Cost-Aware Caching Algorithms for Distributed Storage Servers

Shuang Liang, Ke Chen, Song Jiang, Xiaodong Zhang

Technical Report
OSU-CISRC-7/07-TR52

Cost-Aware Caching Algorithms for Distributed Storage Servers

Shuang Liang[1], Ke Chen[2], Song Jiang[3], and Xiaodong Zhang[1]

[1] The Ohio State University, Columbus, OH 43210, USA
[2] University of Illinois, Urbana, IL 61801, USA
[3] Wayne State University, Detroit, MI 48202, USA

Abstract. We study replacement algorithms for non-uniform access caches that are used in distributed storage systems. Considering access latencies as major costs of data management in such a system, we show that the total cost of any replacement algorithm is bounded by *the total costs of evicted blocks* plus the total cost of the optimal off-line algorithm (OPT). We propose two off-line heuristics: MIN-d and MIN-cod, as well as an on-line algorithm: HD-cod, which can be run efficiently and perform well at the same time.
Our simulation results with Storage Performance Council (SPC)'s storage server traces show that: (1) for off-line workloads, MIN-cod performs as well as OPT in some cases, all is at most three times worse in all test case; (2) for on-line workloads, HD-cod performs closely to the best algorithms in all cases, and is the single algorithm that performs well in all test cases, including the optimal on-line algorithm (Landlord). Our study suggests that the essential issue to be considered be the trade-off between the costs of victim blocks and the total number of evictions in order to effectively optimize both efficiency and performance of distributed storage cache replacement algorithms.

1 Introduction

Widely used distributed storage systems have two unique features: storage device heterogeneity and multi-level caching management. Figure 1 shows an example of a multi-level heterogeneous multi-level heterogeneous distributed storage system, I/O buffer caches are installed at hierarchical levels. Access latencies to data blocks are no longer a constant due to non-uniform access times caused by heterogeneous storage devices and hierarchical caching. This adds another dimension to the management of distributed storage caches, which is a significant impact factor to the system performance. However, most existing replacement algorithms in practice focus on minimizing miss rate as the single metric for performance optimization, treating access latency as a constant. For example, recent studies on replacement algorithms such as 2Q, ARC, LIRS, and MQ mainly aim to improve the traditional LRU heuristic[4], which consider only block recency

[4] A brief overview of these algorithms is available in [1]

or balance both recency and frequency to reduce miss rate. These algorithms may not be suitable to manage caches of variable access latencies in distributed storage systems.

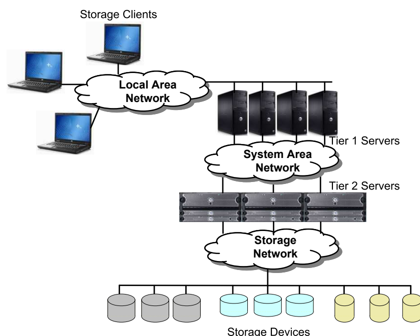


Fig. 1. An example of multi-level distributed storage systems with heterogeneous devices.

The replacement problem for caches with non-uniform access latencies can be modeled by the *weighted caching* problem, which can be solved off-line in $O(kn^2)$ time by reduction to the *minimal cost flow* problem [2], where k is the cache size and n is the number of total requests. However, this optimal algorithm is resource intensive in terms of both space and time for real-world system workloads, particularly when k and n are large. As an example, for a sequence of only $16K$ requests and a buffer cache of as small as 1.5 MBytes, the best known implementation of the minimum-cost flow algorithm [3] takes more than 17 GBytes of memory and multiple days to run on a dual-core 2.8GHz SMP Xeon server. Therefore, as current workloads and cache capacity continue to scale up, it becomes too unrealistic to timely make optimal replacement schedules.

In face of this problem, we study replacement algorithms for non-uniform access latency caches. Similar to previous studies, we use variable *cost* to model the non-uniform access latency in order to improve the efficiency and performance of replacement algorithms. In general, our model can be used for a distributed storage system with other non-uniform features, such as non-uniform energy consumption per access by instantiating costs as energy access consumption for different blocks to minimize total the energy consumption.

We show that for any replacement algorithm, the total access cost is bounded by *the total cost of the evicted blocks* (see Section 3) of the replacement algorithm plus the total access cost of the optimal algorithm (OPT). Therefore, the key to design variable-cost cache replacement algorithms lies in the trade-off between the number of evictions and the cost of victim blocks. Based on this principle, we propose two off-line algorithms: MIN-d and MIN-cod. Specifically, we take the variable cost consideration into MIN – the optimal replacement algorithm for uniform caches [4]. We found that choosing replacement victims based on the

ratio of cost and *forward distance* (see Section 3.1) is effective for minimizing the total costs. Using this heuristic, we also propose an on-line replacement algorithm HD-cod, which adaptively selects victims among blocks of largest recency from different cost groups.

