Analytical Cache Replacement for Large Caches and Multiple Block Containers

David Vengerov
david.vengerov@oracle.com

Garret Swart
garret.swart@oracle.com

Draft of 2011/02/14 14:48

Abstract

An important function of a storage system, such as a disk system, file system or database, is to cache its most frequently accessed data blocks in memory. The goal of caching is both to reduce the number of physical IOs and to reduce the expected latency of each logical IO and thus the aggregate wait time of the system. This paper presents the ANalytical Cache Replacement (ANCR) algorithm that analytically estimates the benefit of caching a particular block based either on the history of previous accesses or on the "container" to which the block belongs (a container could be either a virtualized storage unit, a directory, a file or a table containing the block). The algorithm then removes blocks from the cache probabilistically, with higher probabilities for less beneficial blocks (those which have a smaller probability of being accessed again and that belong to containers that are located on devices with a small access latency). As a result, the ANCR algorithm is biased to cache items retrieved from slower devices and items from containers with higher marginal hit rates as well as to give more space to new blocks when the cache is cold while giving more space to blocks with multiple prior accesses as the cache warms up. Extensive comparisons on simulated workloads with existing state-of-the-art caching algorithms show that ANCR performs competitively with the best algorithms for the case when all blocks belong to a single container and strongly outperforms all considered algorithms when blocks belong to multiple containers stored on devices with different access latencies.

1 Introduction

The primary purpose of a storage system is to store logical data items, organize them into containers, and provide access methods for these data items. In a relational database system the logical data items are records, the containers are tables or partitions of a table, and the access methods are table scan and row ID access. In a file system the logical data items are the bytes of a file, the containers are files or directories of files and the access methods are read and seek. In a disk system the data items are blocks, the containers are LUNs (virtualized storage units), and the access methods are read and write.

Data items are logical and often vary in size, so most systems find it easier to manage a cache of blocks. A block is a physical portion of a storage device and the typical unit for performing random IOs and for managing the cache. In a row organized relational database, a block is formatted as a slotted page, each slot holding a record from the table or table partition. In a file system, a typical block will hold a sequence of bytes from a file.

Each container has its own probability distribution for accesses to individual blocks, which is determined by a combination of the end user behavior and program behavior. Moreover, each container can have its own access latency, since designers often want to cache more frequently accessed data items in lower latency, higher cost, and possibly volatile memory. The goal often pursued by the designers is that of maximizing the logical IOs per second that can be executed, which implies that we want to fill up the cache with data blocks that have the highest product of access rate and cache miss latency.

In this paper we introduce ANalytical Cache Replacement (ANCR), a cache management algorithm that aims to achieve the goal described above. This algorithm is built upon the most popular framework of representing cache as a single block queue, where the decision about keeping or removing each cached block is made only at the tail of the queue. Among the popular algorithms that fit this framework, SLRU is the best one according to the recent surveys [6, 14], and ANCR strongly outperforms SLRU in all scenarios. Moreover, if one considers all *practical* algorithms (those that do not require manual parameter tuning) mentioned in [6, 14] regardless of their implementation framework (a single cache queue or multiple cache queues), then ANCR is shown to perform competitively with the best algorithms when all blocks belong to a single container and strongly outperforms all of them when blocks belong to multiple containers with different access latencies. Thus, the practical value of the work we present is that it shows organizations using the single cache queue framework that they do not need to invest significant efforts in changing their framework to the one using multiple cache queues and

can instead achieve the same or better performance as the best existing caching frameworks by simply modifying the rules according to which new blocks are inserted into the single cache queue and according to which stale blocks are removed from the end of the cache queue.

2 Related Work

A recent 2008-2009 two-part survey of Web proxy cache replacement strategies by ElAarag and Romano [14, 6] describes the most popular strategies cited in the literature and also compares the performance of these strategies on several real Web proxy traces. They classify the strategies as recency-based (simple strategies that rely exclusively on the time elapsed since the recent accesses to each data block), frequency-based (simple strategies that rely exclusively on the frequency of accesses to each data block), and recency-frequency strategies (strategies that combine both recency and frequency information) when choosing the block to be removed from the cache. The simplest and the most well-known recency-based strategy replaces the Least Recently Used (LRU) block from the cache, while the simplest and the most well-known frequency-based strategy replaces the Least Frequently Used (LFU) block from the cache. Some widely-cited examples of the recency-frequency strategies are Segmented LRU (SLRU) [9] and ARC [11]. In fact, the ARC algorithm is widely considered to be a state-of-the-art algorithm for caches with equally-sized blocks (as opposed to Web caches that can store documents of different size). The strategies mentioned above were simulated in Section 4. As expected, while showing a reasonable performance when all blocks belong to a single container, the above strategies performed very poorly when blocks were assigned to multiple containers with different access latencies because these strategies do not take into account the cost of fetching a block from disk after a cache miss.

ElAarag and Romano also add a fourth "function-based" class of strategies that use an analytical function combining the recency and frequency information for each data block in order to calculate the *value* of that block and then replace the block with the smallest value. Any function-based strategy can be modified to work well in a heterogeneous storage environment with different device access latencies by multiplying the value of each block by the latency of reading that block from its storage device and using that product as the final utility of that block. The ANCR algorithm belongs to this class of strategies (but it is constrained to remove stale blocks only from the end of the cache queue), and hence it is reasonable to compare ANCR with the best existing function-based cache replacement strategies and see if any of them can outperform ANCR due to their increased flexibility when selecting the block to be removed.

Some of the function-based strategies mentioned in [6] have many tunable parameters (e.g., MIX and M-Metric) that need to be set by hand for each particular workload, which makes those strategies impractical for real environments. Among the remaining strategies (LUV [2], TSP [18], GD-Size [4], GDSF [1], GD* [8], and LNC-R-W3 [16]), LNC-R-W3 was shown to have the highest hit rate. This strategy is very similar in spirit to the ANCR algorithm, as it makes its decision about removing a block based on the product of the estimated probability of future accesses to that block and the cost of a cache miss for the block (which is the latency incurred when fetching that block from the particular storage device it is residing on). Unlike ANCR, LNC-R-W3 incurs a much larger overhead when choosing the cached block to be removed, since it requires potentially searching through *all* cached blocks, computing the value of each block, and then selecting the block with the smallest value. We simulated performance of LNC-R-W3 in Section 4 and compared it with that of ANCR so as to determine how much potential performance is sacrificed by ANCR in order to achieve an increased computational efficiency when selecting a block to be removed. Surprisingly, we found that ANCR performs very similarly to LNC-R-W3 for the case when all blocks belong to a single container and strongly outperforms LNC-R-W3 for the multi-container case with different latencies.

