# Karma: Know-it-All Replacement for a Multilevel cAche

Gala Yadgar
*Computer Science Department, Technion*

Michael Factor
*IBM Haifa Research Laboratories*

Assaf Schuster
*Computer Science Department, Technion*

## Abstract

Multilevel caching, common in many storage configurations, introduces new challenges to traditional cache management: data must be kept in the appropriate cache and replication avoided across the various cache levels. Some existing solutions focus on avoiding replication across the levels of the hierarchy, working well without information about temporal locality–information missing at all but the highest level of the hierarchy. Others use application hints to influence cache contents.

We present Karma, a global non-centralized, dynamic and informed management policy for multiple levels of cache. Karma leverages application hints to make informed allocation and replacement decisions in all cache levels, preserving exclusive caching and adjusting to changes in access patterns. We show the superiority of Karma through comparison to existing solutions including LRU, 2Q, ARC, MultiQ, LRU-SP, and Demote, demonstrating better cache performance than all other solutions and up to 85% better performance than LRU on representative workloads.

## 1 Introduction

Caching is used in storage systems to provide fast access to recently or frequently accessed data, with non-volatile devices used for data safety and long-term storage. Much research has focused on increasing the performance of caches as a means of improving system performance. In many storage system configurations, client and server caches form a two- or more layer hierarchy, introducing new challenges and opportunities over traditional single-level cache management. These include determining which level to cache data in and how to achieve exclusivity of data storage among the cache levels given the scant information available in all but the highest-level cache. Addressing these challenges can provide a significant improvement in overall system performance.

A *cache replacement policy* is used to decide which block is the best candidate for eviction when the cache is full. The *hit rate* is the fraction of page requests served from the cache, out of all requests issued by the application. Numerous studies have demonstrated the correlation between an increase in hit rate and application speedup [10, 12, 13, 19, 22, 27, 48, 49, 51]. This correlation motivates the ongoing search for better replacement policies. The most commonly used online replacement policy is LRU. Pure LRU has no notion of frequency, which makes the cache susceptible to pollution that results from looping or sequential access patterns [40, 47]. Various LRU variants, e.g., LRU-K [37], 2Q [25], LRFU [28], LIRS [23] and ARC [33], attempt to account for frequency as well as temporal locality.

A different approach is to manage each access pattern with the replacement policy best suited for it. This is possible, for example, by automatic classification of access patterns [13, 19, 27], or by adaptively choosing from a pool of policies according to their observed performance [4, 20]. In *informed caching*, replacement decisions are based on hints disclosed by the application itself [10, 14, 38]. Although informed caching has drawbacks for arbitrary applications (see Section 7), these drawbacks can be addressed for database systems [15, 36, 41]. File systems can also derive access patterns from various file attributes, such as the file extension or the application accessing the file. The Extensible File System [26] provides an interface which enables users to classify files and the system to derive the files' properties. Recent tools provide automatic classification of file access patterns by the file and storage systems [16]. Despite the proven advantage of informed caching, it has been employed only in the upper level cache.

The above approaches attempt to maximize the number of cache hits as a means of maximizing overall performance. However, in modern systems where both the server and the storage controller often have significant

caches, a *multilevel cache hierarchy* is formed. Simply maximizing cache hits on any individual cache in a multilevel cache system will not necessarily maximize overall system performance. Therefore, given a multilevel cache hierarchy, we wish to minimize the *weighted I/O cost* which considers all data transfers between the caches and the cost of accessing each.

Multilevel cache hierarchies introduce three major problems in cache replacement. The first is the hiding of locality of reference by the upper cache [51]. The second is data redundancy, where blocks are saved in multiple cache levels [12, 35]. The third is the lack of information about the blocks' attributes (e.g., their file, the application that issued the I/O request) in the lower level caches [45].

Accesses to the low level cache are misses in the upper level. Thus, these accesses are characterized by weak temporal locality. Since LRU is based on locality of reference, its efficiency diminishes in the second level cache. Policies such as FBR [39], MultiQ [51], ARC [33] and CAR [7] attempt to solve this problem by taking into account frequency of access in addition to recency. MultiQ, for example, uses multiple LRU queues with increasing lifetimes. ARC [33] and its approximation CAR [7] distinguish between blocks that are accessed once and those that are accessed more than once. None of the above policies address the cache hierarchy as a whole, but rather manage the cache levels independently, assuming the upper level cache is managed by LRU.

In *exclusive* caching [29, 49, 51], a data block should be cached in at most one cache level at a time. One way to do this is by means of the DEMOTE operation [49]. The lower level deletes a block from its cache when it is read by the upper level. When the upper level evicts an unmodified block from its cache, the block is sent back to the lower level using DEMOTE. The lower level tries to find a place for the demoted block, evicting another block if necessary.

We propose Karma, a novel approach to the management of multilevel cache systems which attempts to address all of the above issues in concert. Karma manages all levels in synergy. We achieve exclusiveness by using application hints at all levels to classify all cached blocks into disjoint sets and partition the cache according to this classification. We distinguish between a READ, which deletes the read block from a lower level cache, and a READ-SAVE, which instructs a lower level to save a block in its cache (this distinction can be realized using the existing SCSI command set, by setting or clearing the *disable page out* (DPO) bit in the READ command [1]). We also use DEMOTE to maintain exclusiveness in partitions that span multiple caches. By combining these mechanisms, Karma optimizes its cache content according to the different access patterns, adjusting to patterns which change over time.

The hints divide the disk blocks into sets based on their expected access pattern and access frequency. Each set is allocated its own cache partition, whose size depends on the frequency of access to the set, its size, and the access pattern of its blocks. Partitions accessed with higher frequency are placed at a higher cache level. Each partition is managed by the replacement policy most suited for its set. Since the lower cache levels are supplied with the application's hints, they can independently compute the partitioning, allocating space only for partitions that do not fit in the upper levels.

Karma is applicable to any application which is able to provide general hints about its access patterns. Databases are a classic example of such applications, where the access pattern is decided in advance by the query optimizer. We base our experimental evaluation both on real database traces, and on synthetic traces with Zipf distribution. For the real traces, we used the *explain* mechanism of the PostgreSQL database as a source of application hints. For the synthetic workload, we supplied Karma with the access frequency of the blocks in the data set. We simulated a hierarchy of two cache levels and one storage level for comparing Karma to LRU, 2Q [25], ARC [33], MultiQ [51], LRU-SP [10] and Demote [49]. We also defined and implemented extensions to these policies to apply to multiple levels of cache. The comparison is by means of the weighted I/O cost.

Karma compares favorably with all other policies: its use of application hints enables matching the optimal policy to each access pattern, its dynamic repartitioning eliminates the sensitivity to changing access patterns and its exclusive caching enables exploitation of every increase in the aggregate cache size. When the aggregate cache size is very small (less than 3% of the data set), Karma suffers from the overhead of DEMOTE, as demoted blocks are discarded before being re-accessed. For all other cache sizes Karma shows great improvement over all other policies. Karma improves over LRU's weighted I/O cost by as much as 85% on average on traces which are a permutation of queries. On such traces, Karma shows an additional 50% average improvement over the best LRU-based policy (Demote) when compared to LRU and it shows an additional 25% average improvement over the best informed policy (LRU-SP).

The rest of the paper is organized as follows. Our model is defined in Section 2. In Section 3 we define marginal gains, and in Section 4 we describe Karma's policy. The experimental testbed is described in Section 5, with our results in Section 6. Section 7 describes related work. We conclude in Section 8.

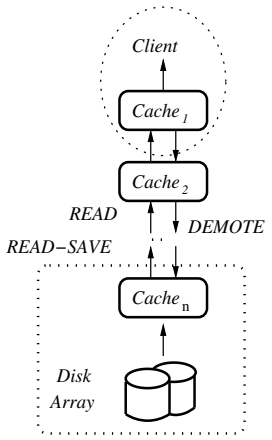



Figure 1: Our storage model consists of $n$ levels of cache, with one cache in each level. The operations are READ and READ-SAVE from $Cache_i$ to $Cache_{i-1}$, and DEMOTE from $Cache_{i-1}$ to $Cache_i$.

## 2 Model Definition

As shown in Figure 1, our model consists of one client, $n$ cache levels, $Cache_1$,...,$Cache_n$, and one storage level, $Disk$, arranged in a linear hierarchy. we refer to more complicated hierarchies in Section 8. $Cache_i$ is of size $S_i$, and access cost $C_i$. The cost of a disk access is $C_{Disk}$. The cost of demoting a block from $Cache_{i-1}$ to $Cache_i$ is $D_i$. We assume that a lower cache level carries an increased access cost, and that demoting and access costs are equal for a given level. Namely, $C_1 = D_1 < C_2 = D_2 < \ldots < C_n = D_n < C_{Disk}$.

Typically, $Cache_1$ resides in the client's memory and $Cache_n$ resides on the storage controller. Additional cache levels may reside in either of these locations, as well as additional locations in the network. The access costs, $C_i$ and $D_i$, represent a combination of computation, network, and queuing delays. $C_{Disk}$ also includes seek times.

