol comes from achieving the following three goals: (1) The multi-level cache retains the same hit rate as that of a single level cache whose size equals to the aggregate size of multi-level caches. (2) The non-uniform locality strengths of blocks are fully exploited and ranked to fit into the physical multi-level caches. (3) The communication overheads between caches are also reduced.

1. Introduction

1.1. Hierarchical Caching and its Challenges

With the ever-widening gap between the speeds of processors and hard disks, practitioners try to make a full use of the available buffer caches along a file block retrieving route for the purpose of satisfying the requests before they reach disk surfaces. Besides the buffer caches at clients, the requested blocks can also be cached at server buffer caches and disk built-in caches, which form a multi-level buffer cache hierarchy. For example, modern high-end disk arrays typically have several gigabytes of cache RAM. Though multiple buffer resources are lined up and their aggregate size is increasingly large, the issue of how to make them work together effectively to deliver the expected performance commensurate to the aggregate size of the distributed buffer caches is still not well addressed. There are two new challenges related to this issue.

The first challenge comes from the weakened locality in the low level buffer caches[1]. Caching works because of the existence of locality, which is an inherent property of application workloads. Only the first level buffer cache is exposed with the original locality and has the highest potential to exploit it. Low level caches hold the misses from their upper level buffer caches. In other words, the stream of access requests from applications is filtered by the high level caches before it arrives at the low level ones. Thus the access stream seen by low level caches has weaker locality than those available to the first level cache. The performance of widely used locality-based replacements such as LRU can be significantly degraded once these replacements are employed in the low level buffer caches. Muntz and Honeyman [7] as well as Zhou et al [14] have observed the serious performance degradation in their file server buffer cache studies. In a work to investigate the cost-effectiveness of disk built-in caches for desktop PCs, Zhu and Hu found that the built-in caches contribute little more to the average response time reduction when its size exceeds 512KB with a client cache size of 16MB [13]. The above cited work indicates applying a replacement independently at a low level buffer cache can lose its chance to exploit the original locality. This motivates us to make replacement decisions based on the original access stream, which is only available at the first level cache.

The second challenge comes from the undiscerning redundancy among levels of the buffer caches. Redundancy means a block is cached and duplicated along its retrieval route in more than one caches. Without a proper coordination among the levels, blocks could reside undiscerningly in multiple buffer caches for a long period of time before they become cold enough to be replaced by a local replacement algorithm. The redundancy can cause the buffer cache hierarchy seriously under-utilized. Even if the aggregate size

∗ This work is supported in part by the U.S. National Science Foundation under grants CCR-0098055 and ACI-0129883.

1 By low level buffer caches, we customarily refer to the caches not close to the workload running clients. Similarly, high levels of buffer caches are those close to the clients. Thus, the first level buffer cache is the client buffer cache with the highest level.

of the multi-level buffer caches could hold the working set, the hierarchy would behave as if it were as big as the single level of cache with the largest size under some access patterns. We propose to use an unified replacement scheme for a multi-level cache hierarchy, which can determine an appropriate place for a block to be cached (if it needs being cached). Thus undiscerning redundancy can be eliminated. The hierarchy can perform as an unified cache with the size equivalent to the aggregate size, so that all the cache spaces are fully utilized.

Wong and Wilkes [12] simply apply an unified LRU algorithm in a two-level buffer cache: client and disk array built-in buffer caches. The first portion of the LRU stack corresponds to the client cache. The second portion of the LRU stack corresponds to the disk array cache. Any blocks moving from the first portion into the second portion due to the increased recency would incur a demotion, an operation that transfers a block from the current level to its next low level cache. Since any recently referenced blocks are brought into the top of LRU stack, all newly referenced blocks are cached in the first level cache. There is no explicit block placement arrangement adapting to their access pattern. By demotion, the blocks go back to the disk cache. Though their scheme has an significant advantage over independent replacements, a critical weakness exists: a large number of demotions. It has been shown that the benefits can be nullified by them once the I/O bandwidth is below a certain threshold [15].

1.2. Our Principles to Address the Challenges

To address the limitations in the unified LRU scheme, a caching algorithm for block placement and replacement in the multi-level buffer hierarchy should have two abilities: (1) distinction of locality strengths; and (2) stability of the distinction, which are also our two principles to address the challenges. Regarding the distinction, if the algorithm can accurately and responsively distinguish blocks with strong locality from those with weak locality[2], then the stronger the locality of blocks is, the higher level of cache they should be placed on. The distinction of this hierarchical locality will make high levels of caches contribute most to the hit rates, which reduces the average access time because of their low hit times. Since the arrangement of block caching positions is based on the distinction of locality strengths, we need to rearrange the blocks once the locality strengths change, which means to transfer blocks among levels. This incurs a communication cost. Thus the stability of the distinctions is critical to keep a low communication cost introduced by an unified caching scheme.

2 By a block with strong locality, we mean it is highly likely to be referenced soon, and it contributes more to the hit rate by being cached than the one with weak locality. The strengths of locality are quantifi ed differently in different replacements, which we will discuss in the next section.

2. Characterizing Non-uniform Locality Strengths in Hierarchical Buffer Caching

2.1. Criteria to Distinguish Locality Strengths

Each replacement algorithm has its own defined criterion to characterize locality strengths and to make distinctions among them. Among these criteria, the one used in the optimal replacement algorithm, OPT, provides the most accurate distinction of locality strengths among accessed blocks. It uses the period of time between the current reference and the next reference to a block, we call it next distance (ND), to determine the locality strengths. Using the time of the last reference to a block to predict the time of its next reference, the LRU algorithm uses Recency (R), which is also its current position in the LRU stack, to simulate ND. Both ND and R of a block could change with every reference to any block. When the stability of the locality strengths characterized by ND or R is a concern, it is unclear where a block should be cached to reduce the communication cost.