Another widely cited function-based cache replacement strategy is LRV [15], which also estimates the value of each block as the product of the estimated access rate and the access latency. However, unlike LNC-R-W3 and ANCR, LRV uses the third popular strategy for selecting a cache block to be removed: it separates cached blocks into multiple FIFO queues (according to the number of hits they received) and then removes the lowest-valued block among the tail blocks of all the cache queues. Performance of the LRV strategy will be evaluated in Section 4 and compared with that of ANCR and LNC-R-W3.

Recently, a cache replacement algorithm called LEC was developed [3], whose authors show that it outperforms on real Web traces all other cache replacement strategies they considered: LRU, LFU, LNC-R-W3, GD-Size, LUV and an early non-adaptive version of LRV. Thus, LEC is a very worthy contender for being included in the list of benchmark strategies against which to compare the ANCR algorithm. Our experiments in Section 4 show that LEC indeed outperforms all other benchmark algorithms we simulated for the case when data blocks are allocated across multiple devices with different access latencies (but the ANCR algorithm still consistently outperforms LEC in this scenario for all cache sizes).

Some researchers have also experimented with a different approach to device-aware cache management: caching blocks from different storage devices in different partitions of the cache and then dynamically tuning the size of each partition so as to have larger partitions containing blocks from slower devices (for which the cache miss cost is higher). A recent example of this approach is [5], which cites an earlier work in [7] as being the "first cost-aware algorithm that utilizes the notion of aggregate partitioning. Both of these works try to equalize the number of accesses to each storage device so as to maximize the throughput of the heterogeneous storage system. This is a different goal from the one considered in this paper, which is that of minimizing the sum of latencies of all cache misses. We did find one thread of work where the authors are trying to partition the cache so as to achieve the latter objective, with [10] being a late example of their work. Unfortunately, the algorithm presented in that work was designed for the case of only two different storage devices and cannot be applied directly to more than two devices being present.

3 ANCR Algorithm

3.1 Basic ANCR algorithm

The ANCR algorithm is built upon the Segmented Least Recently Used (SLRU) algorithm [9]. In SLRU, the cache is represented by a single queue broken up into two segments: "protected" and "probationary," with the protected segment being "on top" of the probationary one. After a cache miss, the new block of data is placed at the top of the probationary segment, and if the cache queue has reached its maximal size, the block at the bottom of the queue is removed. Whenever a cached block gets hit, it is placed at the top of the protected segment.

ANCR starts with a simple modification of SLRU, which doesn't change its logic but just speeds up its execution: a cached block b that gets hit simply increments its hit counter n_b instead of being moved to the top of the protected segment. The counter n_b is initialized to 0 for a new block. When a block b reaches the bottom of the queue, it is removed if $n_b = 0$. On the other hand, if the block at the bottom of the queue has $n_b > 0$, then it is "recycled" to the top of the protected segment. This reduces the cost of processing a cache hit as the queue of blocks itself need not be locked or manipulated. In order to avoid filling the cache with every block that has ever been hit, we must reduce the hit count on blocks that are unlikely to be hit in the future. We can do this by resetting the hit counter, n_b, to 0 when the block is recycled to the top of the queue. This allows low probability blocks that got lucky hits to eventually be removed and makes the algorithm adaptive to changes in the workload.

The key idea of ANCR is to perform block caching so as to actually minimize the expected total cost of all cache misses over time. The cost of *not* keeping a block b in the cache is the product of the access rate to block b and the latency of retrieving that block from disk.

Let's define a block as being "old" if it was recycled at least once to the top of the protected segment (which would happen if the block was hit at least once after being placed into the cache). The expected access rate can be easily estimated for an old block b as $r_b = n_b/t_b$, where n_b is the number of times the block b was accessed and t_b is the time elapsed since b was first placed into the cache (which counts as the first access to the block b). Estimating the access rate for new blocks that did not get hit yet is more difficult. In order to resolve this difficulty, the ANCR algorithm makes another extension of the SLRU algorithm: it "recycles" some fraction of new blocks with $n_b = 0$ from the bottom of the queue to the top of the probationary segment and observes, for each container, the fraction of blocks that get hit during the second passage toward the bottom of the queue. These observations allow the ANCR algorithm to estimate the probability that a new block b with $n_b = 0$ at the bottom of the queue will be accessed if it is given a second chance and is recycled to the top of the probationary segment.

ANCR algorithm uses conditional probabilities to make such estimates. Let A be the event of a new block not getting hit during its first passage through the probationary segment, then being recycled to the top of that segment and then getting hit before reaching the bottom of the queue. Let B_j be the event of a new block belonging to container j not getting hit during its first passage through the probationary segment. Then, the conditional probability formula implies that $P(A|B_j) = P(A \cap B_j)/P(B_j)$, which can be estimated for each container j as the fraction of blocks that satisfy the event B_j and that get hit after being recycled to the top of the probationary segment. Finally, the access rate for a new block from container j that did not get hit during its first passage through the probationary segment is estimated as $R_j = P(A|B_j)/T_j$, where T_j is the average time spent in the cache by a new block from container j before getting hit during the second passage through the probationary segment. The exact steps for computing R_j are described in Section 3.3.

When deciding which block should be removed from the cache, the ANCR algorithm defines the potential "victim set" V as the last K blocks in the queue. The new blocks in V differ from each other only *probabilistically*, and so instead of always removing the last block in V unless it belongs to the container that has the highest estimated cache miss cost for new blocks, ANCR removes the new blocks probabilistically, assigning higher removal probabilities to new blocks from containers

with smaller expected new block costs. This makes ANCR more robust to "random flukes" in observed hits to new blocks from different containers, which could change the container that has the highest cache miss cost for new blocks. Also, this approach prevents ANCR from exhibiting extreme behaviors such as removing *all* new blocks from the low-cost containers, which would significantly hamper the adaptive potential of ANCR in the case of workload changes. In order to minimize the number of blocks that get recycled after each cache miss, the ANCR algorithm considers new blocks starting from the bottom of the queue and removes each block with appropriate probability using the following procedure.