We have evaluated the performance of the proposed algorithms with Storage Performance Council's storage server traces [5] by comparing our algorithms with OPT, Landlord [6] – a theoretically optimal on-line cost-aware algorithm , and other well-known cost-unaware replacement algorithms such as LRU, LFU, MRU, and Minimal Cost First (MCF). The results demonstrate that the proposed algorithms can be executed efficiently. Among all the algorithms, MIN-cod performs best in all cases, whose total cost is the same as OPT in some cases, and is at most four times of the lower bound of OPT in all cases. MIN-d performs similarly to MIN-cod when the cost distribution is small. In on-line scenarios, HD-cod performs close to the best algorithms in all cases and is the single algorithm that performs well in all test cases.

2 Preliminaries

The *weighted caching* problem [6] is defined as follows. Given a request sequence of data blocks:$r_1, ..., r_n$, each block has a cost (or weight) $\mathsf{cost}(r)$. For a cache of size k, upon each request r, if the requested block is not in cache(a miss), it is fetched in with cost(r). At the same time, one block in the cache is replaced to make space for r. If the requested block is already in cache, then no cost is involved. The goal is to minimize the total cost to serve the request sequence. An algorithm for the problem which assumes prior knowledge of the complete request sequence is an *off-line algorithm.* If an algorithm only knows the current and past requests in the sequence, then it is an *on-line algorithm.*

OPT: An Optimal Off-line Algorithm. Chrobak *et al.* [7] gave an optimal off-line algorithm for the weighted caching problem by reducing it to the *minimal cost maximum flow* problem [2]. Let $s_1, ..., s_k$ be the k cache pages. The algorithm builds a $(2 + k + 2n)$-node flow network. The vertex set of the network is $V = \{s, s_1, ..., s_k, r_1, ..., r_n, r'_1, ..., r'_n, t\}$, where s and t are the source and target nodes, respectively. For each $i = 1, \ldots, k$, there is an arc of cost 0 from s to s_i, and an arc of cost 0 from s_i to t. Similarly, a zero cost arc starts from each r'_j and ends at t. From each s_i, there is an arc to r_j, whose cost is equal to $\mathsf{cost}(r_j)$ if s_i does not occupy the block initially, or zero otherwise. For each $i < j$, there is an arc from r'_i to r_j whose cost is equal to $\mathsf{cost}(r_j)$ if $r'_i \neq r_j$, or zero otherwise. Finally, from r_j to r'_j, there is an arc of cost $-K$, where $K > 0$ is a very large number. Each arc in the network has a capacity of one. Figure 2 illustrates a flow network for a two-server and a three-request sequence.

Clearly, the maximum flow of this network is k. As $-K$ is very small, the minimum cost maximum flow of the network should include all the arcs from r_j to r'_j. Then each one of the k different paths from $s \rightarrow t$ provides the optimal schedule for the corresponding server to service its requests, that is, the caching

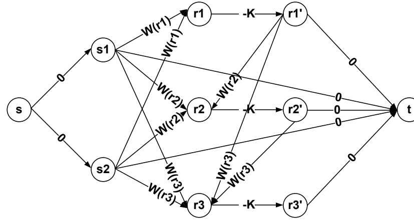


Fig. 2. A flow network constructed for an optimal weighted cache solution of a two-servers and a three distinct request sequence. The capacity of each edge in the network is one.

schedule for a particular cache block to cache different requested blocks. Since the minimum-cost maximum flow problem can be solved in $O(kn^2)$ time and the reduction step takes only $O(n^2 + k)$ time, the problem can be solved in $O(kn^2)$ in total. Therefore, this optimal offline algorithm is resource intensive, especially when k and n are large. For example, for a request sequence of one million blocks, the flow network has around one thousand billion arcs to process and needs several terabytes storage, which is well beyond the capability of current off-the-shelf servers.

2.1 MIN: An Off-line Algorithm for Uniform Cost Cases

Belady's MIN algorithm [4] is based on the assumption of a uniform block access cost. It always replaces the block to be requested furthest in the future. Belady proved that MIN can minimize the total number of misses, thus minimizing the total cost when the access cost to each block is uniform. Compared with OPT, MIN is much more efficient – it can be implemented in $O(n \log k)$ time and $O(n)$ space.

In a variable-cost cache, MIN's replacement decision can be far from optimal, since the furthest blocks can carry high fetch costs and lead to subsequent high miss penalties. Next we analyze its performance in variable-cost cases.