In the unified LRU replacement [12], when a block goes down in the LRU stack with the ongoing references, it may incur demotions once its recencies reach certain values. Had it been known at what recency a block would be re-accessed when the block was requested, we would have cached it directly on the level of cache corresponding to that recency, thus the demotions could be avoided. This motivates us to use this recency at which the block will be referenced next time, which we call Next Locality Distance (NLD), to characterize locality strengths. After a block is accessed, its NLD will not change until its next reference. This helps to stabilize the distinction of locality strengths. Because NLD represents a future access timing, it is not collectible on-line. To simulate NLD in an on-line algorithm, we use Last Locality Distance (LLD), the recency at which a block was accessed last time. However, LLD of a block does not describe the number of recent references after the last reference to the block, which is reflected in its recency. To responsively capture the changes of locality scope (a hot block becomes cold, or vice versa), we use recency to take place of LLD once recency exceeds LLD. That is, we use the larger of LLD and R to simulate NLD, called LLD-R. Using the LRU stack, Figure 1 illustrates the differences of ND, R, NLD, and LLD-R.

2.2. Comparisons of ND, R, NLD, and LLD-R on Distinguishing Locality Strengths

Each of the four measures, ND, R, NLD, and LLD-R, is associated with a replacement algorithm. A replacement algorithm works in the way that it has its accessed blocks ranked according to a certain measure, and selects the least ranked block for replacement once a victim block is needed. For example, the measure used by OPT is ND and the measure used by LRU is R. How well a measure satisfies the two ability requirements — distinction of locality strengths and the stability of the distinction, determines how well the cor-

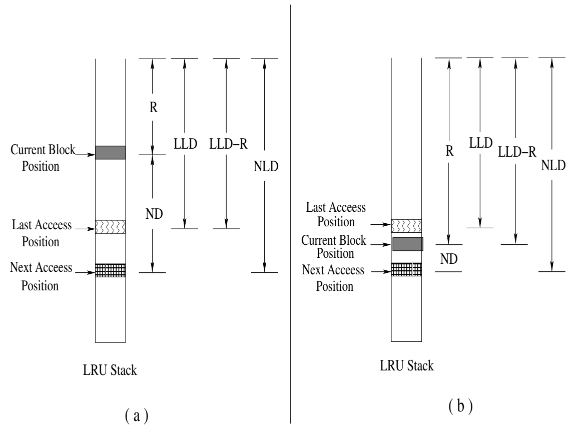

Figure 1. **In the LRU stack, the positions (recencies) of last access, next access and current time of a block are illustrated. Before the current position of the block (R) exceeds last access position (LLD) (see left figure (a)), LLD-R is LLD; after that (see right figure (b)), LLD-R is R to make LLD-R more accurately simulate NLD. It also shows that R and ND change with every reference.**

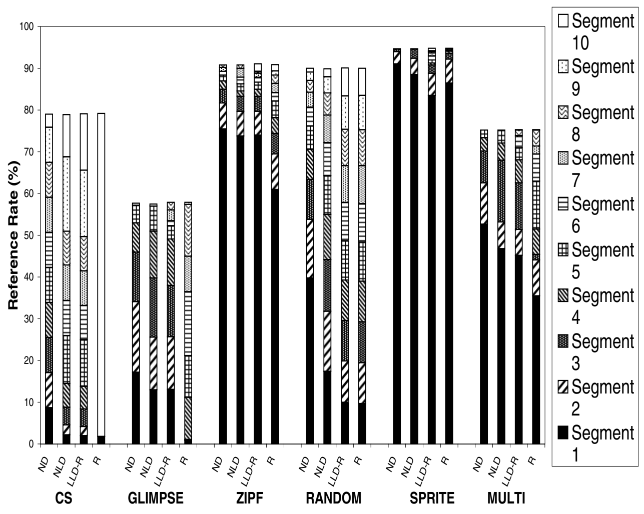

Figure 2. **Reference ratios to each of the segments (the ratios of the number of references to a segment to the number of all references in a workload). It also shows the accumulative reference rates for the first N segments in each workload, where N is 1 through 10.**

responding replacement algorithm serves as an unified replacement algorithm for a multi-level cache hierarchy.

To understand and compare the two abilities of the measures, we use six small-scale workload traces (cs, $glimpse, zipf$, $random, sprite$, and $multi$) with representative access patterns for the evaluation. The descriptions for the traces can be found in [6]. These traces represent the major access patterns common to the I/O requests. Traces cs and $glimpse$ have a looping access pattern, where all blocks are regularly and repeatedly accessed. Trace $sprite$ has a temporally-clustered access pattern, where blocks accessed more recently are the ones more likely to be accessed soon. It is an LRU-friendly pattern. Trace $random$ has a spatially uniform distribution of references across all the accessed blocks. This access pattern is common in database applications. In trace $zipf$ only a few blocks are frequently accessed. Formally, the probability of a reference to the ith block is proportional to $1/i$. Zipf-like access patterns exhibited in trace $zipf$ are typical for file references in Web servers. Trace $multi$ has an access pattern mixed with sequential, looping and probabilistic references.

We maintain an ascendingly ordered list for each measure. Once there is a reference to a block, the measure value of the block, and possibly the measure values of other blocks are changed, and the list is updated to maintain the order. We divide the full length of each list into 10 segments of equal size. We collect the number of references to each segment to observe the locality strength distinction. We also collect the block movements across each of the segment boundaries to observe the stability of the distinctions when the list is updated with references.