Let C_j be the expected cache miss cost for a new block from container j that did not get hit during its first passage through the probationary segment, which can be expressed as $C_j = L_j R_j$, where L_j is the latency cost for a block from container j. Then, ANCR computes the probability P_j of removing such a block so that for any two containers j and k with new blocks in V, the relative removal probabilities are equal to the inverse relative cache miss costs: $P_j/P_k = C_k/C_j$. Any other way of assigning increasing removal probabilities to new blocks belonging to containers with smaller expected new block costs would also work, but the heuristic described above is simple enough to be used for demonstration purposes. Let the smallest cost among all containers with new in the victim set be $C_{min} = \min_{j \in V} C_j$ and let j_{min} be the index of the lowest-cost container. Then, $P_j = P_{j_{min}}(C_{j_{min}}/C_j)$, where the scaling factor $P_{j_{min}}$ is computed by solving $\sum_{j \in V} P_j = 1$, resulting in $P_{j_{min}} = 1/(C_{j_{min}} \sum_{j \in V} 1/C_j)$.

Let $C_{b,j}$ be the expected cache miss cost for an old block b from container j, which can be expressed as $C_{b,j} = L_j r_b$. The ANCR algorithm sequentially considers all new blocks in V starting from the bottom of the queue, and if the considered block b has already made two passages through the probationary segment and has $n_b = 0$, then it is chosen as the victim to be removed. If a new block b from container j is making its first passage through the probationary segment, has $n_b = 0$, and has the expected cache miss cost C_j smaller than that of the lowest-cost old block in V, then it is chosen as the victim with probability P_j. If no new blocks are chosen as victims after considering sequentially all new blocks in V, then ANCR chooses as the victim the first new block b from the bottom of the queue that has $n_b = 0$ and whose expected cache miss cost is smaller than the cost of the lowest-cost old block. If the victim set V does not contain any new blocks, then the old block in V with the smallest cache miss cost is chosen as the victim.

Since no block movement occurs in ANCR after cache hits, the potential victim to be removed after a future cache miss can be found asynchronously by a special thread. All blocks below the victim are then recycled back into the queue, either to the top of the protected segment if they have $n_b > 0$ or to the top of the probationary segment if they have $n_b = 0$. Note that a particular value of K for the size of the victim set V implies that a new block will stay in the cache for at least $ProbationarySegmentSize/K$ cache misses, since it gets "pulled" toward the bottom of the queue with the rate of at most K block positions per each cache miss.

The above procedure of choosing the victim block is not sufficient by itself make the ratios of new blocks among different containers equal to the ratios of expected cache miss costs for new blocks from those containers because some containers can have very high cache miss rates and can overwhelm the cache with new blocks: even if every one of their new blocks with $n_b = 0$ is removed at the bottom of the cache, there will still be too many of their new blocks present in the cache at every moment of time, not allowing the more valuable old blocks from other containers to stay in the cache. In order to avoid this problem, ANCR also controls the process of inserting new blocks into the cache.

The expected *a priori* cache miss cost for a new block from container j arriving into the cache can be computed as $C_j^0 = L_j E[N_j]$, where $E[N_j]$ is the expected number of hits to a new block from container j before it reaches the bottom of the queue (which is estimated by observing the number of times a new block from each container gets hit during its first passage through the probationary segment as described in Section 3.3). If the *actual* cache miss cost for a new block from container j were equal to C_j^0, then the correct decision would be to insert into the cache only new blocks from the container that has the highest cost. However, in reality, the estimated costs C_j^0 only relate *probabilistically* to the actual costs of new blocks, and hence the new blocks should be inserted into the cache probabilistically, with the insertion probability being an increasing function of C_j^0. This probability should also be a decreasing function of the new block arrival rate, so that no container overwhelms the cache with its new blocks.

The ANCR algorithm inserts new blocks into the cache using the following procedure. Let the largest *a priori* cost for new blocks among all containers be $C_{max}^0 = \max_{j \in V} C_j^0$ and let j_{max} be the index of the highest-cost container. When a cache miss occurs for a block from container j, this block is inserted at the top of the probationary segment with probability equal to C_j^0/C_{max}^0 if the arrival rate for new blocks from container j (denoted by λ_j) is less than or equal to the one from container j_{max}, and is otherwise inserted with

probability equal to $(C_j^0/C_{max}^0)(\lambda_{j_{max}}/\lambda_j)$.

Both the probabilistic inserting and the probabilistic removing schemes described above help the ANCR algorithm reduce the number of new blocks in the cache as the cache warms up, since only the most useful new blocks get inserted into the cache and then some of them would still get removed if they have a smaller cache miss cost than the old blocks. When the workload changes and new blocks start getting hit more often, the insertion probability for new blocks automatically increases, and once the expected hit rate for new blocks becomes larger than that of the least-accessed old blocks, the stale old blocks start getting removed from the cache.

3.2 Using a shadow list

When a block that was recently evicted from the cache is accessed again, it is reasonable to place that block at the top of the protected segment, assuming that the expected cost of not caching that block is high enough. In order to enable this behavior, the basic ANCR algorithm described above was extended by allowing it to use a "shadow list" that stores some information about the recently evicted blocks (implemented as a FIFO queue). The ARC, LNC-R-W3 and LRV algorithms also use a shadow list.

In order to prevent the recently evicted old blocks from being "washed out" quickly in the shadow list by the recently evicted new blocks (which get evicted from the cache at a much higher rate than the old blocks), the extended ANCR algorithm uses separate lists for new and old blocks evicted from the cache. The information about new blocks that get evicted from the cache after they have already made two passages through the probationary segment is not stored in the shadow list (such blocks are least likely to be hit again and so are not worth remembering). Each list entry contains the following 4 pieces of data: block index, container index, number of total accesses ever to this block and the time of the first access to this block. The last two pieces of data are being tracked for each cached block, and when a block is evicted from the cache and is placed into a shadow list, this data is copied into the shadow list entry for this block.

When the cache warms up sufficiently so that old blocks start getting removed from the cache, ANCR starts monitoring the expected cache miss cost of each old block that is removed from the cache (this cost is computed as a part of the probabilistic removal scheme in ANCR described in Section 3.1). When the number of old blocks removed reaches 100, ANCR records the standard deviation of the expected cache costs among these blocks and the maximum expected cache miss cost among them. It then computes the expected cost threshold for inserting a block from a shadow list into the cache as the maximum observed cost plus one standard deviation. The rationale for doing this is to make sure that blocks inserted from the shadow list at the top of the protected segment are valuable enough and do not get removed from the cache as soon as they reach the bottom of the cache queue. After 100 old blocks are observed, the counters are reset to 0 and ANCR once again starts observing the maximum expected cache miss cost and its standard deviation over the next 100 old blocks removed from the cache, then it computes the new expected cost threshold, and then the cycle repeats once again, etc.