Definition 1. *A* cache configuration *is the set of (distinct) blocks resident in a given cache C.*

Given a request sequence S and an initial cache configuration $\mathsf{cfg(C)}$*, let* $\mathsf{tc}(\text{ALG}, \mathsf{cfg(C)}, S)$ *denote the total fetch costs incurred by an algorithm* ALG*; and let* $\mathsf{tf}(\text{ALG}, \mathsf{cfg(C)}, S)$ *denote the total number of fetches (misses).*

Let S be a request sequence, and $\mathsf{cfg(C)}$ be an initial cache configuration. Let $\mathsf{cost}_{min} = \min_{r \in S} \mathsf{cost}(r)$ and $\mathsf{cost}_{max} = \max_{r \in S} \mathsf{cost}(r)$ be the minimal fetch cost and the maximum fetch cost among all requests, respectively. it is easy to verify the following relationship.

$$\mathsf{tf}(\text{MIN}, \mathsf{cfg(C)}, S) * \mathsf{cost}_{min} \leq \mathsf{tc}(\text{OPT}, \mathsf{cfg(C)}, S) \leq \\ \mathsf{tc}(\text{MIN}, \mathsf{cfg(C)}, S) \leq \mathsf{tf}(\text{MIN}, \mathsf{cfg(C)}, S) * \mathsf{cost}_{max}$$

When $\mathsf{cost}_{max}/\mathsf{cost}_{min}$ is sufficiently small, MIN works pretty well; indeed, $\mathsf{tc}(\mathrm{MIN}, \mathsf{cfg}(\mathsf{C}), S)$ is bounded together with $\mathsf{tc}(\mathrm{OPT}, \mathsf{cfg}(\mathsf{C}), S)$ – the optimal cost by a narrow range. Actually, if a cache-resident block with an access cost of cost_{min} is to be referenced furthest in the future, it is the optimal victim candidate for replacement. However, it is not easy to choose victims when the furthest-to-be-referenced block has a non-minimal cost. Therefore, a new heuristic is needed to choose victims efficiently and accurately.

3 Cost-Aware Cache Replacement

A storage cache is a fully associative cache. It has two kinds of misses: *cold miss* and *capacity miss* [8]. Therefore, any replacement algorithm's total fetch cost can be divided into cold-miss cost and capacity-miss cost. Since cold miss is compulsory, it is the same for any algorithm including OPT, which means the cold-miss cost is no greater than the total cost of OPT. Therefore, we have the following observation.

Definition 2. *Given a request sequence S, on each fetch of any replacement algorithm, a block is* evicted *if it is replaced and is to be requested later in the remaining request sequence.*

Observation 1. Let S be a request sequence, $\mathsf{cfg}(\mathsf{C})$ be an initial cache configuration, $\textsc{Alg}$ be a replacement algorithm. Let $v_1, v_2, \ldots, v_m$ be the sequence of blocks evicted by $\textsc{Alg}$ when it serves S. It holds that $\mathsf{tc}(\textsc{Alg}, \mathsf{cfg}(\mathsf{C}), S) \leq \mathsf{tc}(\mathrm{OPT}, \mathsf{cfg}(\mathsf{C}), S) + \sum_1^m \mathsf{cost}(v_i)$.

Observation 1 shows that the key to design replacement algorithms for a weighted cache is the trade-off between the cost of eviction victims and the total number of evictions to minimize $\sum_1^m \mathsf{cost}(v_i)$. If a replacement algorithm is wise in choosing replacement victims such that no eviction is needed, then the total cost involved is the same as OPT. Otherwise, it performs at most $\sum_1^m \mathsf{cost}(v_i)$ worse than OPT.

In addition, unlike competitive analysis involving some unknown constant, observation 1 shows that a concrete upper bound of extra cost compared with OPT can be determined by simply adding the costs of all the evicted blocks, which can be implemented with negligible overhead in real systems. Such an upper bound is useful for evaluating the cache efficiency of a replacement algorithm. On the other hand, it can also be used to calculate the lower bound of OPT so as to estimate its total cost by deducting the eviction costs from the total costs of tested replacement algorithms.

Aimed at minimizing $\sum_1^m \mathsf{cost}(v_i)$, next we propose two off-line algorithms and one on-line algorithm.

3.1 MIN-d Algorithm

Our first algorithm, MIN-d, is an extension of MIN. It chooses the minimal-cost block from the $d+1$ furthest blocks ($d \geq 0$) as victim rather than choose the one

furthest block without consideration of costs. In particular, when $d = 0$, MIN-d is MIN. Before giving the details of MIN-d, depicted in Figure **??**, we make the following definition.

Definition 3. *Given a block r resident in the cache, the forward distance of r, denoted by $\mathsf{fwd}(r)$, is the number of distinct accesses from the current position to the next access of r in the request sequence.*

For example, suppose that the current cache configuration is $\{r_1, r_2, r_3\}$, and the remaining request sequence (that has not be served) is $r_2, r_4, r_5, r_4, r_3, r_1$. Then we have $\mathsf{fwd}(r_1) = 4$, $\mathsf{fwd}(r_2) = 0$, and $\mathsf{fwd}(r_3) = 3$.