The model is demand paging, read-only (for the purpose of this work, we assume a separately managed write cache [18]), and defines three operations:

- READ $(x, i)$—move block $x$ from $cache_{i+1}$ to $cache_i$, removing it from $cache_{i+1}$. If $x$ is not found in $cache_{i+1}$, READ$(x, i+1)$ is performed recursively, stopping at $Disk$ if the block is not found earlier.
- READ-SAVE $(x, i)$—copy block $x$ from $cache_{i+1}$ to $cache_i$. Keep block $x$ in $cache_{i+1}$ only if its range is allocated space in $cache_{i+1}$. If $x$ is not in $cache_{i+1}$, READ-SAVE$(x, i + 1)$ is performed recursively, stopping at the $Disk$ if the block is not found earlier.
- DEMOTE $(x, i)$—move block $x$ from $cache_i$ to $cache_{i+1}$, removing it from $cache_i$.

The *weighted I/O cost* of a policy on a trace is the sum of costs of all READ, READ-SAVE and DEMOTE operations it performs on that trace.

## 3 Marginal Gain

The optimal offline replacement policy for a single cache is Belady's MIN [9]. Whenever a block needs to be evicted from the cache, MIN evicts the one with the largest *forward distance* – the number of distinct blocks that will be accessed before this block is accessed again. To develop our online multilevel algorithm, we have opted to use application hints in a way which best approximates this forward distance. To this end, we use the notion of *marginal gains*, which was defined in previous work [36].

The marginal gain for an access trace is the increase in hit rate that will be seen by this trace if the cache size increases by a single block:

$MG(m) = Hit(m) - Hit(m-1)$,

where $Hit(m)$ is the expected hit rate for a cache of size $m$. Below we show how $MG(m)$ is computed for three common access patterns: looping, sequential, and random. Although we focus on these three patterns, similar considerations can be used to compute the marginal gain of any other access pattern for which the hit rate can be estimated [14, 27, 37, 38].

Obviously, the marginal gain depends on the replacement policy of the cache. We assume that the best replacement policy is used for each access pattern: MRU (Most Recently Used) is known to be optimal for sequential and looping references, whereas LRU is usually employed for random references (for which all policies perform similarly [10, 13, 14, 15, 27, 41]).

**Sequential accesses.** For any cache size $m$, since no block is previously referenced, the hit rate for a sequential access trace is $Hit_{seq}(m) = 0$. Thus, the resulting marginal gain is 0 as well.

**Random (uniform) accesses.** For an access trace of $R$ blocks of uniform distribution, the probability of accessing each block is $1/R$ [36]. For any cache size $m \leq R$, the hit rate is thus $Hit_{rand}(m) = m/R$. The resulting marginal gain is:

$$MG_{rand}(m) = \begin{cases} m/R - (m-1)/R = 1/R & m \leq R \\ 0 & m > R. \end{cases}$$

**Looping accesses.** The *loop length* of a looping reference is the number of blocks being re-referenced [27]. For a looping reference with loop length $L$, the expected hit rate for a cache of size $m$ managed by MRU is $Hit_{loop}(m) = min(L, m)/L$. Thus,

$$MG_{loop}(m) = \begin{cases} m/L - (m-1)/L = 1/L & m \leq L \\ L/L - L/L = 0 & m > L. \end{cases}$$

In other words, the marginal gain is constant up to the point where the entire loop fits in the cache and from there on, the marginal gain is zero.

We deal with traces where accesses to blocks of several ranges are interleaved, possibly with different access patterns. Each range of blocks is accessed with one pattern. In order to compare the marginal gain of references to different ranges, we use the frequency of access to each range. Let $F_i$ be the percent of all accesses which address range $i$. Define the *normalized expected hit rate* for range $i$ as $Hit_i(m) \times F_i$, and the *normalized marginal gain* for range $i$ as $NMG_i(m) = MG_i(m) \times F_i$.

Although marginal gains are defined for a single level cache and measure hit rate rather than weighted I/O cost, normalized marginal gains induce an order of priority on all ranges – and thus on all blocks – in a trace. This order is used by our online management algorithm, Karma, to arrange the blocks in a multilevel cache system: the higher the range priority, the higher its blocks are placed in the cache hierarchy. This strategy maximizes the total normalized marginal gain of all blocks stored in the cache.

Note that when all blocks in a range have the same access frequency there is a correlation between the normalized marginal gain of a range and the probability that a block of this range will be accessed. A higher marginal gain indicates a higher probability. Therefore, there is no benefit in keeping blocks of ranges with low marginal gains in the cache: the probability that they will be accessed is small. Ranges with several access frequencies should be divided into smaller ranges according to those frequencies (see the example in Figure 2, described in Section 4).

A major advantage of basing caching decisions on marginal gains is the low level of detail required for their computation. Since only the general access pattern and access frequency are required, it is much more likely that an application be able to supply such information. Our experience shows that databases can supply this information with a high degree of accuracy. We expect our hints can also be derived from information available to the file system [16, 26].

## 4 Karma

Karma calculates the normalized marginal gain of a range of blocks (which corresponds to each of the sets described in Section 1) of blocks and then uses it to indicate the likelihood that the range's blocks will be accessed in the near future. To calculate the normalized marginal gain, Karma requires that all accessed disk blocks be classified into ranges. The following information must be provided (by means of application hints) for each range: an identifier for the range, its access pattern, the number of blocks in the range, and the frequency of access to this range. The computation is described in Section 3. Each block access is tagged with the block's range identifier, enabling all cache levels to handle the block according to its range.

Karma allocates for each range a fixed cache partition in a way that maximizes the normalized marginal gain of the blocks in all cache levels. It places ranges with higher normalized marginal gain in higher cache levels, where the access cost is lower. More precisely: space is allocated in $Cache_i$ for the ranges with the highest normalized marginal gain that were not allocated space in any $Cache_j$, $j < i$. For each level $i$ there can be at most one range which is split and is allocated space in both level $i$ and the adjacent lower level $i + 1$. Figure 2 shows an example of Karma's allocation.

Each range is managed separately, with the replacement policy best suited for its access pattern. When a block is brought into the cache, a block from the same range is discarded, according to the range's policy. This maintains the fixed allocation assigned to each range.

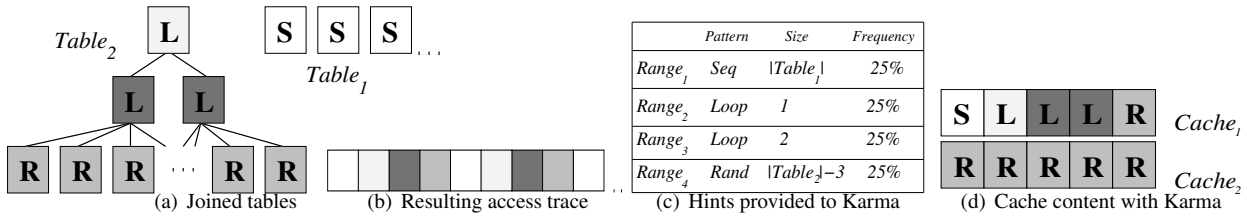
The amount of cache space required for maintaining the information about the ranges and the data structures for the cache partitions is less than one cache block. The pseudocode for Karma appears in Figure 3.

**Hints.** Karma strongly depends on the ability to propagate application information to the lower cache levels. Specifically, the range identifier attached to each block access is crucial for associating the block with the knowledge about its range. A method for passing information (sufficient for Karma) from the file system to the I/O system was suggested [8] and implemented in a Linux 2.4.2 kernel prototype.

For the two tables joined in the example in Figure 2, Karma will be supplied with the partitioning of the blocks into tables and index tree levels, as in Figure 2(c). Additionally, each cache level must know the aggregate size of all cache levels above it. Such information can be passed out-of-band, without changing current I/O interfaces. Each block access will be tagged with its range identifier, enabling all cache levels to classify it into the correct partition.

As in all informed management policies, Karma's performance depends on the quality of the hints. However, thanks to its exclusive caching, even with imperfect hints Karma will likely outperform basic LRU at each level. For example, with no hints provided and the entire data set managed as one partition with LRU replacement, Karma essentially behaves like Demote [49].

**Allocation.** Allocating cache space to blocks according to their normalized marginal gain would result in zero allocation for sequential accesses. Yet, in such patterns the application often accesses one block repeatedly before moving on to the next block. In some database queries, for example, a block may be accessed a few times, until all tuples in it have been processed. Therefore, ranges accessed sequentially are each allocated a




(a) Joined tables (b) Resulting access trace (c) Hints provided to Karma (d) Cache content with Karma

Figure 2: Karma's allocation of buffers to ranges in two cache levels. The application is a database query. (a) Two database tables are joined by scanning one of them sequentially and accessing the second one via an index. (b) The resulting access pattern is an interleaving of four ranges: one sequential (S), two loops (L), and one random (R). (c) This partitioning into ranges is supplied to Karma at the beginning of query execution. (d) Karma allocates one buffer for the sequential accesses (see text), three to hold the root and inner nodes of the index, and the remaining space to the randomly accessed blocks.

single block in the upper level cache (if prefetching is added to the model, several blocks may be allocated to allow for sequential read-ahead).