Figure 2 shows the reference rate distributions in the list for each measure. Each of the measures orders accessed blocks in its list and places the blocks with small values at the head of the list. A good distinction of locality strengths should generate a reference rate distribution with more hits appearing in the head portion of the list than those in its tail portion. Assuming each of the segments corresponds to a level of cache, we can observe the hit rate on each level of cache. From the figure we have the following observations:

(1) ND gives the best reference rate distribution. The higher (closer to the list head, and with a $smaller$ segment number in Figure 2) a segment is, the higher reference rate the segment achieves for ND. This reflects the strong ability of ND to accurately make the distinction of locality strengths. Actually, the distribution generated by ND is optimal considering optimality of the OPT algorithm. While high segments are mapped on the high levels of caches, which have small hit times, such a distribution helps reduce the average access time. However, R gives the worst distribution, though it attempts to directly simulate (predict) ND. This is most obvious for the workload with a looping access pattern: cs and $glimpse$. Most of their references go to the low segments (after Segment 9 in cs, and after Segment 3 for $glimpse$). R only performs well on the workloads with an LRU-friendly access pattern, such as $sprite$.

(2) NLD performs well for all the workloads with various access patterns. This reflects its ability to make consistently accurate distinction. Except for trace $random$, LLD-R performs very closely to NLD, though it does not depend on future knowledge. Without looking ahead, all the on-line al-

	ND	R	NLD	LLD-R
Ability to distinguish locality strengths	strong	weak	strong	strong
Stability of distinctions	weak	weak	strong	strong
On-line measures	no	yes	no	yes

Table 1. Comparisons of the four measures.

gorithms could perform the same as RANDOM replacement for trace $random$ at most, which randomly selects a block for replacement and has a hit rate proportional to the cache size. Both LLD-R and R obtain such a distribution for the trace.

(3) For the two on-line measures, LLD-R produces significantly better locality distinctions than R for workloads cs, $glimpse$, $zipf$, and $multi$. For LRU-friendly workload $sprite$, both R and LLD-R perform very well, and R performs a little better than LLD-R.

Figure 3 shows block movement ratios of the number of block movements at each of the segment boundaries to the number of all references for each of the four measures. For example, the first point from the left on a curve represents the ratio of the number of times that the blocks cross the boundary between the first and second segments to that of all references. A small movement ratio means a high stability for the distinction of locality strengths. When the segments are mapped to the levels of caches and a boundary corresponds to the separation of two adjacent levels of caches, a movement ratio determines the communication overhead in unified caching. Then we have the following observations:

(1) ND and R have the highest movement ratios, which have been expected because of their volatility. Comparatively, NLD and LLD-R have much lower movement ratios.

(2) The ratio gaps between NLD (resp. LLD-R) and ND (resp. R) are especially pronounced with the looping pattern trace $glimpse$. However, even for the LRU-friendly workloads like $sprite$ and $zipf$, the gaps are still considerably large. This demonstrates that an on-line unified caching based on LLD-R can promise a much smaller additional communication cost than that based on R.

(3) The ratios of LLD-R are smaller than those of NLD in most cases.

Table 1 summarizes the four measures distinguishing locality strengths, showing that using LLD-R is a desired basis to building an unified caching protocol.

3. The Unified and Level-aware Caching (ULC) Protocol

3.1. An Executive Summary

We have shown that the position of a block in the list ordered by LLD-R provides a strong hint for placing the block on a level corresponding to its list position, or not caching it at all[3]. This also assures us that the block would still stay there with a high probability when the block is accessed next time. Effectively using the hint, we propose a multi-level buffer placement and replacement protocol, called *Unified and Level-aware Caching* (ULC) protocol to effectively exploit hierarchical locality. Based on the access patterns and available cache sizes on each level, ULC running at the first level client dynamically ranks the accessed blocks into levels L_1, L_2, ..., and L_{out} according to their LLD-R positions, thus directing them to be placed (cached) at level L_1 cache, level L_2 cache, ..., or not cached at any levels at the time of the retrieval, respectively. The size of the first level cache determines the number of L_1 blocks, those with the smallest LLD-R values, and the same relationship holds for other levels of caches. Low level buffer caches are not responsible for extracting locality from the filtered request stream presented to them any more. Every block request from the high level buffer cache carries a level tag, so the low level caches only take their actions accordingly. If the attached level tag matches its level number, this level will cache the retrieved block. Otherwise, the block is discarded after it is sent to its next upper level cache. When the block positions need adjusting, the client sends block demotion instructions to low level caches, which lets a block originally residing in a cache be demoted into its next low level cache. Our client-directed protocol attempts to answer the following questions: (1) how to exploit the locality in the entire buffer cache hierarchy thoroughly and consistently; (2) how to make the exploited locality usable by all buffer caches in the hierarchy; and (3) how to minimize the overhead of the protocol.

3.2. A Detailed Description

In Section 2.2 we have shown the LLD-R measure is a promising basis on which to build a multi-level caching protocol. However, an implementation exactly based on LLD-R ranking criterion will take at least O(log$n$) time, where n is the number of distinct accessed blocks. This is the cost of block ordering. In order to develop an efficient algorithm with the time complexity $O(1)$, we transform the process to determine the position of a block in LLD-R ordered list into two separate steps: (1) When a block gets accessed, its recency is 0, so its LLD-R is LLD, which is the recency at which it was just accessed. We use the LLD to determine in which segment the block will be cached at the time of retrieval. (2) Once a block is assigned into a specific segment, we use R to determine its position in the segment. Each segment corresponds to a level of cache, and the size of the segment is the same as that of the cache.