Algorithm 1 MIN-d Algorithm

```
Input: request sequence S and initial cache configuration cfg(C).
for each request r in S do
  if r ∉ cfg(C) then
    Let Q ⊆ cfg(C) be the set of d + 1 resident cache blocks that have the largest
    forward distances.
    Let b ∈ Q be the block that has the smallest cost among Q. Replace b and read
    block r.
  end if
  Update cfg(C) and the forward distances of resident cache blocks.
end for
```

We claim that MIN-d's total number of misses is small, if d is relatively small compared to the cache size. Actually, the extra number of misses for MIN-d can be at most $n * \ln \frac{k-1}{k-d-1}$, where k is the cache size, n is the number of total requests.

Bound of Miss Count for MIN-*d* In what follows, we fix a request sequence S of length n. For simplicity of exposition, we assume that the ith request of S occurs at time i. Let $S[i, j]$ be the subsequence of S consisting of the requests at time from i to j (inclusive).

Given a set H and two elements x, y, the notation $H - x + y$ refers to the set $(H \setminus \{x\}) \cup \{y\}$.

Definition 4. *Let $\mathsf{seq}(t, b)$ be the first occurrence of a block b in the sequence S after time t; if b is not requested in S after time t then let $\mathsf{seq}(t, b) = \infty$. For example, if $S = \{a, b, a, d, c, b\}$, then $\mathsf{seq}(4, b) = 6$ and $\mathsf{seq}(3, a) = \infty$.*

Definition 5. *When S is being served by a given cache C, the cache configuration of C changes only when a miss occurs. Let $\mathsf{cur}_i(C)$ be the time when the ith miss occurs (namely, if the rth request of S incurs the ith miss, then $\mathsf{cur}_i(C) = r$). Let $\mathsf{cfg}_i(C)$ be the cache configuration of C after i misses have occurred. For example, $\mathsf{cfg}_0(C)$ is the initial cache configuration of C.*

Let $\mathsf{HS}_i(C) = \{\mathsf{seq}(\mathsf{cur}_i(C), b) \neq \infty \mid b \notin \mathsf{cfg}_i(C)\}$. In words, if a block b is not in the cache configuration $\mathsf{cfg}_i(C)$ and if b is requested after $\mathsf{cur}_i(C)$, then the first occurrence of b after $\mathsf{cur}_i(C)$ is in $\mathsf{HS}_i(C)$. Note that the next cache miss after $\mathsf{cur}_i(C)$, namely $\mathsf{cur}_{i+1}(C)$, would occur at the earliest time in $\mathsf{HS}_i(C)$.

Let H_1 and H_2 be two sets of positive integers, we write $H_2 \prec H_1$ if $|H_2 \cap [1, x]| \geq |H_1 \cap [1, x]|$, for any $x \geq 1$. For example, $\{1, 2, 4, 5, 7\} \prec \{2, 3, 5, 6\}$.

Claim 1. (i) Let H_1' and H_2' be the (nonempty) set after removing the smallest element from H_1 and H_2, respectively. If $H_2 \prec H_1$ then $H_2' \prec H_1'$.
(ii) Let $H_1' = H_1 + h_1$ and $H_2' = H_2 + h_2$. If $H_2 \prec H_1$ and $h_2 \leq h_1$, then $H_2' \prec H_1'$.

Lemma 1. *Given a request sequence S, let C_1 and C_2 be two caches such that $|C_1| = d + 1 + |C_2|$ and $\mathsf{cfg}_0(C_2) \subset \mathsf{cfg}_0(C_1)$. If MIN-$d$ is used for C_1 and MIN is used for C_2, then $\mathsf{cur}_r(C_2) \leq \mathsf{cur}_r(C_1)$ and $\mathsf{HS}_r(C_2) \prec \mathsf{HS}_r(C_1)$, for each $r \geq 0$ (satisfying $\mathsf{HS}_r(C_1) \neq \emptyset$).*

Proof. By induction on r. When $r = 0$, it is easy to verify that the claim holds. Assume the claim holds for $r = i$. Next we show that the claim holds for $r = i+1$.

Let $z_1 = \mathsf{cur}_{i+1}(C_1)$ and $z_2 = \mathsf{cur}_{i+1}(C_2)$. By definition, z_1 and z_2 are the earliest times in $\mathsf{HS}_i(C_1)$ and $\mathsf{HS}_i(C_1)$, respectively. This immediately implies $z_2 \leq z_1$, since $\mathsf{HS}_i(C_2) \prec \mathsf{HS}_i(C_1)$. It remains to prove $\mathsf{HS}_{i+1}(C_2) \prec \mathsf{HS}_{i+1}(C_1)$.