We allow for locality within the currently accessed block by always bringing it into $Cache_1$. When this block belongs to a range with no allocated space in $Cache_1$, we avoid unnecessary overhead by reading it using READ-SAVE, and discarding it without using DEMOTE.

**Lazy repartitioning.** When Karma is supplied with a new set of hints which indicate that the access patterns are about to change (e.g., when a new query is about to be processed), it repartitions all cache levels. Cache blocks in each level are assigned to the new partitions. As a result, disk blocks which do not belong to any of the new partitions in the cache level where they are stored become candidates for immediate eviction.

A block is rarely assigned a new range. For example, in a database blocks are divided according to their table or index level. Therefore, when a new query is processed the division into ranges will remain the same; only the access frequency and possibly the access pattern of the ranges will change. As a result, the blocks will remain in the same partition, and only the space allocated for this partition will change. This means that the sizes of the partitions will have to be adjusted, as described below.

Karma never discards blocks while the cache is not full, nor when the cache contains blocks which are immediate candidates for eviction (blocks from old ranges or blocks that were READ by an upper level). Karma ensures that even in transitional phases (between the time a new set of hints is supplied and the time when the cache content matches the new partitioning) the cache keeps the blocks with the highest marginal gain. As long as a cache (at any level) is not full, non-sequential ranges are allowed to exceed the size allocated for them. When no space is left and blocks from ranges with higher marginal gain are accessed, blocks from ranges which exceeded their allocated space are first candidates for eviction, in reverse order of their marginal gain.

Note that during repartitioning the cache blocks are not actually moved in the cache, as the partitioning is logical, not physical. The importance of handling this transitional stage is that it makes Karma less vulnerable to the order in which, for example, several queries are processed in a row.

**Replacement.** Karma achieves exclusive caching by partitioning the cache. This partitioning is maintained by use of DEMOTE and READ-SAVE, where each cache level stores only blocks belonging to its assigned ranges. For each $i$, $1 \leq i \leq n-1$, $Cache_i$ demotes all evicted blocks which do not belong to sequential ranges. When $Cache_i$ is about to read a block without storing it for future use, it uses READ-SAVE in order to prevent $Cache_{i+1}$ from discarding the block. Only one such block is duplicated between every two cache levels at any moment (see Figure 3). For each $j$, $2 \leq j \leq n$, $Cache_j$ does not store any READ blocks, but only those demoted by $Cache_{j-1}$ or read using READ-SAVE.

Special attention must be given to replacement in ranges which are split between adjacent cache levels $i$ and $i+1$. The LRU (or MRU) stack must be preserved across cache levels. $Cache_i$ manages the blocks in a split range with the corresponding policy, demoting all discarded blocks. $Cache_{i+1}$ inserts demoted blocks at the most recently used (MRU) position in the stack and removes them from the MRU or LRU position, according to the range's policy. Blocks READ by $Cache_i$ are removed from the stack and become immediate candidates for eviction. This way, the stack in $Cache_{i+1}$ acts as an extension of the stack in $Cache_i$.

## 5 Experimental Testbed

While Karma is designed to work in $n$ levels of cache, in our experiments we compare it to existing algorithms on a testbed consisting of two cache levels ($n = 2$).

### 5.1 PostgreSQL Database

We chose PostgreSQL [34] as a source of application hints. Each of its data files (table or index) is divided into disk blocks. Accesses to these blocks were traced

```
Partition (Block X)
  Return the partition to which X belongs

Transition (Block X)
  Return (cache is not full) or (LowestPriority(X) < Partition(X))

LowestPriority(Block X)
  Return partition with lowest priority exceeding its allocated size.
  If none exists return Partition(X).

Evict(Block X)
  If (X was in Reserved) or (Partition(X) is Seq)
    Discard X
  Else
    DEMOTE X
    Discard X

Insert (Block X, Partition P)
  If (cache is not full)
    Put X in P
  Else
    Remove block Z from LowestPriority(X)
  Evict (Z)
  Put X in P

Hit (Block X, Action A)
  If (first level) or (A = DEMOTE/READ-SAVE)
    update place in stack
  Else // low level and A = READ
    Put X in ReadBlocks
```

```
Miss (Block X, Action A)
  If (low level) and (A = READ)
    READ X
    Discard X
  Else // (first level) or (A = DEMOTE/READ-SAVE)
    If (Partition(X) fits in the cache) or (cache is not full)
      If (A ≠ DEMOTE)
        READ X
      Insert (X,Partition(X))
    Else // Partition(X) doesn't fit at all
      If (first level) // (A ≠ DEMOTE)
        READ-SAVE X
        Remove block Y from Reserved
        If (Transition(Y))
          Insert (Y, Partition(Y))
        Else
          Discard Y
        Put X in Reserved
      Else // low level
        If (Transition(X))
          If (A ≠ DEMOTE)
            READ X
          Insert (X,Partition(X))
        Else // no space for X
          If (A ≠ DEMOTE)
            READ-SAVE X
          Discard X
```

Figure 3: Pseudocode for Karma. ***Reserved***—A reserved buffer of size 1 in the first cache level for blocks belonging to ranges with low priority. ***ReadBlocks***—A low priority partition holding blocks that were READ by an upper level cache and are candidates for eviction. ***READ, READ-SAVE, DEMOTE***—The operations defined by the model in Section 2.

by adding trace lines to the existing debugging mechanism.

Like most database implementations, PostgreSQL includes an *explain* mechanism, revealing to the user the plan being executed for an SQL query. This execution plan determines the pattern and frequency with which each table or index file will be accessed during query execution. We used the output of explain to supply Karma with the characterization of the access pattern for each range, along with a division of all the blocks into ranges. Blocks are first divided by their table, and in some cases, a table can be sub-divided into several ranges. For example, in B-tree indices each level of the tree is characterized by different parameters (as in Figure 2).

## 5.2 TPC Benchmark H Traces

The TPC Benchmark H (TPC-H) is a decision support benchmark defined by the Transaction Processing Council. It exemplifies systems that examine large volumes of data and execute queries with a high degree of complexity [2]. In our experiments we used the default implementation provided in the benchmark specification to generate both the raw data and the queries.

We simulated our cache replacement policies on two types of traces:

- Repeated queries: each query is repeated several times, requesting different values in each run. This trace type models applications that access the database repeatedly for the same purpose, requesting different data in different iterations.

- Query sets: each query set is a permutation of the 22 queries of the benchmark, executed serially [2]. The permutations are defined in Appendix A of the benchmark specification. The query sets represent applications that collect various types of data from the database.

Queries 17, 19, 20, 21 and 22 have especially long execution traces (each over 25,000,000 I/Os). As these traces consist of dozens of loop repetitions, we used for our simulation only the first 2,000,000 I/Os of each trace.

## 5.3 Synthetic Zipf Workload

In traces with Zipf distribution, the frequency of access to block $i$ is proportional to $1/i^{\alpha}$, for $\alpha$ close to 1. Such distribution approximates common access patterns, such as file references in Web servers. Following previous studies [37, 49], we chose Zipf as a non-trivial random workload, where each block is accessed at a different, yet predictable frequency. We use settings similar to those used in previous work [49] and set $\alpha = 1$, for 25,000 different blocks.

Karma does not require information about the access frequency of each block. We supplied it with a parti-

| Alg | Basic-Alg | Double-Alg | Global-Alg |
|---|---|---|---|
| LRU | Basic-LRU (LRU+LRU) | Double-LRU | Demote |
| 2Q | Basic-2Q (LRU+2Q) | **Double-2Q** | **Global-2Q** |
| ARC | Basic-ARC (LRU+ARC) | **Double-ARC** | **Global-ARC** |
| MultiQ | Basic-MultiQ (LRU+MultiQ) | **Double-MultiQ** | **Global-MultiQ** |
| LRU-SP | Basic-LRU-SP (LRU-SP+LRU) | N/A | ***(SP-Karma)*** |

Table 1: The policies in our comparative framework. In addition to the policies known from the literature, we defined the Double and Global extensions indicated in bold. We compared Karma to all applicable Basic and Double policies, as well as to Demote and to Global-MultiQ.

tioning of the blocks into several ranges and the access frequency of each range. This frequency can be easily computed when the distribution parameters are available.

Although the blocks in each range have different access frequencies, for any two blocks $i$ and $j$, if $i < j$ then the frequency of access to block $i$ is greater then that of block $j$. The blocks are assigned to ranges in increasing order, and so for any two ranges $I$ and $J$, if $I < J$ then all the blocks in range $I$ have greater access frequencies than the blocks in range $J$.

## 5.4 Comparative framework

We compared Karma to five existing replacement algorithms: LRU, 2Q, ARC, MultiQ, and LRU-SP, each of which we examined using three different approaches: *Basic*, *Double*, and *Global*. In the Basic approach, each algorithm is used in conjunction with LRU, where the existing algorithm is used on one level and LRU on the other, as was defined in the respective literature. In the Double approach, each is used on both cache levels. The third approach is a global management policy, where each algorithm must be explicitly adapted to use DEMOTE.