As is shown in Figure 4, the recently accessed blocks are maintained in an unified LRU stack, simplified as $uniLRU$-$stack$. These blocks could be cached in any level of buffer caches, or even not cached[4]. For each level of buffer cache

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

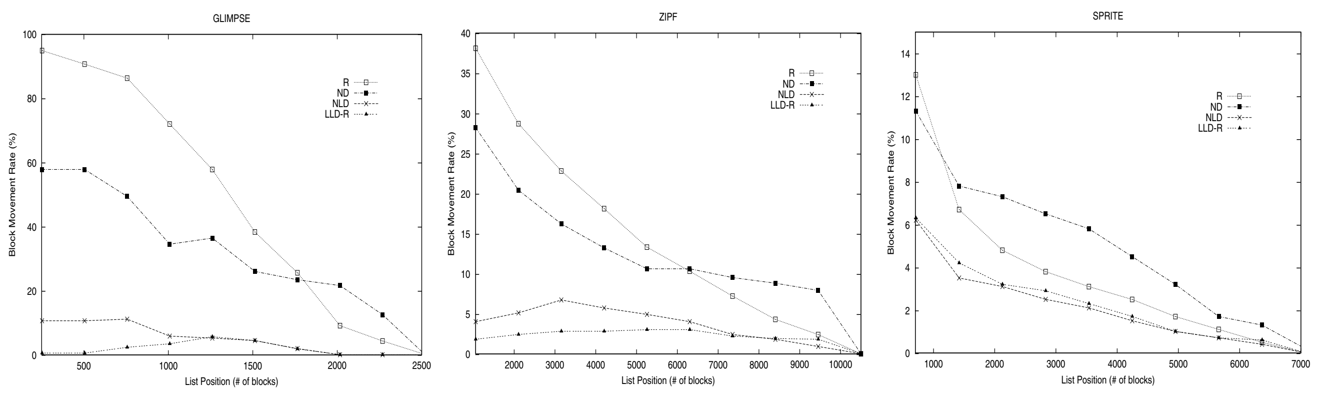

Figure 3. Movement ratio curves showing the ratios between the number of block movements on a position of ordered lists and that of total references for four measures on 3 workloads. The figures for cs, $random$ and $multi$ can be found in [6].

there is a yardstick block in $uniLRUstack$, which is the block cached in that level of the cache and has the maximal recency among blocks cached in that level. We call them Y_1, Y_2, ...,Y_n for level L_1, L_2, ..., L_n cache, respectively. The size of $uniLRUstack$ actually is determined by the position of Y_n, the last yardstick, which always sits in the bottom of $uniLRUstack$. Any blocks with recencies larger than that of Y_n will be removed from $uniLRUstack$ and become an L_{out} block, which is not cached in any level of caches. Only when a block gets accessed with the recency between the recencies of Y_{i-1} and Y_i does the block become L_i block, which means it will be cached in the level L_i cache. All of blocks cached on the same level can be viewed as an LRU stack, called LRU_i, where the order of blocks is determined by their recencies in $uniLRUstack$ 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 L_1 cache nor going to be cached there because their LLDs 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 replaced from the L_1 cache.

There are two structures for the buffer cache hierarchy. One is the single-client structure, in which there is only one client connected to one server[5], and another is the multi-client structure, in which more than one clients share the same server, and blocks requested by different clients are shared in the server. There are two additional challenges for the multi-client ULC protocol: (1) How to cache shared blocks in server buffer caches, which could carry different level tags set by different clients. (2) How to allocate server cache buffers to different clients.

3.2.1. The Single-client ULC Protocol The single-client ULC algorithm runs at the client, which holds the first level cache. It has the knowledge of the size of the buffer cache on each level. For each block in $uniLRUstack$, there are two associated statuses: level status and recency status. Level status indicates at which level the block is cached, such as L_1, L_2, ..., L_n, or L_{out}. When a block gets accessed, we need to know its recency to determine its level status, as the recency is going to be its LLD. It takes at least O(N) time to maintain the exact recency information for all blocks, where N is the aggregate size of the buffer caches. Actually we only

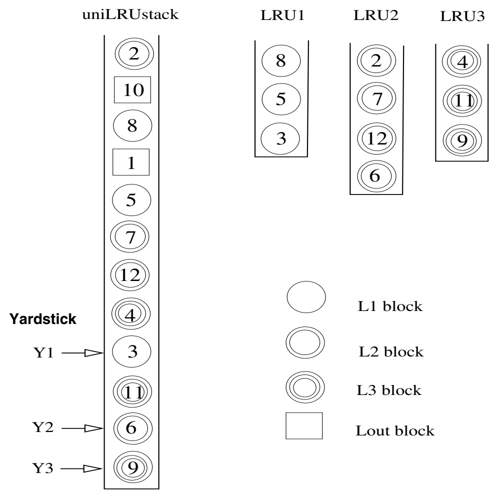

Figure 4. An example to show the data structure of ULC for a 3-level hierarchy. The blocks with their recencies less than that of yardstick Y_3 are kept in $uniLRUstack$. The level status (L_1, L_2 or L_3) of a block is determined by its position between two yardsticks where it was accessed last time. Its recency status (R_1, R_2 or R_3) is determined by its position between two yardsticks where it sits currently. To decide which block should be replaced in each level, the blocks in the same level can be viewed to be organized in a separate LRU stack (LRU_1, LRU_2, or LRU_3), and the bottom block is for replacement.

4 In a protocol implementation, only some metadata, such as a block identifi er and two statuses used in the ULC protocol, are stored in the stack for each block, not the block itself.

5 Here we call the high level buffer cache, client, and low buffer cache, server, when we discuss two adjacent levels.

need to know the recencies of whatever two yardsticks the recency lies in. Thus we maintain a recency status R_i for each block, which means its recency is between the recencies of yardsticks Y_{i-1} and Y_i (or just less than Y_i if i is 1). The cost to maintain recency statuses is O(1), which will be explained.