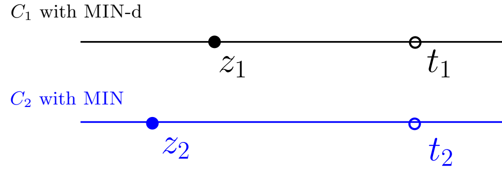


Fig. 3. Here, $z_1 = \mathsf{cur}_{i+1}(C_1)$ is the time when the $(i+1)$th miss occurs in C_1. At time z_1, the block h_1 is fetched into C_1 and v_1 is the victim block. The time t_1 is the first occurrence of v_1 (in S) after z_1. The numbers z_2 and t_2 are defined similarly on C_2. Note that $z_1 \geq z_2$, but t_1 may be smaller than t_2.

Let h_1 be the block being requested (in S) at time z_1, and v_1 be the victim block (in C_1) replaced by MIN-d at the same time. Let h_2 be the block being requested at time z_2, and v_2 be the victim block (in C_2) replaced by MIN at the same time. Let $t_1 = \mathsf{seq}(z_1, v_1)$ and $t_2 = \mathsf{seq}(z_2, v_2)$. See Figure 3.

For simplicity, we focus on the cases when $t_1 \neq \infty$ and $t_2 \neq \infty$, and omit the other cases (when $t_1 = \infty$ or $t_2 = \infty$) since they are similar. Consider the difference between $\mathsf{HS}_i(C_1)$ and $\mathsf{HS}_{i+1}(C_1)$. It is easy to see that z_1 is the earliest time in $\mathsf{HS}_i(C_1)$, and it is not in $\mathsf{HS}_{i+1}(C_1)$. Also note that t_1 is in $\mathsf{HS}_{i+1}(C_1)$ but not in $\mathsf{HS}_i(C_1)$. All other elements in $\mathsf{HS}_i(C_1)$ remains unchanged in $\mathsf{HS}_{i+1}(C_1)$. Therefore, it holds that

$$\mathsf{HS}_{i+1}(C_1) = \mathsf{HS}_i(C_1) - z_1 + t_1. \tag{1}$$

Similarly, we have

$$\mathsf{HS}_{i+1}(C_2) = \mathsf{HS}_i(C_1) - z_2 + t_2. \tag{2}$$

1. $t_1 \geq t_2$. By Claim 1 (i), we have $\mathsf{HS}_i(C_1) - z_1 \prec \mathsf{HS}_i(C_2) - z_2$, since z_1 and z_2 are the earliest times in $\mathsf{HS}_i(C_1)$ and $\mathsf{HS}_i(C_1)$, respectively. Now, by Claim 1 (ii), it is easy to verify $\mathsf{HS}_{i+1}(C_1) \prec \mathsf{HS}_{i+1}(C_2)$, since $t_2 \leq t_1$.
2. $t_1 < t_2$. We need to show that $|\mathsf{HS}_{i+1}(C_2) \cap [1, x]| \geq |\mathsf{HS}_{i+1}(C_1) \cap [1, x]|$, for any integer $x \geq 1$.

   (i) $x \leq z_1$. We have $\mathsf{HS}_{i+1}(C_1) \cap [1, x] = \emptyset$ (recall that z_1 is the earliest time in $\mathsf{HS}_i(C_1)$, which implies that the earliest time in $\mathsf{HS}_{i+1}(C_1)$ is larger than z_1). The claim trivially follows.

   (ii) $z_1 < x < t_1$. Note that $z_2 \leq z_1$ and $t_1 < t_2$. By Eq. (1) and Eq. (2), we have $|\mathsf{HS}_{i+1}(C_1) \cap [1, x]| = |\mathsf{HS}_i(C_1) \cap [1, x]| - 1$ and $|\mathsf{HS}_{i+1}(C_2) \cap [1, x]| = |\mathsf{HS}_i(C_2) \cap [1, x]| - 1$. The claim immediately follows, by the induction hypothesis $|\mathsf{HS}_i(C_1) \cap [1, x]| \leq |\mathsf{HS}_i(C_2) \cap [1, x]|$.

   (iii) $x \geq t_1$. Note that $\mathsf{HS}_{i+1}(C_1) \cap [1, z_1] = \emptyset$ and $\mathsf{HS}_{i+1}(C_2) \cap [1, z_2] = \emptyset$, by similar arguments to (i). As such, it suffices to prove

   $$|\mathsf{HS}_{i+1}(C_1) \cap [z_1 + 1, x]| \leq |\mathsf{HS}_{i+1}(C_2) \cap [z_2 + 1, x]| . \tag{3}$$