Although Global-2Q and Global-ARC were not actually implemented in our experiments, we describe, in what follows, how they would work in a multilevel cache. It is also important to note that Global-MultiQ is not an existing policy: we defined this algorithm for the purpose of extending the discussion, and it is implemented here for the first time. We refer to the special case of LRU-SP in Section 5.5. The algorithms and approaches are summarized in Table 1. The actual experimental results are described in Section 6.

**Least Recently Used (LRU).** LRU is the most commonly used cache management policy. *Basic-LRU* and *Double-LRU* are equivalent, using LRU on both cache levels. *Global-LRU* is the Demote policy [49], where the upper level cache demotes all blocks it discards. The lower level cache puts blocks it has sent to the upper level at the head (closest to being discarded end) of its LRU queue, and puts demoted blocks at the tail.

**2Q.** 2Q [25] uses three queues. One LRU queue, $A_m$, holds "hot" pages that are likely to be re-referenced. A second FIFO queue, $A_{in}$, holds "cold" pages that are seen only once. The third LRU queue, $A_{out}$, is a ghost cache, holding meta-data of blocks recently evicted from the cache. As 2Q was originally designed for a second level cache, *Basic-2Q* uses LRU at the first cache level and 2Q at the second. *Double-2Q* uses 2Q at both cache levels. *Global-2Q* keeps $A_{out}$ and $A_{in}$ in the second level cache, dividing $A_m$ between both cache levels. In all these cases, we use the optimal parameters for each cache level [25]. $A_{in}$ holds 25% of the cache size and $A_{out}$ holds blocks that would fit in 50% of the cache.

**ARC.** ARC [33] was also designed for a second level cache. It divides blocks between two LRU queues, $L_1$ and $L_2$. $L_1$ holds blocks requested exactly once. $L_2$ holds blocks requested more than once. The bottom (LRU) part of each queue is a ghost cache. The percentage of cache space allocated to each queue is dynamically adjusted, and history is saved for as many blocks that would fit in twice the cache size. *Basic-ARC* uses LRU at the first cache level and ARC at the second. *Double-ARC* uses ARC at both cache levels. *Global-ARC* keeps the ghost caches in the second level cache, as well as the LRU part of what is left of $L_1$ and $L_2$. ARC is implemented for each cache level with dynamic adjustment of the queue sizes [33].

**MultiQ.** MultiQ [51] was originally designed for a second level cache. It uses multiple LRU queues, each having a longer lifetime than the previous one. When a block in a queue is accessed frequently, it is promoted to the next higher queue. On a cache miss, the head of the lowest non-empty queue is evicted. *Basic-MultiQ* uses LRU at the first cache level and MultiQ at the second. *Double-MultiQ* uses MultiQ at both cache levels. We implemented MultiQ for each cache level with 8 queues and a ghost cache. The *Lifetime* parameter is set according to the observed temporal distance. We extended MultiQ to *Global-MultiQ* in a straightforward way. The ghost cache is allocated in the second level cache, and the queues are divided dynamically between the cache levels, with at most one queue split between the levels. Whenever a block is brought into the first level cache, the block at the bottom of the lowest queue in this level is demoted to the second level cache.

## 5.5 Application Controlled File Caching

In **LRU-SP** [10], applications may use specific interface functions to assign priorities to files (or ranges in files). They may also specify cache replacement policies for each priority level. Blocks with the lowest priority are

first candidates for eviction. In the original paper, applications were modified to include calls to these interface functions.

As a policy assuming hints from the application, LRU-SP is designed for a first level cache. Therefore, we implement *Basic-LRU-SP* with LRU at the second level. *Double-LRU-SP* would let the application manage each cache level directly yet independently. This seems unreasonable and thus we did not implement this extension. Without hints available at the second level cache, the simple addition of DEMOTE will not result in global management, since informed decisions cannot be made in the second level. Thus, extending LRU-SP to *Global-LRU-SP* is not applicable.

**A new multilevel extension.** To evaluate the contribution of the different mechanisms of Karma, we defined a new policy, *SP-Karma*, for managing multilevel caches, which added to LRU-SP most of the features we defined in Karma. This extension resulted in a new cache management algorithm which is similar to Karma and allows us to better understand the value of specific attributes of Karma. In particular, we added DEMOTE for cooperation between cache levels, we derived priorities using Karma's calculation of normalized marginal gains (this mechanism was also used to supply hints to Basic-LRU-SP above), and we supplied these priorities to both cache levels. Since SP-Karma now bases its decisions on Karma's priorities, the significant difference between our two policies is the use of READ-SAVE.

The results of SP-Karma resembled those of Karma, but Karma achieved better performance on all traces. The use of READ-SAVE resulted in Karma executing fewer DEMOTE operations, thus outperforming SP-Karma by up to 1% in the large caches and 5% in the small ones. Since we defined our new policy, SP-Karma, to be very similar to Karma and since it shows similar results, in Section 6 we compare only Karma to the existing policies.

## 6 Results

We simulated the policies described in Section 5.4 on a series of increasing cache sizes and measured the weighted I/O cost:

$$\begin{aligned} \textit{Weighted I/O cost} = {} & C_2 \times (\text{misses in } Cache_1) \\ & + D_2 \times (\text{DEMOTEs to } Cache_2) \\ & + C_{Disk} \times (\text{misses in } Cache_2) \end{aligned}$$

For all experiments (excluding the one depicted in Figure 9), we set $C_2 = D_2 = 1$ and $C_{Disk} = 20$, as in [49]. For all experiments, excluding the one depicted in Figure 10, we assume, as in previous studies [11, 49], that the caches are of equal size ($S_1 = S_2$). The cache size in the graphs refers to the aggregate cache size as a fraction of the size of the data set. Thus, a cache size of 1 means that each cache level is big enough to hold one-half of the data set. A cache size of 2 means that the entire data set fits in the top level, and all policies should perform equally.

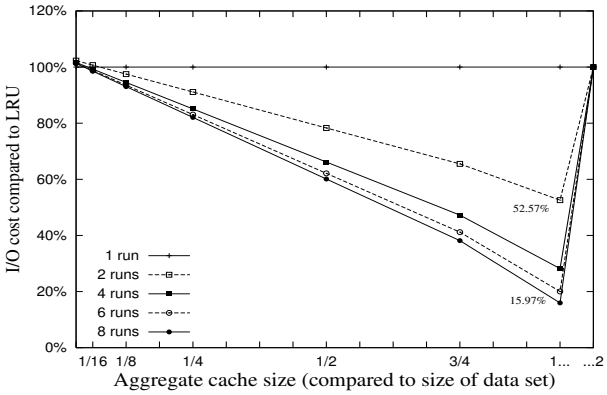



Figure 4: Karma's improvement over LRU on Query 3 run multiple times.

The results are normalized to their weighted I/O cost compared to that incurred by LRU. This gives a better representation of the improvement over the default LRU management, making it easier to compare the policies.

**How does Karma compare to Basic-LRU?** We ran all policies on traces of each query, using several cache sizes. Karma generally yielded similar curves for all queries. In Figure 4 we take a close look at Query 3 as a representative example, in order to understand the behavior of Karma with different cache sizes. Query 3 is sequential, scanning 3 tables. PostgreSQL creates a hash table for each database table and joins them in memory. Subsequent runs of Query 3 result in a looping reference.

We ran Query 3 eight times, to show how the I/O cost incurred by Karma decreases, in comparison to that of LRU, as the number of runs increases. LRU suffers from the three problems of multilevel caching. The second level cache experiences no locality of reference, as all repeated accesses are hidden by the first level. Even when the aggregate cache is as large as the data set LRU does not exploit the entire space available due to redundancy between levels. Finally, when the cache is smaller than the data set LRU is unable to maintain used blocks until the next time they are accessed, and so it does not benefit from increasing the cache until its size reaches the size of the data set. Karma does not suffer any of those drawbacks, and its weighted I/O cost decreases significantly as the cache size increases. Although the portion of the loop which fits in the cache is equal for all runs (for each cache size), the hit rate of LRU remains zero, while Karma's hit rate increases in both cache lev-

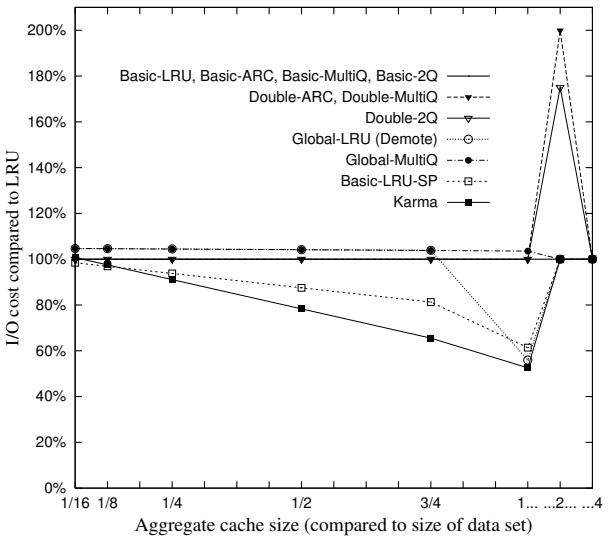



Figure 5: Weighted I/O cost for all policies on Query 3 run twice. Policies with identical results are plotted using a single line. Other than Karma, only the Global policies are able to fit the entire data set in the cache when the aggregate cache size equals the size of the data set. Unlike the non-informed policies, Karma and Basic-LRU-SP reduce their I/O cost gradually as the cache size increases.

els. This causes an increasing improvement in Karma's weighted I/O cost compared to that of LRU.