Initially, if level L_i is not full and the levels that are higher than it 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 $uniLRUstack$ are given level status L_{out}. There are two circumstances for a block not in $uniLRUstack$. 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 so that it leaves $uniLRUstack$ from the bottom. For these blocks their level status is L_{out}, and recency status is R_{out}.

We define an operation for yardsticks in $uniLRUstack$ called $YardStickAdjustment$, which moves a yardstick from the current yardstick block with level status L_i in the direction towards 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}. When a yardstick block changes its position in $uniLRU$-$stack$, we need to conduct yardstick adjustment to ensure the yardstick is on the block with correct recency status and with the largest recency among the blocks on the 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 $uniLRUstack$. To place the block at the correct recency position in LRU_{i+1}, we define another operation for a demoted block called $DemotionSearching$, which searches in the direction towards the stack bottom in $uniLRUstack$ for next block with a higher level status.

There are two types of requests in ULC, which are sent from the client to the low level caches to coordinate various levels of caches to work under an unified caching algorithm.

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(b, i, j) ($i < j$): demote block b from level L_i into level L_j.

If there is a reference to block b with level status L_i and recency status R_j, there are only two cases we need to deal with: $i = j$ and $i > j$. The case $i < j$ is not possible because block b is demoted to level L_{i+1} before j is larger than i. When block b is referenced, it is moved to the top of $uniLRUstack$ and its recency status becomes R_1. This also makes it stay in the top of stack LRU_i. If $i > 1$, block b goes to stack $tempLRU$ in the client and is going to be replaced soon from the client cache. Then for each of the two cases, we act as follows: (1) $i = j$. Block b remains in its current level of cache with the same level status (Retrieve(b, i, i)). (2) $i > j$. 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 its next low level cache, whose yardstick block may have to be demoted in turn if its status level is higher than L_i. Yardstick adjustment and demotion searching are conducted here.

3.2.2. The Multi-client ULC Protocol When there are multiple clients sharing one server, the cache buffers in the server are no longer solely used by one client. In the single client ULC protocol, the number of the blocks with level status L_i (also called L_i blocks), or the size of stack LRU_i, is determined by the size of level L_i cache. If the buffers at level L_i are shared by multiple clients, an allocation policy is needed on level L_i for the performance of the entire system. To obtain best performance, it is known that allocation should follow the dynamic partition principle: each client should be 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 [3]. 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 be cached in this server. A block is cached on the highest level among all the clients' direction. If there is only one client, the bottom block of $gLRU$ is always the yardstick block Y_i in $uniLRU$-$stack$, and also is the bottom block of stack LRU_i in the client. Because the server cache buffer 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 decreased 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. Then they are piggybacked on the next retrieved block, thus saving extra messages. Figure 5 shows an example to illustrate the multi-client case.

4. Performance Evaluation

This section presents our trace-driven simulation results. We compare ULC with two other multi-client caching schemes: independent LRU, simplified as $indLRU$, which is a commonly used scheme, and unified LRU, simplified as $uniLRU$, an LRU-based unified caching protocol[12].

4.1. Performance Metric

We use average block access time, T_{ave}, to evaluate the performance of various protocols. This metric measures the average time required to access a block perceived by applications. The access time is determined by the hit rates

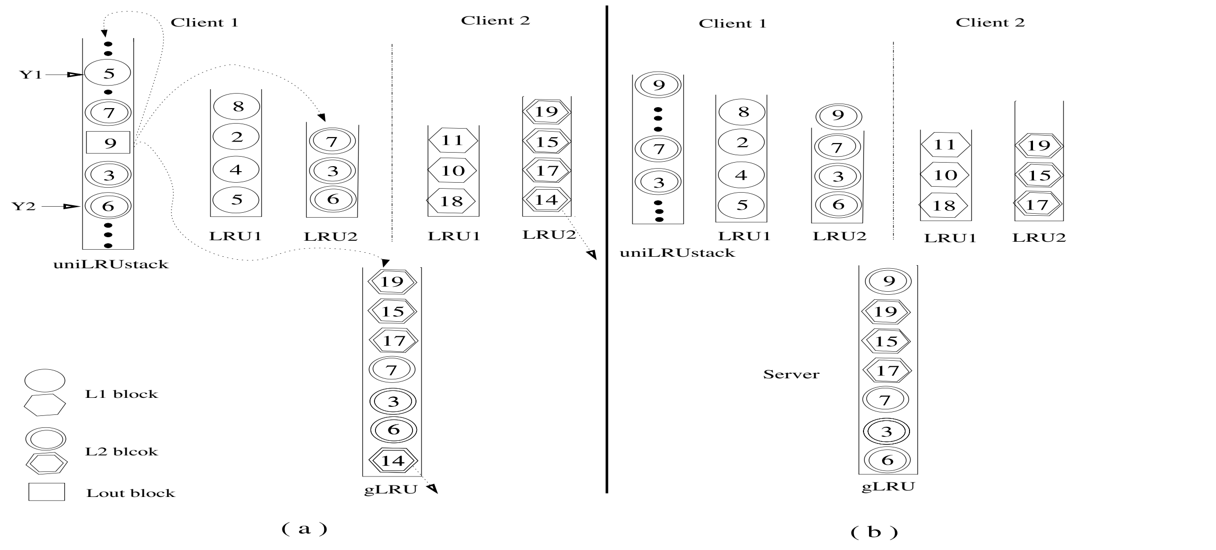

Figure 5. **An example to explain how a requested block is cached in the server cache, and how the allocation scheme adjusts the size of the server cache used by various clients in a multi-client two-level caching structure. Originally in (a) server stack $gLRU$ holds all the L_2 blocks from clients 1 and 2, which are also in their LRU_2 stacks, respectively. Then block 9 is accessed in client 1. Because block 9 is between yardstick Y_1 and Y_2 in its $uniLRUstack$, it turns into 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 buffers re-allocation (b), the size of server cache for client 1 is increased by 1 and that for client 2 is decreased by 1. So the clients and the server cooperate to make the server cache efficiently allocated with the aim of high performance for the entire system.**