When an access is made to a block b that is not in the cache but whose data is in a shadow list, the number of total accesses ever to the block is incremented by 1, and the expected miss cost for block b is computed just like for an old block (since b will necessarily have at least 2 accesses). Then, if the miss cost is higher than the expected cost threshold, the block b is inserted at the top of the protected segment of the cache and its data is deleted from the shadow list; otherwise, the block b is not inserted into the cache. In order to use more accurate estimates of the expected cache miss costs, ANCR requires for blocks in the new shadow list to be accessed at least twice (so that they would have at least 3 total accesses) before their expected cache miss cost is computed and compared with the cost threshold for inserting a block from a shadow list.

3.3 Collecting per-container statistics

The ANCR algorithm keeps a list of $ContainerStat$ objects, each of which contains some statistics about its corresponding container. Initially, this list is empty. When a cache access is made, ANCR checks whether the access is for a container with an existing $ContainerStat$ object, and if this is not the case, then a new $ContainerStat$ object is created. Initially, the $active$ variable in each such object is set to $FALSE$, and the only statistics being collected are the current number of new and old blocks in the cache from the container tracked by the object and also the current number of new and old blocks in the shadow list, if such a list is used. If, at some point, the total number of new and old blocks in the cache and in the shadow list becomes zero for any $ContainerStat$ object, the object gets removed from the list of such objects.

Initially, when the cache has not warmed up sufficiently (old blocks have not completely filled up the protected segment of the cache), the ANCR algorithm works just like SLRU. The probabilistic insertion and the probabilistic removal features are activated only when the cache warms up sufficiently. At that point, statistics required by these features is collected for T cache accesses while the algorithm still acts like

SLRU. The following specific variables are updated for each $ContainerStat$ object: $num\_first\_pass\_blocks$ (number of blocks that were inserted at the top of the probationary segment and then reached the bottom of that segment), $num\_first\_pass\_hits$ (the total number of cache hits to $num\_first\_pass\_blocks$), $num\_second\_chance\_blocks$ (number of blocks that were inserted at the top of the probationary segment, then reached the bottom of that segment without being hit, and were then recycled to the top of the probationary segment), num_good_blocks (number of blocks that were hit during their second passage through the probationary segment), and $cache\_accesses\_till\_first\_hit$ (average number of cache accesses between the moment a new block was placed into the cache and the moment that block was hit during its second passage through the probationary segment).

After T cache accesses are processed, the ANCR algorithm iterates through all $ContainerStat$ objects and for those that have $num\_first\_pass\_hits > 1$, $expected\_new\_block\_cost$ (that was denoted by $E[N_j]$ in Section 3.1) is computed as $latency \cdot num\_first\_pass\_hits/num\_first\_pass\_blocks$. For such containers the $active$ variable is set to $TRUE$ and $num\_first\_pass\_blocks$, $num\_first\_pass\_hits$ are reset to 0. Those containers that don't have $num\_first\_pass\_hits > 1$ keep using the old value of $expected\_new\_block\_cost$ and keep incrementing $num\_first\_pass\_blocks$ and $num\_first\_pass\_hits$ for another T cache accesses.

Similarly, after T cache accesses are processed, for those $ContainerStat$ objects that have $num\_good\_blocks > 1$, $expected\_0hit\_block\_cost$ (that was denoted by C_j in Section 3.1) is computed as $latency \cdot (num\_good\_blocks/num\_second\_chance\_blocks)$ / $cache\_accesses\_till\_first\_hit$ and the variables $num\_second\_chance\_blocks$, num_good_blocks, and $cache\_accesses\_till\_first\_hit$ are reset to 0. Those containers that don't have $num\_good\_blocks > 1$ keep using the old value of $expected\_0hit\_block\_cost$ and keep incrementing $num\_second\_chance\_blocks$, num_good_blocks, and $cache\_accesses\_till\_first\_hit$.

If the cache has warmed up sufficiently to activate probabilistic insertion and removal of blocks but the $active$ variable is $FALSE$ for some $ContainerStat$ object, then the new blocks from the corresponding container are inserted with probability 1 at the top of the probationary segment and removed with probability 0.5 if they were not hit by the time they reached the bottom of the probationary segment, so as to make sure that enough of such blocks get recycled to the top of the probationary segment and num_good_blocks can be accurately estimated.

Each $ContainerStat$ object also has a variable called $cache\_accesses\_since\_last\_block\_access$, which keeps track of the number of accesses made to the cache since the last time a block from this container was accessed. If $cache\_accesses\_since\_last\_block\_access$ becomes greater than T for some container, then its blocks start getting removed with probability of 1 when they reach the bottom of the cache, so as to speed up the adaptation of the cache to workload changes.

3.4 Filtering old blocks when containers have different disk access latencies

When containers have different disk access latencies, the ANCR algorithm may not be able to ensure a proper allocation of cache space among the *old* blocks simply by using a higher removal probability and a lower insertion probability for *new* blocks from low-latency containers. This will be especially apparent in large caches, where competition for cache space among the old blocks arises only when all high, medium and low-frequency blocks are cached in steady state and only the lowest-frequency blocks remain uncached. In that case, it may just take too long for ANCR to gradually "weed out" old blocks from low-latency containers, since such a process can happen only when there are no new blocks present in the victim set. In order to speed up this "weeding out" process, before searching for a victim among the new blocks, the ANCR algorithm with probability 0.5 considers all old blocks in the victim set and checks to see whether one of them should be removed. The specific steps that are performed are described in Figure 1, which gives the pseudo-code of the steps performed by ANCR after a cache miss occurs.

4 Experimental Results

A cache simulator was used to compare ANCR with LRU, LFU, SLRU, ARC, LNC-R-W3, LRV and LEC algorithms. For a cache size of N blocks, the probationary segment size for SLRU and ANCR was N/2. The statistics collection window T for ANCR was set equal to N and the victim set size K was set equal to N/100. The size of the old shadow list for ANCR was set to 25% of the cache size and the size of the new shadow list was set to 75% of the cache size. The ARC, LNC-R-W3, LRV, and LEC algorithms also used an LRU shadow list of size N to store information about the recently evicted blocks. The above parameter settings for ANCR were based on what seemed like good logical choices, but they can be optimized to improve the performance of ANCR beyond what is reported in this paper. However, discussion of the various optimization strategies for these parameters is outside the scope of this paper.