**How does Karma compare to single level non-informed policies?** Figure 5 contains the results for all policies on Query 3. The Basic implementations of 2Q, MultiQ, and ARC behave like LRU. Their division into separate LRU queues does not help in identifying the looping reference if it is larger than the cache size. When the cache size increases to the point where the entire loop fits in the cache, there can be no improvement over LRU.

The Double extensions of 2Q, MultiQ, and ARC allocate a ghost cache (whose size is 0.22%, 0.16% and 0.43% of the cache size, respectively) in both cache levels. This small portion is sufficient to prevent the data set from fitting in one level when the cache size is 2, leading to worse performance than LRU.

Karma is informed of the looping reference and manages it with MRU replacement. This enables Karma to benefit from every increase in the cache size. When the entire loop fits in the aggregate cache Karma benefits from its exclusive caching and shows the largest improvement over the I/O cost of LRU. We refer to the other global and informed policies later in this section.

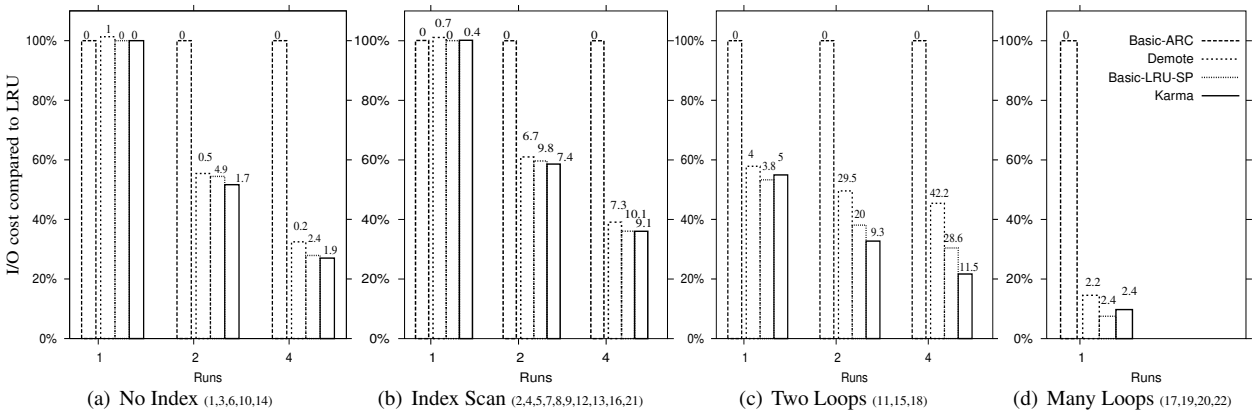
The results for the other queries are similar to those for Query 3. Figure 6 summarizes the results for multiple runs of each query, comparing the weighted I/O cost of Karma to three representative policies. Each query was executed four times, with different (randomly generated) parameters. The traces are a concatenation of one to four of those runs, and the results are for an aggregate cache size which can contain the entire data set. Most of the queries were primarily sequential, causing all policies to perform like LRU for a single run. Repeated runs, however, convert the sequential access into a looping access, which is handled poorly by LRU when the loop is larger than the cache. These experiments demonstrate the advantage of policies which are not based purely on recency.

The queries were of four basic types, and the results are averaged for each query type. The first type includes queries which consist only of sequential table scans. Karma's I/O cost was lower than that of LRU and LRU-based policies on these queries by an average of 73% on four runs (Figure 6(a)). In the second type, an index is used for a table scan. Indices in PostgreSQL are constructed as B-trees. An index scan results in some locality of reference, improving LRU's performance on query traces of this type, compared to its performance on queries with no index scans. Karma's I/O cost on these queries was lower than that of LRU by an average of 64% on four runs (Figure 6(b)). In the third query type, one or more tables are scanned several times, resulting in a looping pattern within a single execution of the query. In queries of this type, I/O cost lower than that of LRU is obtained by some policies even for one run of the query. Four runs of each of these queries consisted of eight runs of the loop, resulting in significant reduction in I/O cost as compared to the other query types. Karma's I/O cost on these queries was lower than that of LRU by an average of 78.33% on four runs (Figure 6(c)). In the last type, each query instance included dozens of iterations over a single table. These traces were trimmed to the first 2,000,000 I/Os, which still contained more than ten loop iterations. On a single run of these queries Karma performed better than LRU by an average of 90.25% (Figure 6(d)).

The results in Figure 6 for Basic-ARC correspond to those in Figure 5. Like the rest of the LRU-based policies constructed for a single cache level, it is not exclusive, and thus it is unable to exploit the increase in cache size and fit the data set in the aggregate cache.

To see how the different policies perform on more heterogeneous traces, we use the query sets described in Section 5.2, which are permutations of all queries in the benchmark. Our results for the first 20 sets are shown in Figure 7 for the largest cache sizes, where the policies showed the most improvement. The left- and right-hand columns show results for an aggregate cache size that can hold half of the data set or all of it, respectively.

The I/O cost of Basic-ARC and Double-ARC is not much lower than that of LRU. ARC is designed to handle traces where most of the accesses are random or exhibit locality of reference. Its dynamic adjustment is aimed at




(a) No Index (1,3,6,10,14) (b) Index Scan (2,4,5,7,8,9,12,13,16,21) (c) Two Loops (11,15,18) (d) Many Loops (17,19,20,22)

Figure 6: Improvement in I/O cost of Karma, Basic-ARC, Demote and Basic-LRU-SP compared to LRU, on repeated runs of all queries. The columns show the average I/O cost of these policies for each query type, when the aggregate cache size equals the size of the data set. Each column is tagged with the standard deviation for this query type. Karma improved over the I/O cost of LRU on repeated runs of the queries by an average of 73%, 64%, 78% and 90% on query types a, b, c, and d, respectively.

preventing looping and sequential references from polluting the cache. It is not designed to handle traces where larger segments of the trace are looping or sequential.

The I/O cost of Basic-2Q and Basic-MultiQ is lower than that of LRU for the larger aggregate cache. Both policies benefit from dividing blocks into multiple queues according to access frequency. Double-2Q outperforms Basic-2Q by extending this division to both cache levels. Double-MultiQ, however, suffers when the ghost cache is increased in two cache levels and does not show average improvement over Basic-MultiQ. The high standard deviation of the Basic and Double extensions of 2Q and MultiQ demonstrate their sensitivity to the order of the queries in a query set, and consequently, their poor handling of transitional stages.

Karma outperforms all these policies. Its reduction in I/O cost is significantly better than that of the non-informed single level policies. This reduction is evident not only when the entire data set fits in the cache, but in smaller cache sizes as well. The low standard deviation shows that it copes well with changing access patterns resulting in transitional stages.

Figure 8 shows how Karma compares to existing policies on a Zipf workload. We chose to present the results for Double-ARC because it showed better performance than all non-informed policies that were designed for a single cache level. This is because ARC avoids caching of blocks that are accessed only once in a short period of time. In a Zipf workload, these are also the blocks which are least likely to be accessed again in the near future. Note that ARC's improvement is most evident when the cache is small. When the cache is larger, such blocks occupy only a small portion of the cache in LRU, and so the benefit of ARC is less distinct.

Karma improves over the I/O cost of LRU by as much as 41%, adding as much as 25% to the improvement of

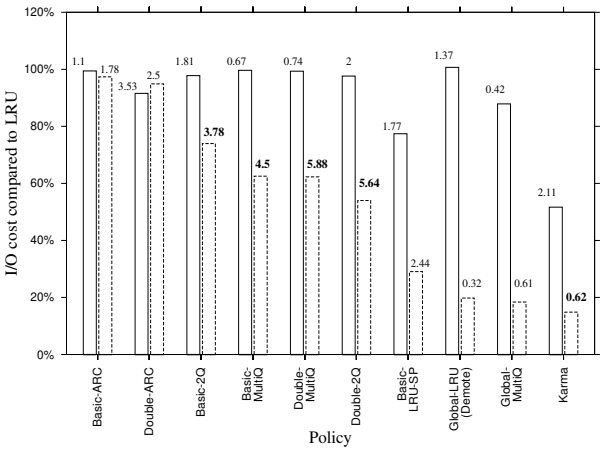



Figure 7: Weighted I/O cost of Karma and the existing policies on 20 query sets, as compared to LRU. The columns show the weighted I/O cost for each policy averaged over 20 query sets for each of two cache sizes. The left bar for each policy is for an aggregate cache size of 1/2 and the right one is for 1. Each column is tagged with the standard deviation for this policy.The policies are sorted in descending order of their I/O cost for the large cache size. Karma shows improvement over all cache sizes by combining knowledge of the access pattern with exclusive caching. It improves the I/O cost of LRU by 47% and 85% for the small and big cache sizes, respectively.