and miss penalties at different levels of the caching hierarchy, as well as other communication costs. Generally, we can estimate 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 rate 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 rate for the cache hierarchy (equivalent to $1 - \sum_{i=1}^{n} h_i$), T_m is the cost for the miss, and $T_{demotion}$ is the demotion cost for block placements required by an 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, thus their costs could be hidden from applications, for two reasons: (1) Demotions are highly possible to occur in a bursting fashion, especially for an LRU-based unified replacement, where 50%, even around 90% of the references incur demotions. A small number of dedicated buffers have difficulty in buffering the delayed blocks, thus its performance is unpredictable. (2) Reserving a large number of buffers for delayed demotions actually reduces the cache size and would hurt the hit rates.

4.2. Simulation Environment

In our trace-driven simulation experiments we assume 8 KB cache block. We use five large scale traces to drive the simulator, including two synthetic traces: $random$ and $zipf$ and five other real-life workload traces. We have described the two synthetic traces in Section 2. Here we significantly increase the scale of these two traces: $random$ accesses 65536 unique blocks with a 512MB data set. It contains about 65M block references. $zipf$ accesses 98304 unique blocks with a 768MB data set. It contains about 98M 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. [9]. The size of the data set served was 524 MB which is stored in 13,457 files. A total of about 1.5M 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 Redhat Linux 6.2 desktop [2]. It contains text editor, compiler, IDE, browser, email, and desktop environment usage. It has around 100K references. The size of the data set it accessed is around 600M.
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 Redhat Linux 7.1[2]. It has around 3.9M references. The data set size is around 256M.

We also select three traces for multi-client simulation. One of them is the original $httpd$ trace with seven access streams, each for one client. The other two multi-client 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 [12]. The system has 6 HP 9000 K580 servers running HP-UX 10.20. The size of the data set accessed by all six clients is 18.6G.
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 [9]. 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.

4.3. Comparisons of Multi-level Schemes in a Three-level Structure

To demonstrate the ability of multi-level caching schemes (ULC, indLRU, and uniLRU) to make distinctions of locality strengths as well as the ability to keep their stability, we test them in a three-level caching hierarchy for the five single client traces, simulating a scenario where the block transfer route consists of a client, a server and its disk array containing a large RAM cache. For a common local network environment, we assume the cost to transfer an 8KB block between the client and the server through LAN is 1ms, 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 [12]. We assume the cache sizes of the client, the server, and the disk array are 100MB each for traces $random$, $zipf$, *httpd*, and $dev1$, and the cache sizes are 50MB each for trace *tpcc*1 due to its comparatively small data set. We report the hit rates in each of the three levels, demotion rates on each boundary, and average access time for each workload with the three multi-level caching schemes in Figure 6.

Confirming the experimental results in [12], we observe that there are significant performance improvements of uniLRU over indLRU for all the five traces, from 17% to 80% 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 rates; (2) generating additional demotion cost. UniLRU eliminates the redundancy in the hierarchy, making the low levels of caches contribute to the hit rate just as if they stayed in the first level. For example, in a random access pattern, the contribution of a cache to the hit rate should be proportional to its size. However, the second and third levels of caches gain much lower hit rates (1.7% and 0.3% respectively) than that of first level cache (19.5%) for trace $random$ in indLRU (see the first graph). The unified replacement scheme uniLRU makes the low levels of caches much better utilized. Their hit rates (19.6% and 19.5% respectively) are almost the same as that of first level cache (19.5%). However this improvement comes with a considerably high price: high demotion rates. For example, in trace $random$ uniLRU has a first boundary demotion rate 80.5%, which means 80.5% of block references accompany "write-backs" to the server. Furthermore, it has a 60.9% demotion rate at the second boundary (see the second graph). The worst case for the demotion rates of uniLRU is trace *tpcc*1, which has a looping access pattern. Its first boundary demotion rate is 100%! This is because uniLRU has little power 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 *tpcc*1 are accessed after they are demoted into the second level of cache. So the hit rate of the second level cache is very high (92.5%) and 44.7% of the average access time is spent on the demotion. According to the requirement on the ability of distinguishing locality strengths for a multi-level caching scheme, the distribution that the level L_1 hit rate (0.03%) is much less than the L_2 hit rate (92.5%) under uniLRU shows a bad case.

Compared with uniLRU, ULC protocol has an access-time-aware hit rate distribution along the levels of caches: more hits appearing on upper levels. For example, the hit rates of the level L_1, L_2, and L_3 are 50.3%, 45.1%, and 3.4%, respectively for trace *tpcc*1. And such a distribution is achieved without paying high costs of demotions. For example, the two boundary demotion rates of *tpcc*1 are 1.4% and 1.3%, respectively (see the second graph). It is also shown that ULC has significant demotion rate reductions over uniLRU for all 5 traces. This explains why the proportion of demotion cost in the average access time for ULC is much smaller (from 1% to 8.3% with an average of 4.1%) than that for uniLRU (from 12.6% to 44.7% with an average of 21.5%) (see the third graph). The access time breakdowns also show that ULC still performs significantly better than uniLRU except for trace $random$, even if we assume the demotions could be moved off the critical path for response time. Actually this is an unrealistic assumption. The experiments on the client-server system running a TPC-C benchmark show that demotions can significantly delay the network and lower the system throughput [15]. In summary, our ULC achieves from 11% to 71% reduction on average access time with an average of 34.6% over that of uniLRU.