The authors of the LRV algorithm suggest that the cached blocks can be broken up into 10 groups (each

- If the block is in the shadow list, insert the block at the top of the protected segment.
- If the block is not in the shadow list, insert it at the top of the probationary segment.
- Find a block in the victim set to be removed:
  - Draw a random number z from a uniform distribution on [0,1].
  - IF ($z < 0.5$) THEN {
    - Look through all old blocks in the victim set and for each container k find the lowest-cost old block b_k
    - Let M be the index of the highest-latency container and let L_k be the latency of container k
    - Loop through all containers and if $L_k < L_M$ for container k, then compute $z_k = (r_k/r_M)/(L_M/L_k)$, where r_k is the access rate to b_k
    - Find $z_{min} = \min_k(z_k)$ and let $j = \text{argmin}_k(z_k)$
    - If $z_{min} < 1$, choose as victim the block b_j (lowest-cost old block from container j that had the lowest value of z_k).

    } ELSE {
    - Let b_{old} be the lowest-cost old block in the victim set with the cost C_{min}
    - For each new block b in the victim set starting from the last
      - IF (b has already been recycled to the top of the probationary segment and still has 0 hits) THEN select b as the victim
      - ELSE IF (b belongs to container k and $C_k < C_{min}$) THEN select the new block as the victim with probability P_k
    - If no victim had been chosen among the new blocks, then select b_{old} as the victim.

    }
  - Recycle blocks from the end of the queue until the victim block is reached:
    - IF (the tail block is a new block with 0 hits)
    - THEN recycle it to the top of the probationary segment
    - ELSE recycle it to the top of the protected segment
  - Remove the victim block

Figure 1: Pseudo-code of the steps performed by ANCR after a cache miss. C_k is the expected cache miss cost for a new block from container k that did not get hit during its first passage through the probationary segment and P_k is the probability of removing such a block.

implemented as an LRU list) according to the number of hits they received, and we followed this suggestion when implementing LRV. When selecting the block to be removed, the LRV algorithm removes the lowest-valued LRU block from the 10 queues. The final value of the LRU block b from queue j was computed as $V_b^j = L_b A_b^j$, where L_b is the latency of retrieving block b from disk and A_b^j is the estimated future access probability for the block b computed as $A_b = (1 - D(t - t_1))P(j)$, where t is the current time, t_1 is the time of the latest access to block b, $D(t)$ is the estimated probability distribution function of times between consecutive accesses to the same block and $P(j)$ is the estimated probability of a block that was accessed j times being accessed again in the future. Consistent with the suggestion given by the authors, we estimated $P(j)$ as B_{j+1}/B_j, where B_j is the number of cached blocks that were accessed j times. The function $D(t)$ was estimated using the procedure given in the Appendix B of [15] with the two parameters t_a and t_b being set to $N/10$ and N (consistent with the suggestion given by the authors about t_b being much larger than t_a). We also tried using $t_a = N/2$ and $t_b = 5N$ and obtained exactly the same results for LRV.

The final value of a block b in LNC-R-W3 was computed as $V_b = L_b M_b/(t - t_{M_b})$, where L_b is the latency of retrieving block b from disk, $M_b = min(n_b, K)$ with n_b being the total number of accesses to block b (it gets reset to 0 when the block data gets removed from the shadow list), K is a tunable parameter, t is the current time and t_j is the time of the jth previous access to block b. We tried various values of K for the LNC-R-W3 algorithm and found that the cache miss ratio monotonically decreased as a function of K and leveled off at around $K = 10$. Thus, we decided to use $K = 10$ in the experiments below, so as to be consistent with the 10 block groups used for LRV. When searching for the block to be removed, the algorithm first tries to remove the lowest-valued block from among those that received only 1 hit, then from among those that received only 2 hits, etc., and if all cached blocks have received 10 or more hits, then the algorithm searches linearly through all of them for the lowest-valued block.

The authors of the LEC algorithm suggest that the cached blocks can be broken up into groups (each implemented as an LRU list) according to the product of the number of hits they received and the disk access latency, and we used 10 groups for LEC to make it similar to LRV and LNC-R-W3 in this respect. When selecting the block to be removed, the LEC algorithm removes the lowest-valued LRU block from the 10 queues. The final value of the LRU block b from queue j was computed as $V_b^j = L_b A_b^j$, where L_b is the latency of retrieving block b from disk and A_b^j is the estimated future access probability for the block b computed as $A_b = n_b/(t - t_1)$,

where n_b is the number of hits the block b received so far, t is the current time and t_1 is the time of the latest access to block b.

4.1 Single-container experiment

The first experiment focused on the simple scenario of using only one container. Two different probability distributions for accessing data blocks were simulated. The first distribution was defined by the NURand(8191,1,100000) function (described below), which is used by the TPC-C benchmark (an industry-standard online transaction processing benchmark) for generating the item numbers for the "New-Order" transactions [17]. There are 100000 items that can be accessed by such transactions, and the particular item number for each access is chosen using the following procedure. First, a random integer A is drawn from a uniform distribution on [1, 8191] and another integer B is drawn from a uniform distribution on [1,100000]. Then, these integers are converted into a binary format and a third integer C is obtained by performing a bitwise logical OR operation on the corresponding bits of A and B. The final item number is equal to C modulo 100000 plus 1. The number of items accessed by each New-Order transaction is a randomly chosen integer from the range [5, 15].

The total number of transactions processed during a simulation run in this experiment was $20N$. The cache was warming up for $18N$ transactions and then the last $2N$ transactions were treated as an evaluation time period, over which the cache miss ratio (the fraction of TPC-C item accesses that resulted in a cache miss) was computed. Longer warmup times did not improve performance of the considered algorithms, suggesting that $18N$ transactions was enough for the cache to reach a steady state. Enough repetitions of each simulation run were performed so that the standard deviation of the average cache miss ratio for each algorithm would be less than 1% of the average.

The results of this experiment are presented in Table 1 for different values of the cache size N. As expected, the simple LRU and LFU algorithms obtained the largest cache miss ratio. The SLRU algorithm does not allow the new blocks to enter the protected area of the cache that holds old blocks. As a result, old blocks spend more time passing through the cache queue than new blocks (which get inserted in the middle of the cache queue) and hence they have a greater chance of being hit while traveling to the bottom of the queue. The SLRU algorithm recycles blocks that were hit to the top of the protected segment, which implies that old blocks (that have a higher chance of being hit) have a higher chance of being recycled to the top of the queue rather then being removed. In contrast, old blocks in LRU spend the same time passing through the cache queue as the new blocks (which get inserted at the top of the cache queue) and hence have a relatively smaller chance of being recycled to the top of the queue than in the SLRU algorithm. As a result, SLRU is able to keep more old blocks in the cache queue and obtained a smaller cache miss ratio than LRU in this experiment.