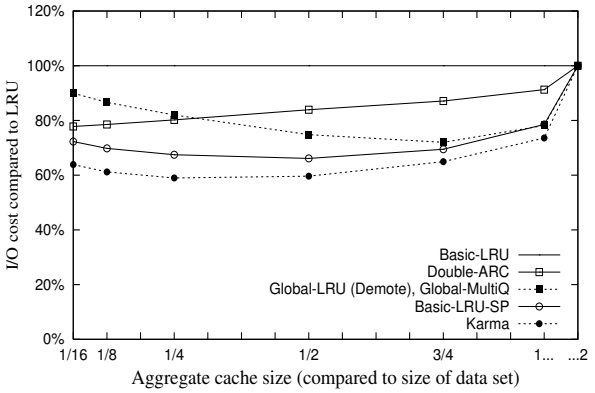



Figure 8: Weighted I/O cost for selected policies on a Zipf workload. Even when the access pattern exhibits strong temporal locality, Karma outperforms all LRU-based policies and the basic hint-based policy. Karma improves over the I/O cost of LRU by at least 26%, and by as much as 41%.

Double-ARC. Since the access frequency of the blocks is available to Karma, it does not bring into the cache blocks with low frequency. Exclusiveness further increases its effective cache size, resulting in improved performance.

**How does Karma compare to global non-informed policies?** When run on Query 3, Demote (Global-LRU) (Figure 5) exhibits different behavior than the single level policies. It manages to reduce the I/O cost significantly when the entire data set fits in the aggregate cache. This is the result of the exclusive caching achieved by using DEMOTE. Still, for smaller cache sizes, it is not able to "catch" the looping reference and does not benefit from an increase in cache size. Instead, its overhead from futile DEMOTE operations means that its performance is worse than LRU's for all cache sizes in which the aggregate cache cannot contain all the blocks. We expected Global-MultiQ to perform at least as well as Demote, but due to its large ghost cache it is unable to fit the entire data set into the aggregate cache, even when the cache is large enough. Instead, it only suffers the overhead of futile DEMOTE operations. Being a global policy, Karma is able to exploit the entire aggregate cache. Since it manages the looping accesses with MRU replacement, it improves gradually as the cache size increases.

Figure 6 shows only the results for the largest cache size, where Demote shows its best improvement. In fact, we expect any global policy to achieve this improvement when the entire data set fits in the aggregate cache. Even there, Karma achieves lower I/O cost than Demote, thanks to its use of READ-SAVE and MRU management for loops. Unlike Demote, Karma achieves this reduction in smaller cache sizes as well.

The performance of the global policies on the query sets is represented well in Figure 7. It is clear that when the cache size is smaller than the data set, they are not able to improve the I/O cost of LRU significantly. This improvement is only achieved when the entire data set fits in the aggregate cache (represented by the right-hand column). Karma, however, improves gradually as the cache size increases. Combining exclusive management with application hints enables it to maximize the cache performance in all cache sizes.

When the access pattern does not include looping references (Figure 8), the global policies improve gradually as the cache size increases. Although a Zipf workload exhibits significant locality of reference, adding exclusiveness to LRU does not achieve good enough results. In the smallest cache size Demote and Global-MultiQ improve over the I/O cost of LRU by 10%. Karma, with its knowledge of access frequencies, achieves additional improvement of 26%.

**How does Karma compare to hint-based policies?** Basic-LRU-SP using Karma's hints and priorities performs better than any non-informed single level policy, for all our traces. Using hints enables it to optimize its use of the upper level cache. When the access pattern is looping, the combination of MRU management in the upper cache and default LRU in the lower results in exclusive caching without the use of DEMOTE. Note the surprising effect on the queries with many loops (Figure 6(d)), where Basic-LRU-SP outperforms Karma when the entire data set fits in the aggregate cache. Karma pays for the use of DEMOTE, while Basic-LRU-SP enjoys "free" exclusive caching, along with the accurate priorities generated by Karma. The average I/O cost of Basic-LRU-SP is 92.5% lower than that of LRU. Karma's average I/O cost is 90.25% lower than that of LRU. When the aggregate cache is smaller than the data set, or when the pattern is not a pure loop (Figures 5, 6(a-c), 7, and 8), Karma makes optimal use of both caches and outperforms Basic-LRU-SP significantly.

**How is Karma affected by the model parameters?** We wanted to estimate Karma's sensitivity to varying access costs in different storage levels. Figure 9 shows how Karma behaves on query set 1 (the behavior is similar for all traces) when the disk access delay ranges from 10 to 100 times the delay of a READ from the second level cache (or a DEMOTE to that cache). When the delay for a disk access is larger, the "penalty" for DEMOTE is less significant compared to the decrease in the number of disk accesses. When DEMOTE is only ten times faster than a disk access, its added cost outweighs its benefits in very small caches.

**How is Karma affected by the cache size at each level?** Figure 10 compares the behavior of Karma to other policies on cache hierarchies with varying lower-level sizes. The details are numerous and so we present here only results for the best LRU-based policies, Double-2Q and Global-Multi-Q. Global-Multi-Q

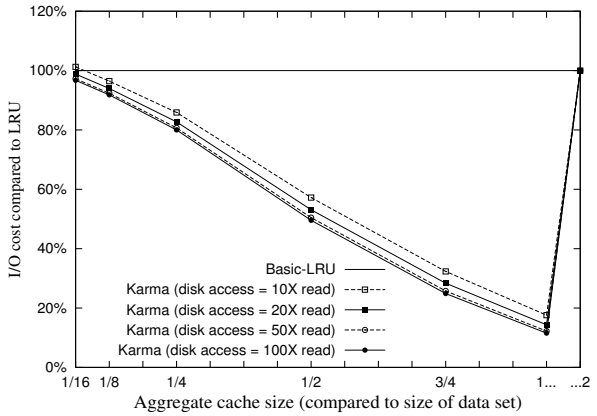



Figure 9: Karma's I/O cost compared to that of LRU for different disk access delays, for query set 1. When delays for a disk access are longer, there is more benefit in using DEMOTE, despite its additional cost.

performs futile DEMOTEs both when the cache sizes are small and when the lower cache is larger than the upper cache. In the first case, demoted blocks are discarded before being used, and in the second they are already present in the lower cache. As a result of data redundancy between caches, Double-2Q is highly sensitive to the portion of the aggregate cache that is in the upper level. Karma does not suffer from those problems and outperforms both policies (and those omitted from the graphs) in all cache settings.

We expect Karma to maintain its advantage when the difference in cache sizes increases. In the Basic and Double policies the benefit of the smaller cache will become negligible due to data redundancy, whereas in the Global policies the amount of futile Demote operations will increase. Karma, on the other hand, benefits from any additional cache space, in any level, and maintains exclusive caching by using only the necessary amount of DEMOTEs.

## 7 Related Work

We discussed several existing cache replacement policies in Section 5. Here we elaborate on additional aspects of related work.

**Multilevel.** Wong and Wilkes [49] introduced the DEMOTE operation to prevent inclusion between cache levels. They assume that network connections are much faster than disks. In such settings, performance gains are still possible even though a DEMOTE may incur a network cost. Cases where network bandwidth is the bottleneck instead of disk contention were addressed in a later study [51]. There, instead of evicted blocks being demoted from the first to the second level cache, they are reloaded (prefetched) into the second level cache from the disk. A complementary approach has the application attach a "write hint" [29] to each WRITE command, choosing one of a few reasons for performing the write. The storage cache uses these hints to "guess" which blocks were evicted from the cache and which are cached for further use. In X-Ray [45] the information on the content of the upper level cache is obtained using gray-box techniques, and derived from operations such as file-node and write log updates.

In ULC [24], the client cache is responsible for the content of all cache levels underneath. The level of cache in which blocks should be kept is decided by their expected locality of reference. A "level tag" is attached to each I/O operation, stating in which cache level the block should be stored. In *heterogeneous caching* [4], some degree of exclusivity can be achieved without any cooperation between the levels, when each cache level is managed by a different policy. A multilevel cache hierarchy is presented, where each cache is managed by an adaptive policy [20], ACME (Adaptive Caching using Multiple Experts): the "master" policy monitors the miss rate of a pool of standard replacement policies and uses machine learning algorithms to dynamically shift between those policies, in accordance with changing workloads.

Karma uses READ-SAVE to avoid unnecessary DEMOTEs. As with the average read access time in ULC [24], our storage model enables specific calculations of the DEMOTE operations that actually occurred, instead of estimating them in the cost of each READ, as in the original evaluation [49]. The use of READ-SAVE reflects a non-centralized operation, as opposed to the level tags. Our results in Section 6 (Figures 5, 7, and 8) show that exclusive caching is not enough to guarantee low I/O cost. Karma is able to make informed replacement decisions to achieve better results.