4.4. Comparisons of Caching Schemes for Multi-client Workloads

Because the performance of uniLRU scheme can significantly deteriorate due to buffer competition and data sharing among clients for the multi-client structure, Wong and Wilkes also proposed two adaptive cache insertion policies to supplement their primitive scheme [12]. Among their three multi-client traces *httpd*, *openmail*, and *db*2, *httpd* is the one with data sharing. While they did not state which version of their unified LRU schemes should be used for a specific workload, we ran all the versions and report the best results for comparisons. For the multi-client scheme evaluation, we include Multi-Queue (MQ), a replacement algo-

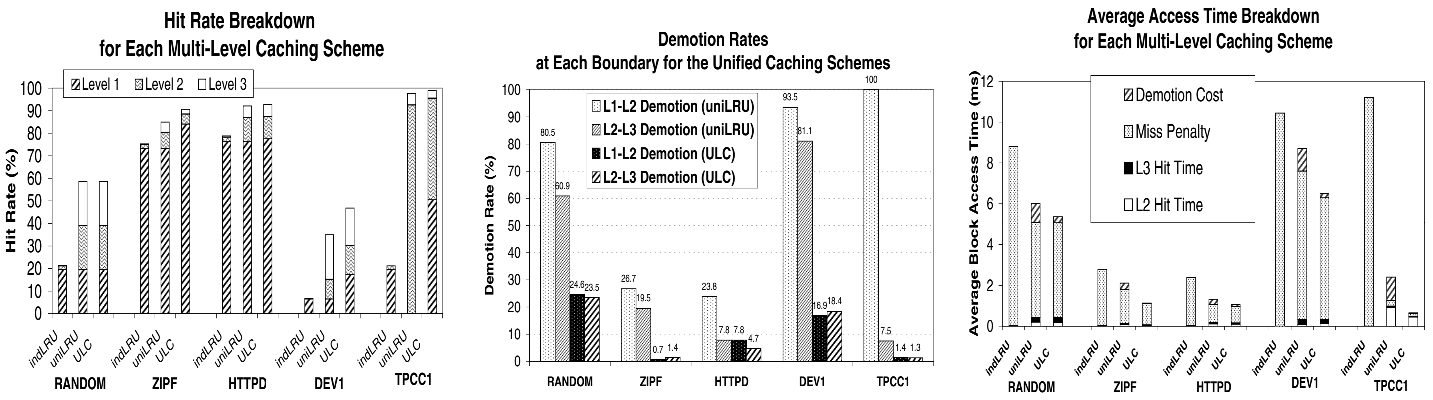

Figure 6. Hit rates in each of the three levels, demotion rates at each of two boundaries (between L1 and L2, and between L2 and L3 cache), and average access time for each workload with the multi-level caching schemes indLRU, uniLRU and ULC.

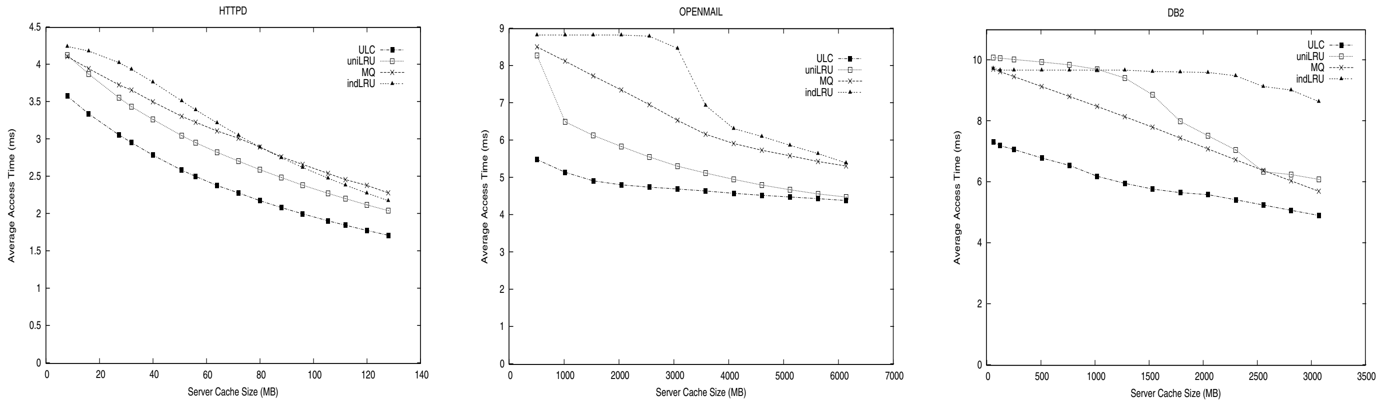

Figure 7. Average access times of multi-client traces $httpd$, $openmail$, and $db2$ with various server cache sizes. Among them $httpd$ is with 7 clients, $openmail$ is with 6 clients, and $db2$ is with 8 clients. Each client contains 8MB, 1GB, or 256MB respectively.

rithm dedicated for second level buffer caches[14]. To overcome LRU's weakness with weak locality, MQ sets up multiple queues and uses access frequencies to determine which queue a block should be in. In the client-server caching hierarchy, the environment that MQ is designed for, we use MQ in the server and use LRU in the client independently.

Figure 7 shows that for all the workloads ULC achieves the best performance. For most of the time, indLRU has the worst performance. However, there are two cases where indLRU beats uniLRU or 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 *db*2. This is because *db*2 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 rates (6.9% and 7.9%, respectively for the two levels of caches, compared with that of MQ (12.3%) and that of ULC (35.1%). Thus it is the large demotion cost of uniLRU (with an average demotion rate 88.6% for the 8 clients, compared with that of ULC (7.2%)) 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 the traces *httpd* and *openmail*, uniLRU beats MQ by eliminating data redundancy.