Policy	N**=5000**	**10000**	**20000**	**40000**
LRU	0.581	0.407	0.227	0.079
LFU	0.531	0.353	0.192	0.063
SLRU	0.501	0.343	0.187	0.065
ARC	0.482	0.339	0.199	0.074
LRV	0.428	0.303	0.174	0.075
LNC-R-W3	0.423	0.290	0.156	0.057
LEC	0.441	0.307	0.161	0.057
ANCR	0.421	0.294	0.157	0.053

Table 1: Cache miss ratios for simulated TPC-C New-Order transactions

Surprisingly, for large cache sizes (say $N = 40000$), SLRU even obtained a smaller cache miss ratio than the more sophisticated ARC algorithm, which is supposed to represent the state-of-the-art for caches with equally-sized blocks. In order to find out why this was the case, we first looked at the number of old blocks in the cache and found that ARC cached 0 or 1 new block from the new queue and the rest from the old queue (the total size of the old queue and the new queue in ARC can be at most $2N$, with at most N blocks from both queues being cached), while SLRU only had around 3/4 of old blocks in the queue (those that have been recycled to the top of the protected segment and are traveling down to the bottom of the queue). However, the total number of cache hits received by the old blocks in SLRU was only 1% smaller than in ARC, and when the hits received by the new blocks were added in, the total number of cache hits for SLRU became greater than for ARC. We measured the average access rate to old blocks in SLRU and found that it was indeed about 30% larger than for ARC. This observation suggested to us that ARC somehow selects lower quality "newly baked" old blocks to be added to the old queue in steady state.

The reason for this, as we discovered, is that when the cache size is large and the cache miss ratio is small, the outflow rate of new blocks from the new queue (due to them being hit and transferred to the top of the old queue) becomes equal to the inflow rate of new blocks due to cache misses *before* the new queue grows to N blocks. In particular, for $N = 40000$, the steady state length of the new queue in ARC was around 12000 and

that of the old queue was around 68000. Algorithmically, when a cache miss occurs in ARC and a new block is added to the top of the new queue, a block gets removed from the bottom of the new queue only if its length is equal to N. Otherwise, a block is removed from the bottom of the old queue if the total length of both queues is equal to $2N$, which is the case in the steady state.

Thus, for large cache sizes (such as $N = 40000$ or more) every new block in ARC eventually gets added to the top of the old queue as a "newly baked" old block. In contrast, under SLRU, new blocks get added to the top of the protected segment only if they get hit twice in a reasonably short period of time (while traveling from the middle of the cache queue to its bottom), which implies that SLRU has a much higher threshold (in terms of the required access rate) that needs to be surpassed by a new block in order to get promoted to the old list. This is the reason why the average access rate to old blocks under ARC was smaller than under SLRU and why the cache miss ratio for ARC for large cache sizes was higher than the cache miss ratio for SLRU.

The ANCR algorithm consistently obtained an even smaller cache miss ratio than either SLRU or ARC. The reason for its performance advantage over SLRU is clear: it removes an old block from the victim set only if there are no new blocks there with 0 hits, and it uses a more accurate criterion for determining the worst old block to remove (lowest expected cache miss cost) as opposed to a much more crude criterion used by SLRU (which removes a block at the bottom of the queue that was not hit during its latest passage through the queue).

In order to figure out why ANCR outperformed ARC for small cache sizes (we have already demonstrated that ARC manifests a clearly suboptimal handling of the old queue for large cache sizes), we first observed for $N = 10000$ that 99% of cached blocks for ARC were old blocks vs. 94% for ANCR, but the average access rate to old blocks for ANCR was about 15% larger than for ARC. This implies that ANCR somehow collected higher quality old blocks than ARC. The reason for this, as we discovered, is that ARC has a much larger turnover of old blocks than ANCR, constantly diluting the "proven" high-quality old blocks with low-quality newly arrived blocks. In particular, under ARC about $2N$ old blocks were removed from the bottom of the old queue during the evaluation period (approximately $20N$ cache accesses), while under ANCR only about $0.14N$ old blocks were removed. Thus, even for small cache sizes, ARC manifests a clearly suboptimal handling of the old queue.

Algorithmically, if the length of the new queue and of the old queue oscillates around N for ARC (which happens for small cache sizes such as $N = 20000$ or smaller in our experiments), then whenever a block in the new queue gets hit, it is moved to the top of the old queue, and the length of the new queue becomes less than N. Then, during the next cache miss, a new block is added to the top of the new queue and a block from the bottom of the old queue gets removed. We verified with our cache simulator that the number of old blocks removed for ARC is exactly equal to the number of hits to blocks in the new queue.

When a cache miss occurs in ANCR, a new block is inserted at the top of the probationary segment and then a block is removed from the victim set. If the victim set size is K and the Kth block from the bottom of the queue is removed, then $K - 1$ blocks are recycled from the bottom of the queue and the newly inserted block travels toward the bottom of the queue by $K - 1$ positions. Thus, the spacing between any two new blocks in the probationary segment is at most $K-1$. An old block is removed by ANCR only when there are no new blocks with 0 hits in the victim set (any new block with 0 hits had a smaller expected cost than any old block, and thus it would get removed if it were present in the victim set), which can happen only if a new block gets hit while it is in the probationary segment of the queue.

When $N = 10000$ and $K = N/100$, there are approximately $N/(2K) = 50$ new blocks in ANCR in a steady state, and hence the probability of accessing one of those 50 blocks is much smaller than the probability of accessing one of the 10000 blocks that are in the new queue for ARC in a steady state. This is the reason why ANCR has a much smaller turnover of old blocks, which allows it preserve in the cache the blocks that have the highest probability of being accessed without constantly diluting them with the "newly baked" old blocks.

The cache miss ratio of LRV, LNC-R-W3 and LEC is pretty much the same as that of ANCR, confirming the fact that these algorithms represent the current state-of-the-art in the field of cache replacement. In order to demonstrate that the relative performance of different algorithms in Table 1 does not depend on the particular probability distribution used by the TPC-C New-Order transactions, we repeated the experiments described above for a power law distribution (a heavy-tailed distribution), which is arguably more realistic than the one used in TPC-C. Power law distributions were shown to fit very well a wide variety of phenomena in the real world: city populations, computer files, the frequency of use of words in any human language, the frequency of occurrence of personal names in most cultures, the numbers of papers scientists write, the number of citations received by papers, the number of hits on web pages, the sales of books, music recordings and almost every other branded commodity, etc. [12]. Continuous probability distributions that follow a power law

Policy	N=5000	10000	20000	40000
LRU	0.497	0.405	0.301	0.180
LFU	0.469	0.376	0.274	0.161
SLRU	0.434	0.353	0.262	0.159
ARC	0.416	0.343	0.264	0.169
LRV	0.386	0.309	0.224	0.142
LNC-R-W3	0.391	0.313	0.227	0.134
LEC	0.418	0.329	0.237	0.139
ANCR	0.402	0.321	0.228	0.136

Table 2: Cache miss ratios for Zipf(0.9) distribution of block accesses (the 80-20 Pareto rule).

are called Pareto distributions, and the discrete ones are called Zipf distributions, after two early researchers who championed their study. Since we are dealing with discrete items in this work, we will refer to the power law distribution used to specify the access probability to individual blocks as the Zipf distribution.