**Detection-based caching.** Detection based policies use history information for each block in the cache (and sometimes for blocks already evicted from the cache) to try to "guess" the access pattern of the application. This may help identify the best candidate block for eviction. DEAR [13], AFC [14], and UBM [27], which are designed for a file system cache, all collect data about the file the block belongs to or the application requesting it, and derive access patterns (sequential, looping, etc.).

In PCC [19], the I/O access patterns are correlated with the program counter of the call instruction that triggers the I/O requests, enabling differentiation between multiple patterns in the same file if they are invoked by different instructions. Each pattern is allocated a partition in the cache, whose size is adjusted dynamically until the characterization of patterns stabilizes.

MRC-MM [50] monitors accesses to virtual memory pages in order to track the page miss ratio curve (MRC) of applications. The MRC in each "epoch" is then used

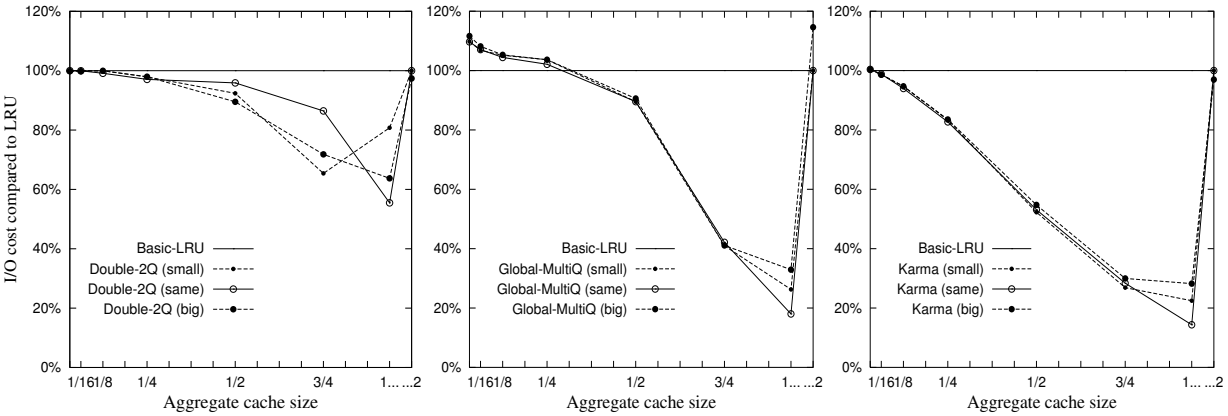



Figure 10: Weighted I/O cost on query set 1 (the behavior is similar for all traces) when cache levels are of different sizes. For lack of space we show results only for Karma and for the best LRU-based policies. *Same:* $|Cache_2| = |Cache_1|$. *Big:* $|Cache_2| = |Cache_1| \times 2$. *Small:* $|Cache_2| = |Cache_1|/2$. Double-2Q is highly sensitive to the difference in size and Global-MultiQ suffers from overhead of DEMOTE, while Karma outperforms all policies for all cache settings.

to calculate the marginal gain of all processes. Memory utilization is maximized by allocating larger memory portions to processes with higher marginal gain.

SARC [17], designed for a second level cache, detects sequential streams based on access to contiguous disk tracks. This information is used for sequential prefetching and cache management. The cache is dynamically partitioned between random and sequential blocks.

Since Karma is provided with the characterization of block ranges, it does not incur the additional overhead of gathering and processing access statistics. The access pattern and access frequency for each range are known and on-the-fly adjustment of partition sizes is avoided. Karma adjusts its partitions only in transitional phases, when the application changes its access pattern.

**Informed caching.** A different approach to determining access patterns is to rely on application hints that are passed to the cache management mechanism. This eliminates the need for detection, thus reducing the complexity of the policy. However, relying on hints admittedly limits applicability. Existing hint based policies require the applications to be explicitly altered to manage the caching of their own blocks. LRU-SP [10] and TIP2 [38] are two such policies.

In TIP2, applications disclose information about their future requests via an explicit access string submitted when opening a file. The cache management scheme balances caching and prefetching by computing the value of each block to its requesting application.

Unlike TIP2, Karma does not require an explicit access string, but a general characterization of the observed access patterns (i.e., looping, random or sequential). In this way, it is similar to LRU-SP. This makes Karma useful for applications such as databases, where accesses can be characterized in advance into patterns.

The ability of databases to disclose information about future accesses has made them ideal candidates for hint generation. Database query optimizers [44] choose the optimal execution path for a query. They aim to minimize the execution cost, which is a weighted measure of I/O (pages fetched) and CPU utilization (instructions executed). Once the optimal path is chosen, the pattern of access to the relevant blocks is implicitly determined. It can then be easily disclosed to the cache manager. A fundamental observation [47] was that in order for an operating system to provide buffer management for database systems, some means must be found to allow it to accept "advice" from an application program concerning the replacement strategy. The following studies base their replacement policy on this method.

A *hot set* is a set of pages over which there is a looping behavior [41]. Its size can be derived from the query plan generated by the optimizer. In the derived replacement policy [41], a separate LRU queue is maintained for each process, with a maximal size equal to its hot set size. DBMIN [15] enhances the hot set model in two ways. A hot set is defined for a file, not for an entire query. Each hot set is separately managed by a policy selected according to the intended use of the file.

By adding marginal gains to this model, MG-x-y [36] is able to compare how much each reference string will "benefit" from extra cache blocks. The marginal gain of random ranges is always positive, and so MG-x-y avoids allocating the entire cache to only a small number of such ranges by imposing a maximum allocation of $y$ blocks to each of the random ranges.

Karma builds upon the above policies, making a fine-grained distinction between ranges. A file may contain several ranges accessed with different characteristics. As in DBMIN and MG-x-y, each range is managed with a policy suited for its access pattern. Like MG-x-y, Karma uses marginal gains for allocation decisions, but instead

of limiting the space allocated to each range it brings every accessed block into $Cache_1$ to capture fine-grain locality. Most importantly, unlike the above policies, Karma maintains all its benefits over multiple cache levels.

A recent study [11] evaluates the benefit of *aggressive collaboration*, i.e., use of DEMOTE, hints, or level tags, over *hierarchy-aware* caching, which does not require modifications of current storage interfaces. Their results show that the combination of hints with global management yields only a slight advantage over the hierarchy aware policies. However, they experiment with very basic hints, combined with LRU or ARC management, while it is clear from our results that simply managing loops with MRU replacement is enough to achieve much better results. Since Karma distinguishes between access patterns and manages each partition with the policy best suited for it, its improvement is significant enough to justify the use of hints.

**Storage system design.** Modern storage systems are designed as standalone platforms, separated from their users and applications by strict protocols. This modularity allows for complex system layouts, combining hardware from a range of manufacturers. However, the separation between the storage and the application layers precludes interlayer information sharing that is crucial for cooperation between the systems components - cooperation we believe will lead to substantial performance gains.

Many recent studies attempt to bypass this inherent separation: the levels gather knowledge about each other by tracking the implicit information exposed by current protocols. For example, in several recent studies [32, 43] the operating system extracts information about the underlying disk queues and physical layout from disk access times. In the gray-box approach [5, 46], the storage system is aware of some operating system structures (such as inodes), and extracts information about the current state of the file system from operations performed on them. In C-Miner [30], no knowledge is assumed at the storage level. The storage system uses data mining techniques in order to identify correlations between blocks. The correlations are then used for improving prefetching and data layout decisions.

In contrast, other studies explore the possibility of modifying traditional storage design. Some offer specific protocol additions, such as the DEMOTE operation [49]. Others suggest a new level of abstraction, such as object storage [6]. Other work focused on introducing new storage management architectures aimed at optimizing database performance [21, 42].

Karma requires some modification to existing storage interfaces, which is not as substantial as that described above [6, 21, 42]. This modification will enable the storage system to exploit the information available in other levels, without paying the overhead of extracting and deriving this information. We believe that the additional complexity is worthwhile, given the large performance benefits presented in Section 6.

| Policy | 1st level | 2nd level | Extra Information | Exclusive |
|---|---|---|---|---|
| LRU | ✓ | ✓ | X | X |
| DEAR,UBM AFC,PCC | ✓ | X | file,application PC | X |
| LRU-SP | ✓ | X | pattern,priority | X |
| TIP2 | ✓ | X | explicit trace | X |
| HotSet,DBMIN MG-x-y | ✓ | X | query plan | X |
| X-Ray Write hints | X | ✓ | indirect hints | ✓ |
| 2Q,MultiQ ARC,CAR,SARC | X | ✓ | X | X |
| Demote Global-L2 | X | ✓ | X | ✓ |
| ULC | ✓ | ✓ | client instructions | ✓ |
| ACME | ✓ | ✓ | (machine learning) | ✓ |
| Karma | ✓ | ✓ | ranges | ✓ |

Table 2: Summary of cache management policies, the extra information they require, and whether or not they are suitable for use in first and second level caches.

Table 2 summarizes the policies discussed in this paper. The policies are compared according to their ability to perform well in more than one cache level, to achieve exclusiveness in a multilevel cache system, and to use application hints.