   Let B_1 be the set of distinct blocks in $S[z_1+1, x]$, and $R_1 \subseteq B_1$ be the set of distinct blocks in $S[z_1 + 1, x]$ that are in $\mathsf{cfg}_{i+1}(C_1)$. Similarly, let B_2 be the set of distinct blocks in $S[z_2+1, x]$, and $R_2 \subseteq B_2$ be the set of distinct blocks in $S[z_2 + 1, x]$ that are in $\mathsf{cfg}_{i+1}(C_2)$. Now, notice that the LHS of Eq. (3) is the number of distinct blocks in $S[z_1 + 1, x]$ that are not in $\mathsf{cfg}_{i+1}(C_1)$, which is equal to $|B_1| - |R_1|$, and the RHS of Eq. (3) is the number of distinct blocks in $S[z_2+1, x]$ that are not in $\mathsf{cfg}_{i+1}(C_2)$, which is equal to $|B_2| - |R_2|$. Therefore, we need to prove that

   $$|B_1| - |R_1| \leq |B_2| - |R_2| .$$

   By the MIN-d algorithm, v_1 is one of the $d+1$ furthest block to be requested at time z_1. Therefore, $|R_1| \geq |C_1| - (d + 1) = |C_2|$. On the other hand, $|R_2| \leq |C_2|$. It follows that $|R_1| \geq |R_2|$. Furthermore, by the definition, $B_1 \subseteq B_2$, since $z_1 \geq z_2$. This implies that $|B_1| \leq |B_2|$. It thus follows that $|B_1| - |R_1| \leq |B_2| - |R_2|$, as required.

□

The following corollary is straightforward:

Corollary 1. *Let S be a request sequence, C_1 and C_2 be two caches such at $|C_1| = |C_2| + d + 1$ and $\mathsf{cfg}_0(C_1) \supseteq \mathsf{cfg}_0(C_2)$. It holds that* $\mathsf{tf}(\text{MIN-}d, \mathsf{cfg}(\mathsf{C}_1), S) <= \mathsf{tf}(\text{MIN}, \mathsf{cfg}(\mathsf{C}_2), S)$.

Lemma 2. *Let S be a given request sequence and C_1 and C_2 be two caches such that $|C_1| = |C_2| + 1$ and $\mathsf{cfg}(\mathsf{C}_1) \supseteq \mathsf{cfg}(\mathsf{C}_2)$. The number of fetches by* MIN *on C_2 is at most $n/|C_2|$ larger than the number of fetches by* MIN *on C_1. That is,* $\mathsf{tf}(\text{MIN}, \mathsf{cfg}(\mathsf{C}_2), S) \leq \mathsf{tf}(\text{MIN}, \mathsf{cfg}(\mathsf{C}_1), S) + n/|C_2|$.

Proof. Let cfg_i and $\overline{cfg}_i$ be the configurations of C_1 and C_2 after the ith element of S has been served by MIN, respectively. First we prove $\overline{cfg}_j \subseteq cfg_j$, for $j = 0, \ldots, n$. Since $|C_1| = |C_2| + 1$, this implies that only one element is in cfg_i but not in $\overline{cfg}_i$, and we call this element the *bubble* of cfg_i, denoted by bb_i. We define the rank of bb_i, denoted by $rank(\mathsf{bb}_i)$, to be the number of elements in cfg_i that have smaller forward distances than bb_i (namely, those elements will be requested before bb_i).

Clearly, it holds that $\overline{cfg}_0 \subseteq cfg_0$. Now suppose that $\overline{cfg}_i \subseteq cfg_i$, we show $\overline{cfg}_{i+1} \subseteq cfg_{i+1}$ in what follows. Let s_{i+1} be the $(i+1)$th element requested in S. There are three cases:

(i) s_{i+1} is in both cfg_i and $\overline{cfg}_i$. Then $cfg_{i+1} = cfg_i$ and $\overline{cfg}_{i+1} = \overline{cfg}_i$, and as such, $\overline{cfg}_{i+1} \subseteq cfg_{i+1}$. We have $\mathsf{bb}_{i+1} = \mathsf{bb}_i$ and $rank(\mathsf{bb}_{i+1}) \geq rank(\mathsf{bb}_i) - 1$.

(ii) s_{i+1} is in cfg_i but not in $\overline{cfg}_i$. In this case, we have $\mathsf{bb}_i = s_{i+1}$ and $rank(\mathsf{bb}_i) = 0$. Let f be the element having the largest forward distance in C_2, note that f is also the element having the largest forward distance in C_1 (because $C_1 = C_2 + s_{i+1}$ and s_{i+1} has a forward distance 0). By the algorithm MIN, we have $cfg_{i+1} = cfg_i$ and $\overline{cfg}_{i+1} = \overline{cfg}_i - f + s_{i+1}$. It is easy to verify that $\overline{cfg}_{i+1} \subseteq cfg_{i+1}$ still holds, and the new bubble bb_{i+1} is f. We have $rank(\mathsf{bb}_{i+1}) \geq |C_2| = |C_2| + rank(\mathsf{bb}_i)$.

(iii) s_{i+1} is in neither $\overline{cfg}_i$ nor cfg_i. Then MIN fetches s_{i+1} into both C_1 and C_2, and evicts one element from each of C_1 and C_2. It is easy to verify that $\overline{cfg}_{i+1} \subseteq cfg_{i+1}$ and $rank(\mathsf{bb}_{i+1}) \geq rank(\mathsf{bb}_i) - 1$.