Under Zipf distribution, P(access to block k) = $(1/k^{\alpha})P$(access to block 1), where α is the Zipf parameter. In order to get a sense of how the parameter α affects the shape of the Zipf distribution, consider the famous 80-20 rule (also known as Pareto principle) that states that roughly 80% of effect comes from 20% of the causes in many real world situations. In these terms, the Zipf parameter $\alpha = 1.5$ implies that approximately 97% of accesses are made to the top 3% of blocks (a very skewed access distribution), the Zipf parameter $\alpha = 0.9$ implies that approximately 80% of accesses are made to the top 20% of blocks (a realistic distribution observed in nature by Vilfredo Pareto), and the Zipf parameter $\alpha = 0.5$ implies that approximately 62% of accesses are made to the top 38% of blocks (a pretty flat access distribution). In our simulations we found that very skewed access distributions resulted in small performance differences between the various cache replacement algorithms, since in order to catch the majority of the cache accesses the algorithm needs to cache only a small fraction of the highest frequency blocks, which are very easy to detect using even the crude LRU methods. On the other hand, very flat access distributions also resulted in small performance differences between the various cache replacement algorithms, since it doesn't matter too much which blocks are cached under those distributions. Therefore, in order to make the problem most challenging, we chose Zipf(0.9) access distribution as the middle ground between skewed and flat access distributions, which also corresponds to the famous 80-20 rule deduced by Pareto.

Each transaction in this experiment accessed only one item in the table (as opposed to the average of 10 items for TPC-C), and hence in order to be consistent with the previous experiment, the number of transactions both during the warmup and during the evaluation phase was increased by a factor of 10. Table 2 shows the cache miss ratios of the considered algorithms under such a distribution. As expected, the observed relative values of the cache miss ratios were almost the same as in Table 1, but with LRV and LNC-R-W3 clearly outperforming LNCR for small caches. Apparently, the fact that ANCR allows new blocks to spend some time in the cache before removing them (unless they get hit again) gives it a slight disadvantage in small caches under some workloads, where frequently-accessed blocks quickly get into the cache and from that point on it is optimal to bias the algorithm strongly in favor of removing quickly any new blocks (such a bias is present in both LRV and LNC-R-W3 as we checked on our simulator).

4.2 Multi-container experiments

The Standard Specification for TPC-C [17] states: "Horizontal partitioning of tables is allowed. Groups of rows from a table may be assigned to different files, disks, or areas." With this in mind, we decided to repeat the experiments of the previous section for the case when the 100000 items were split into 5 equal containers holding the following ranges of item numbers: 1 - 20000, 20001 - 40000, 40001 - 60000, 60001 - 80000, 80001 - 100000. We also assigned different latencies to different containers so as to see how it would impact the relative performance of the previously considered cache replacement algorithms. The access latency in the currently popular storage devices ranges from 0.1 ms for a flash disk to 62.5 ms for an 84% loaded SATA disk (which has a service rate $\mu = 100$ IOPS, arrival rate $\lambda = 84$ IOPS, and latency $1/(\mu - \lambda) = 0.0625$ seconds). In order to cover this range of latencies, the latency of container j in this set of experiments was 2^{5-j}.

The total cache miss cost was used as the more appropriate metric for evaluating the cache replacement algorithms in the presence of different container latencies. It was computed as the total sum, over all cache misses, of latencies incurred when accessing missed blocks on storage devices. The results in Table 3 show that the ranking of the considered cache replacement algorithms is almost the same as in Table 1, with the notable exception that LEC now consistently outperforms LNC-R-W3 and LRV, and is the second best algorithm after ANCR. The reason why LEC outperforms LNC-R-W3 and LRV in this scenario is that these two algorithms try to remove blocks with fewer hits first, thereby often removing high-latency blocks. On the other hand, LEC separates blocks into queues according to the product of the number of hits they received and the disk

Policy	N=5000	10000	20000	40000
LRU	9.0	12.9	14.6	10.2
LFU	8.4	11.4	12.5	8.0
SLRU	8.0	11.0	12.2	8.3
ARC	7.6	10.9	12.9	9.5
LRV	6.7	9.7	11.3	9.8
LNC-R-W3	6.5	8.5	7.7	2.91
LEC	4.7	5.2	4.2	1.96
ANCR	3.8	4.0	3.1	1.29

Table 3: Cache miss costs in millions for simulated TPC-C New-Order transactions when the item database was partitioned into 5 containers with different latencies.

Policy	N=5000	10000	20000	40000
LRU	7.8	12.7	18.7	22.5
LFU	7.3	11.8	17.1	20.2
SLRU	6.8	11.0	16.4	20.0
ARC	6.5	10.7	16.5	21.3
LRV	6.0	9.6	13.9	17.7
LNC-R-W3	5.8	9.2	13.3	12.7
LEC	4.9	4.9	8.6	8.4
ANCR	4.5	6.1	5.9	2.5

Table 4: Cache miss costs in millions for the Zipf(0.9) distribution (Pareto 80-20 rule) when the items were partitioned into 5 containers with different latencies.

access latency, which helps it to avoid removing high-latency blocks simply because they received fewer hits than some lower-latency blocks.

The ANCR algorithm performs filtering of blocks from low-latency containers both at the level of new blocks (by learning to use a higher removal probability and a lower insertion probability for new blocks from low-latency containers) and also at the level of old blocks (by using the algorithm described in Section 3.4), which allows it to have proportionally larger fractions of old blocks from high-latency containers than in all other benchmark algorithms (as we verified in our simulator) and correspondingly to have the smallest total cache miss cost.

Note that column 2 in Table 3 has larger cache miss costs than column 1 because the evaluation period was equal to $2N$ and hence more misses took place during the evaluation period for $N = 10000$ than for $N = 5000$. Eventually, for $N = 40000$, the cache becomes so large that it covers almost all of the frequently accessed blocks, and even though more transactions get processed during the evaluation period, the actual number of cache misses decreases greatly, which explains why column 4 has smaller cache miss costs than column 3.