## 8 Conclusions and Future Work

We defined a model for multilevel caching and defined the weighted I/O cost of the system for this model as the sum of costs of all operations performed on a trace. We propose a policy which solves the three problems that can occur in a multilevel cache: blurring of locality of reference in lower level caches, data redundancy, and lack of informed caching at the lower cache levels. None of the existing policies address all these problems.

Our proposed policy, Karma, approximates the behavior of the optimal offline algorithm, MIN. Like MIN, it aims to optimize the cache content by relying on knowledge about the future, instead of on information gathered in the past. Karma uses application hints to partition the cache and to manage each range of blocks with the policy best suited for its access pattern. It saves in the cache the blocks with the highest marginal gain and achieves exclusive caching by partitioning the cache using DEMOTE and READ-SAVE. Karma improves the weighted I/O cost of the system significantly. For example, on a permutation of TPC-H queries, Karma improves over pure LRU by an average of 85%. It adds an average

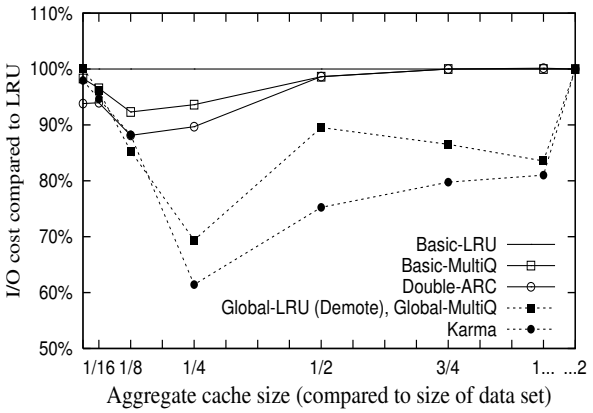



Figure 11: Preliminary results for an OLTP workload. Using TPCC-UVa [31], an open-source implementation of the TPC-C [3] benchmark, we created a trace of over 10,000,000 I/Os accessing 130,000 distinct blocks. The hints for Karma were generated using the "explain" output for each transaction's queries, and the frequency of each transaction. We present only the best LRU based policies. Karma's improvement over LRU is greater than the improvement of all of these policies, in all cache sizes but one, where it equals the improvement of Global-MultiQ.

of 50% to the improvement of Demote over LRU and an average of 25% to that of LRU-SP.

When more features are added to Karma, we believe it will be able to achieve such improvement on workloads that are essentially different from decision support. We intend to add calculations of marginal gain for random ranges which are not necessarily of uniform distribution. Karma will also handle ranges which are accessed concurrently with more than one access pattern. Figure 11 shows our initial experiments with an OLTP workload, demonstrating that even without such additions, Karma outperforms existing LRU-based algorithms on such a trace by as much as 38%.

The framework provided by Karma can be extended to deal with further additions to our storage model. Such additions may include running multiple concurrent queries on the same host, multiple caches in the first level, or prefetching. DEMOTE and READ-SAVE can still be used to achieve exclusiveness, and the marginal gains will have to be normalized according to the new parameters. Karma relies on general hints and does not require the application to submit explicit access strings or priorities. Thus, we expect its advantages will be applicable in the future not only to databases but to a wider range of applications.

## Acknowledgments

We thank our shepherd Scott Brandt and the anonymous reviewers for their helpful comments. We also thank Kai Li and Avi Yadgar for fruitful discussions.

## References

[1] Working draft SCSI block commands - 2 (SBC-2), 2004.

[2] TPC benchmark H standard specification, Revision 2.1.0.

[3] TPC benchmark C standard specification, Revision 5.6.

[4] Ismail Ari. *Design and Management of Globally Distributed Network Caches*. PhD thesis, University of California Santa Cruz, 2004.

[5] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and control in gray-box systems. In *Symposium on Operating Systems Principles (SOSP)*, 2001.

[6] Alain Azagury, Vladimir Dreizin, Michael Factor, Ealan Henis, Dalit Naor, Noam Rinetzky, Ohad Rodeh, Julian Satran, Ami Tavory, and Lena Yerushalmi. Towards an object store. In *NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, 2003.

[7] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with adaptive replacement. In *USENIX Conference on File and Storage Technologies (FAST)*, 2004.

[8] Tsipora Barzilai and Gala Golan. Accessing application identification information in the storage tier. Disclosure IL8-2002-0055, IBM Haifa Labs, 2002.

[9] L.A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[10] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, 1996.

[11] Zhifeng Chen, Yan Zhang, Yuanyuan Zhou, Heidi Scott, and Berni Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *SIGMETRICS*, 2005.

[12] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction-based placement for storage caches. In *USENIX Annual Technical Conference*, 2003.

[13] Jongmoo Choi, Sam H. Noh, Sang Lyul Min, and Yookun Cho. An implementation study of a detection-based adaptive block replacement scheme. In *USENIX Annual Technical Conference*, 1999.

[14] Jongmoo Choi, Sam H. Noh, Sang Lyul Min, and Yookun Cho. Towards application/file-level characterization of block references: a case for fine-grained buffer management. In *SIGMETRICS*, 2000.

[15] Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *International Conference on Very Large Data Bases (VLDB)*, 1985.

[16] Gregory R. Ganger, Daniel Ellard, and Margo I. Seltzer. File classification in self-$*$ storage systems. In *International Conference on Autonomic Computing (ICAC)*, 2004.

[17] Binny S. Gill and Dharmendra S. Modha. SARC: Sequential prefetching in adaptive replacement cache. In *USENIX Annual Technical Conference*, 2005.

[18] Binny S. Gill and Dharmendra S. Modha. WOW: Wise ordering for writes-combining spatial and temporal locality in non-volatile caches. In *USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[19] Chris Gniady, Ali R. Butt, and Y. Charlie Hu. Program counter based pattern classification in buffer caching. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[20] Robert B. Gramacy, Manfred K. Warmuth, Scott A. Brandt, and Ismail Ari. Adaptive caching by refetching. In *Neural Information Processing Systems (NIPS)*, 2002.

[21] Stavros Harizopoulos and Anastassia Ailamaki. A case for staged database systems. In *Conference on Innovative Data Systems Research (CIDR)*, 2003.

[22] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. Dulo: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[23] Song Jiang and Xiaodong Zhang. LIRS: An efficient low interreference recency set replacement policy to improve buffer cache performance. In *SIGMETRICS*, 2002.

[24] Song Jiang and Xiaodong Zhang. ULC: A file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. In *International Conference on Distributed Computing Systems (ICDCS)*, 2004.

[25] Theodore Johnson and Dennis Shasha. 2Q: a low overhead high performance buffer management replacement algorithm. In *International Conference on Very Large Data Bases (VLDB)*, 1994.

[26] John F. Karpovich, Andrew S. Grimshaw, and James C. French. Extensible file system (ELFS): an object-oriented approach to high performance file I/O. In *OOPSLA*, 1994.

[27] Jong Min Kim, Jongmoo Choi, Jesung Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[28] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong-Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *SIGMETRICS*, 1999.

[29] Xuhui Li, Ashraf Aboulnaga, Kenneth Salem, Aamer Sachedina, and Shaobo Gao. Second-tier cache management using write hints. In *USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[30] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, and Yuanyuan Zhou. C-miner: Mining block correlations in storage systems. In *USENIX Conference on File and Storage Technologies (FAST)*, 2004.

[31] Diego R. Llanos and Belén Palop. An open-source TPC-C implementation for parallel and distributed systems. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[32] Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger. Freeblock scheduling outside of disk firmware. In *USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[33] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[34] Bruce Momjian. *PostgreSQL: Introduction and Concepts*. Addison-Wesley, 2000.

[35] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems - or - your cache ain't nuthin' but trash. In *USENIX Winter Conference*, 1992.

[36] Raymond Ng, Christos Faloutsos, and Timos Sellis. Flexible buffer allocation based on marginal gains. In *ACM SIGMOD International Conference on Management of Data*, 1991.

[37] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *ACM SIGMOD International Conference on Management of Data*, 1993.

[38] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 224–244. IEEE Computer Society Press and Wiley, NY, 2001.

[39] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *SIGMETRICS*, 1990.

[40] Giovanni Maria Sacco and Mario Schkolnick. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *International Conference on Very Large Data Bases (VLDB)*, 1982.

[41] Giovanni Maria Sacco and Mario Schkolnick. Buffer management in relational database systems. *ACM Transactions on Database Systems (TODS)*, 11(4), 1986.

[42] Jiri. Schindler, Anastassia Ailamaki, and Gregory R. Ganger. Lachesis: Robust database storage management based on device-specific performance characteristics. In *International Conference on Very Large Data Bases (VLDB)*, 2003.

[43] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned extents: Matching access patterns to disk drive characteristics. In *USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[44] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *ACM SIGMOD International Conference on Management of Data*, 1979.

[45] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Database-aware semantically-smart storage. In *USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[46] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-smart disk systems. In *USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[47] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, 1981.

[48] Masamichi Takagi and Kei Hiraki. Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches. In *International Conference on Supercomputing (ICS)*, 2004.

[49] Theodore M. Wong and John Wilkes. My cache or yours? Making storage more exclusive. In *USENIX Annual Technical Conference*, 2002.

[50] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[51] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):505–519, 2004.