5. Related Work and Discussions

Addressing the challenges of replacements in buffer caching hierarchy, researchers have mainly tried these two approaches: (1) re-designing low level cache replacement; (2) extending existing replacement into an unified hierarchy replacement through coordination. The Multi-Queue (MQ) [14] and Unified LRU [12] are two representatives of these two approaches, respectively. To reduce the high traffic caused by demotions in Unified LRU, Chen et al [15] proposed to re-load evicted blocks from disks rather than from clients. Our technique deals with the reduction of demotions by effectively utilizing history access patterns.

Jiang and Zhang [5] propose the LIRS replacement algorithm to address the performance degradation of LRU on workloads with weak localities. They use a LIRS stack to track the recencies of accessed blocks. The blocks with small recencies at which they get accessed, are kept in the cache. This single-level cache replacement motivates us to investigate if the last locality distance, LLD, can be effectively used to exploit hierarchical locality, so that blocks with different locality strengths can be arranged into correct cache levels.

The work on cooperative caching [4, 8, 10] is to coordinate the buffer caches of many client machines distributed on a LAN to form a fourth level in the network file system's cache hierarchy. Besides local cache, server cache, server disk, data can also be cached in another client's cache. Some associated issues include idle cache availability, consistent sharing. Our ULC protocol is intended for the conventional file buffer cache hierarchy, while the characterization of non-uniform locality is expected to enhance the effectiveness of data placements in the cooperative caching.

As far as the cache hierarchy between processor and memory is considered, the interaction of replacements at various levels and its performance implication do not pose a problem. Multi-level inclusivity between L_1, L_2, $..L_n$ cache could be accepted as a principle to simplify the cache coherence protocol [1]. This is because a lower level cache is more than ten times larger than its upper level cache. With this large difference, the size reductions of useful caches due to data redundancy have only limited negative performance impact on the low level caches. In contrast, the sizes of buffer caches in the hierarchy do not follow this regularity: a client buffer cache could even be larger than the second level cache.

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

The underlying algorithms on almost all the existing file systems are LRU or its variants. ULC basically inherits their data structure – LRU stack. The operation costs associated with the stacks are $O(1)$ time with each reference request. Regarding space cost used for the stacks, we need 17 bytes (8 bytes for file identifier and block offset, 8 bytes for two pointers in a double linked list, and 1 byte for statuses) for a block in the client, which only represents 0.2% of an 8 Kbytes block. The metadata in the shared server cache needs additional one or two bytes for recording block owner. The stack sizes on other levels except the first one are determined by their cache sizes. Thus a server with a 1GB cache only uses 2.2MB for its metadata. The first level cache has to hold $uniLRUstack$, whose actual size is determined by the working set size of applications running on the client. The relatively cold blocks (with low level statuses) can be trimmed from the stack without compromising the ULC locality distinction ability if needed to save space cost. E.g. an 8.5MB metadata in the client can support a workload working set up to 4GB. This is highly affordable in a system endeavoring for improved file I/O performance through large caches.

6. Conclusions

An effective caching scheme for multi-level cache hierarchy is important to the performance of applications because increasingly more applications rely on the hierarchy for their file accesses. For this purpose we have proposed the ULC caching protocol with the distinguished performance merits: (1) It significantly reduce average block access time perceived by applications; (2) It can be implemented efficiently with a cost comparable with that of LRU.

References

[1] J.-L. Baer, and W.-H. Wang, "On the Inclusion Properties for Multi-level Cache Hierarchies", *Proceedings of Annual International Symposium on Computer Architecture*, 1988.

[2] Trace Distribution Center, Performance Evaluation Laboratory, Brigham Young University. http://tds.cs.byu.edu/

[3] P. Cao, E. W. Felten and K. Li, "Application-Controlled File Caching Policies", *Proc. of the USENIX Summer 1994 Technical Conference.*

[4] M. D. Dahlin, R. Y. Wang, T. E. Anderson, D. A. Patterson. "Cooperative Caching: Using Remote Client Memory to Improve File System Performance", *In Proc. of First Symposium on OSDI*, 1994.

[5] S. Jiang and X. Zhang, "LIRS: An Effi cient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance", *In Proc. of ACM SIGMETRICS 2002*, June 2002.

[6] S. Jiang and X. Zhang, "File Caching in Multi-level Buffer Cache", Tech. Rept, College of William & Mary, Feb, 2003

[7] D. Muntz and P. Honeyman, "Multi-Level Caching in Distributed File System – or – Your Caching ain't nuthin' but trash", *Proc. USENIX Winter Technical Conference*, 1992

[8] P. Sarkar and J. Hartman, "Effi cient cooperative caching using hints", *In Proc. of Third Symposium on OSDI*, 1996

[9] M. Uysal, A. Acharya, and J. Salts, "Requirements of I/O Systems for Parallel Machines: An Application-driven Study", *Tech. Rep. CS-TR-3802, Univ. of Maryland*, 1997.

[10] 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", *In Proc. of ACM SIGMETRICS 1998*, June 1998.

[11] J. Wilkes, "The Pantheon Storage-System Simulator", *Technical Report, HPL-SSP-95-14 rev. 1, HP Lab*, May 1996

[12] T. M. Wong and J. Wilkes, "My Cache or Yours? Making Storage more Exclusive", *Proc. of the 2002 Annual USENIX Tech. Conf.*.

[13] Y. Zhu and Y. Hu, "Can Large Disk Built-in Caches Really Improve System Performance?", *Proc. of ACM SIGMETRICS* , June 2002

[14] Y. Zhou, J. F. Philbin and K. Li, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches", *Proc. of 2001 Annual USENIX Technical Conference*, June 2001.

[15] Z. Chen, Y. Zhou, and K. Li, "Eviction-based Placement for Storage Caches", *Proc. of the 2003 Annual USENIX Tech. Conf.*