In order to make sure that ANCR not only outperforms the other algorithms but also make decisions that seem reasonable to human administrators, we printed some statistics at the end of the evaluation period from one simulation run for $N = 10000$. The final insertion probabilities for new blocks from each container were (1.0, 0.32, 0.07, 0.05, 0.05) and removal probabilities for new blocks with 0 hits were (0.003, 0.014, 0.024, 0.059, 0.90), which fits well with the fact that blocks from containers 1 and 2 are more valuable (since they have a higher cache miss latency). In order to ensure continued adaptability of ANCR (at the expense of some optimality) we decided to set 0.05 as the minimum insertion probability for any container so as to make sure that new blocks will keep passing through the cache and ANCR will be able to keep updating statistics about the new blocks. Just as expected, the above insertion and removal probabilities led to the following final cache composition for ANCR: (86, 61, 22, 31, 21) new blocks and (6629, 2675, 429, 37, 9) old blocks by container. As expected, ANCR allocated most of the cache space to old blocks while simultaneously giving a preference to caching old blocks from containers with higher access latencies.

Finally, in order to demonstrate that the relative performance of different algorithms in Table 3 does not depend on the particular probability distribution used for generating the TPC-C New-Order transactions, we repeated the above experiment for the Zipf(0.9) distribution of block accesses (Pareto 80-20 rule). The results are shown in Table 4 and they follow exactly the same relative pattern as the one in Table 3, which suggests that this pattern will hold for most other realistic block access distributions as well.

5 Conclusion

This paper presented a novel analytical cache replacement algorithm ANCR that explicitly estimates the expected cost of not caching a particular data block and then uses this information to bias cache occupancy in favor of containers with higher access probabilities and higher latencies. ANCR algorithm was compared using a cache simulator with the well-known caching algorithms such LRU, LFU and SLRU, as well as with the state-of-the-art caching algorithms ARC, LRV, LNC-R-W3 and LEC. Two different probability distributions for accessing individual blocks were simulated: the one used for generating the New-Order transactions in the TPC-C benchmark and the Zipf (power law) distribution (Pareto 80-20 rule). In all considered scenarios, the

ANCR algorithm consistently outperformed LRU, LFU, SLRU and ARC algorithms while performing competitively with LRV, LNC-R-W3 and LEC algorithms for the case when all blocks belonged to a single container. In the scenario where blocks belonged to multiple containers with different disk access latencies, ANCR strongly outperformed all other algorithms.

It is worth mentioning that such a good performance was obtained by ANCR despite the fact that it was constrained to use a single cache queue (consistent with the most popular queuing approach, LRU) and remove blocks only from the tail of the queue, while the other state-of-the-art caching algorithms either had to potentially search through the whole queue to find the worst block to remove (LNC-R-W3) or separated blocks into multiple cache queues (LRV and LEC), which creates additional operating overhead.

References

[1] M. Arlitt, L. Cherkasova, J. Dilley, R. J. Friedrich and T. Y. Jin, *Evaluating Content Management Techniques for Web Proxy Caches*, ACM SIGMETRICS Performance Evaluation Review 27/4 (2000), p. 3-11.

[2] H. Bahn, S.H. Noh, S.L. Min, K. Koh, *Efficient replacement of nonuniform objects in web caches*, IEEE Computer 35/6 (2002), p. 65-73.

[3] H. Bahn, *Web cache management based on the expected cost of web objects*, Information and Software Technology 47/9 (2005), p. 609-621.

[4] P. Cao, S. Irani, "Cost-aware WWW proxy caching algorithms," Proceedings of the First USENIX Symposium on Internet Technology and Systems (1997), p. 193-206.

[5] A. Chakraborty and A. Singh, "A Utility-based Approach to Cost-Aware Caching in Heterogeneous Storage Systems," In Proceedings of the Parallel and Distributed Processing Symposium (2007), p. 1-10.

[6] H. ElAarag and S. Romano, "Comparison of Function Based Web Proxy Cache Replacement Strategies," In Proceeding of the 12th International Symposium on Performance Evaluation of Computer & Telecommunication Systems (2009), p. 252-259.

[7] B. C. Forney, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. "Storage-Aware Caching: Revisiting Caching for Heterogeneous Storage Systems." In Proceedings of The First USENIX Conference on File and Storage Technologies (2002), p. 61-74.

[8] S. Jin, and A. Bestavros, "GreedyDual*: Web Caching Algorithms Exploiting the Two Sources of Temporal Locality in Web Request Streams," Proceedings of the 5th International Web Caching and Content Delivery Workshop (2000).

[9] R. Karedla, J. S. Love, B. G. Wherry, *Caching strategies to improve disk system performance*, Computer 27/3 (1994), p. 38-46.

[10] Y.-J. Kim and J. Kim, *DAC: a device-aware cache management algorithm for heterogeneous mobile storage systems*," IEICE Transactions on Information and Systems, 91-D/12 (2008), p. 2818-2833.

[11] N. Megiddo, D. S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," Proceedings of the 2nd USENIX Conference on File and Storage Technologies (2003), p 115-130.

[12] M. E. J. Newman, *Power laws, Pareto distributions and Zipf's law*, Contemporary Physics, 46/2 (2005), p. 323-351.

[13] S. Podlipnig and L. Boszormenyi, *A Survey of Web Cache Replacement Strategies*, ACM Computing Surveys, 35/4 (2003), p. 374-398.

[14] S. Romano and H. ElAarag, "A Quantitative Study of Recency and Frequency based Web Cache Replacement Strategies", Proceedings of the 11th Communication and Networking Symposium (CNS.08), Spring Simulation Multiconference (2008), p. 70-78.

[15] L. Rizzo and L. Vicisano, *Replacement Policies for a Proxy Cache*, IEEE/ACM Transactions on Networking, 8/2 (2000), p. 158-170.

[16] P. Scheuermann, J. Shim and R. Vingralek, "A Case for Delay-conscious Caching of Web Documents," Proceedings of the 6th International WWW Conference (1997), p. 997-1005.

[17] TPC BENCHMARK^{TM} C. Standard Specification. Revision 5.11. February 2010. http://www.tpc.org//tpcc//default.asp

[18] Q. Yang, H. H. Zhang, and H. Zhang, "Taylor Series Prediction: A Cache Replacement Policy Based on Second-order Trend Analysis, Proceedings of the 34th Hawaii International Conference on Systems Sciences (2001).