The above arguments have shown that $\overline{cfg}_j \subseteq cfg_j$, for $j = 0, \ldots, n$. Note that an extra fetch (that is, a fetch that occurs for C_2 but not for C_1) occurs only in the case (ii) above. Now, the crucial observation is that case (ii) cannot happen too often (by analyzing the rank of bb_i). We omit the further details. □

The following theorem follows from Corollary 1 and Lemma 2.

Theorem 1. *Let S be a request sequence of length n and C be a cache of size k. We have $\mathsf{tf}(\text{MIN-}d, \mathsf{cfg(C)}, S) \leq \mathsf{tf}(\text{MIN}, \mathsf{cfg(C)}, S) + n \ln \frac{k-1}{k-d-1}$, namely, the* MIN-$d$ *algorithm performs at most $n \ln \frac{k-1}{k-d-1}$ more fetches than* MIN.

Proof. Let C_{i+1} be a cache such that $|C_{i+1}| = |C| - (i+1) = k - i - 1$ and $\mathsf{cfg(C_{i+1})} \subseteq \mathsf{cfg(C)}$, for $i = 0, \ldots, d$. By Corollary 1, it holds that

$$\mathsf{tf}(\text{MIN-d}, \mathsf{cfg(C)}, S) <= \mathsf{tf}(\text{MIN}, \mathsf{cfg(C_{d+1})}, S).$$

By Lemma 2, for each $i = 0, \ldots, d$, it holds that

$$\mathsf{tf}(\text{MIN}, \mathsf{cfg(C_{i+1})}, S) \leq \mathsf{tf}(\text{MIN}, \mathsf{cfg(C_i)}, S) + \frac{n}{k-i-1},$$

which implies that $\mathsf{tf}(\text{MIN}, \mathsf{cfg(C_{d+1})}, S) \leq \mathsf{tf}(\text{MIN}, \mathsf{cfg(C)}, S) + \sum_{i=0}^{d} \frac{n}{k-i-1}$. Since $\sum_{i=0}^{d} \frac{n}{k-i-1} \leq n \ln \frac{k-1}{k-d-1}$, the claim follows.

3.2 MIN-cod Algorithm

The MIN-d algorithm takes block cost into consideration for replacement decisions without significantly increasing the number of misses. However, it is conservative in nature as the scope of the candidate victim blocks is small ($d+1$ furthest blocks). In reality, it is possible that some blocks to be accessed recently are much cheaper than the $d+1$ furthest blocks such that evicting those near blocks to keep those expensive blocks despite of more misses is still beneficial. Obviously, MIN-d cannot make efficient decisions in these cases, thus its performance is limited, especially when the cost differences among the blocks are large. Therefore, we propose an algorithm that more aggressively pursues an optimal trade-off between the number of evictions and block costs by considering every block as a potential replacement candidate.

Algorithm 2 MIN-Cod Algorithm

```
Input: request sequence S and initial cache configuration cfg(C).
for each request r in S do
  if r ∉ cfg(C) then
    Let b ∈ cfg(C) be the resident cache block that has the smallest Cod value cost(b)/fwd(b)
    in current cache configuration cfg(C). If there is a tie, choose b as the one with
    largest forward distance. Replace b and read block r.
  end if
  Update cfg(C) and the forward distances of resident cache blocks.
end for
```

As described in Figure ??, the algorithm MIN-cod makes replacement decisions based on the ratio of the cost over forward distance (Cod) among all the resident blocks in cache. If two blocks have the same ratio, the block with a larger forward distance is chosen. Clearly, if a block has the minimal cost among all resident blocks and is the furthest block, MIN-cod will replace it upon a miss, which is necessary for OPT too. However, if one block has a smaller cost and a shorter forward distance than another block, then it is unclear which one is a better victim block to reduce the total cost. Note that the number of evictions for keeping a block is closely related with its forward distance. Assuming keeping a block is beneficial, then it must not be evicted before its next request, otherwise the sooner it is evicted the better so as to save space for other blocks. Since the space for this block is occupied from the current request to the block's next request, keeping a block can be viewed as effectively reducing the cache size by one during this period. Based on the reasoning of Lemma 2, it is not difficult to know that the upper bound of extra misses caused for keeping this block in comparison to keeping a nearer block is roughly in proportion to its forward distance. Therefore, the $cost/fwd$ essentially represents the minimal average cost savings per extra miss. The Cod heuristic chooses to replace the furthest block that generates the smallest saving.

The Running Times of MIN-d and MIN-cod A naive implementation of MIN-d and MIN-cod, on each request, scans the resident blocks in cache to find the victim and update `fwd` values. Therefore, the total execution times of both algorithms are $O(nk)$.

There are two observations that can lead to a faster implementation for MIN-d, which uses only $O(n \log k)$ time. First, MIN-d only requires to maintain the *relative* forward distances among the blocks to choose victims from. Second, after serving a new request, the cache configuration changes by only one element, and at most one resident cache block changes its relative forward distance. Therefore, if we use a priority queue to maintain the (relative) forward distances of resident blocks, we only need $\log k$ time for processing each request, resulting a total running time of $O(n \log k)$.

In real systems, the number of different fetch costs of blocks is relatively small, because only a limited number of different storage devices and levels exist in a system. Therefore, by keeping the resident blocks in binary trees of different costs, MIN-cod only needs to compare the blocks of the largest (relative) forward distance within each tree to find the right victim, whose total execution steps are in proportion to the number of different trees, thus can be considered as $O(1)$. Since the maintenance of each binary tree needs $O(\log k)$ time, the overall running time is bounded by $O(n \log k)$, which is much faster than OPT.

3.3 An On-line Algorithm HD-cod

The off-line algorithms assume complete knowledge of future requests, which is not always realistic in practice. In this section, we present HD-cod, an on-line algorithm based on MIN-cod.

In on-line algorithms, we can only estimate the forward distance of a resident block. To this end, we use the *recency* of a resident cache block b as the estimated forward distance of b. (Recency is a concept borrowed from the well-known LRU replacement algorithm.) More specifically, the recency of b is the difference between its current request sequence number and the request sequence number of the last request of b.

It is widely recognized that the LRU replacement algorithm, which estimates the forward distance of a block by its recency, works well for most workloads with strong temporal locality. However, it performs poorly for workloads with weak locality such as those with looping or random access patterns, where a recent access of a block does not indicate its re-access is near. These observations suggest that different considerations of forward distance can be used when evaluating Cod value of each block for the choice of replacement victims.

In HD-cod, we use $\frac{cost}{fwd^{\alpha}}$ to evaluate each block, where α is a workload dependent parameter in $[0, 1]$. For workloads with LRU-like temporal locality, α approaches 1 because the forward distance estimation is accurate, so that the Cod heuristic is appropriate. For non-LRU-like workload, α approaches 0 because the estimation of forward distance is inaccurate and the forward distance becomes less relevant, so that the replacement decision can be more dependent on the cost.

Algorithm 3 Calculating α

```
Input: c[i] - total hit count of region i (the region number increases as the recency increases;
Input: cum[i] - cumulative hit count of the first i regions;
Input: seg - total number of regions;
Input: peak - a constant threshold to identify a peak in the density curve.
α = 0; flag = 0;
for each region i do
  r = c[i] / c[i−1]
  if (r > peak and cum[i] < 0.5) then
    α = α − 2^(1−i);
  else
    if (1/r > peak and cum[i] > peak ∗ i/seg) then
      if (flag! = 0) then
        α = α + 2^(1−i);
      else
        flag = 1;
        α = α + cum[i];
      end if
    end if
  end if
end for
```

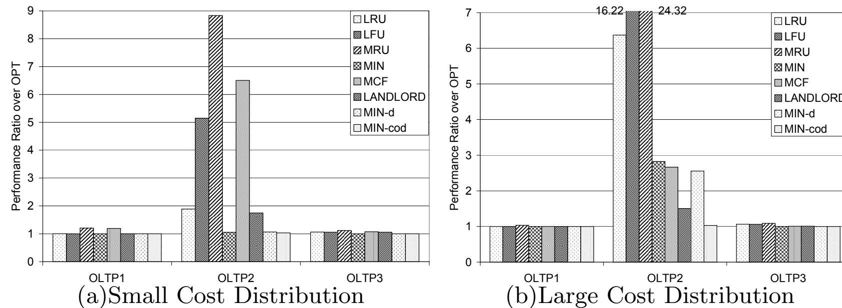
To determine the workload type, HD-cod maintains an LRU queue (ordered by recency) for all the resident blocks. It divides the queue into multiple contiguous regions of a fixed size, which is a system run-time parameter and usually small. HD-cod traces the hit count in each recency region to generate the hit density curve of the workloads, so that α can be set dynamically based on the locality feature. Therefore the algorithm is called Hit Density(HD)-cod. For workload with LRU-like locality, the hit density continuously decreases with the first few regions holding most hits. For non-LRU-like locality, the hits are distributed among the regions irregularly. On each replacement decision, HD-cod walks through the regions to calculate α, as shown in Algorithm 3. Since HD-cod maintains each queue using LRU order whose overhead is very small, its time complexity is $O(n)$.

4 Evaluation

Methodology: We evaluate our proposed algorithms through trace-driven simulation. We compare the performance of our proposed algorithms with the optimal off-line algorithm OPT, representative non-cost-aware algorithms including MIN, Least Recently Used (LRU), Most Recently Used (MRU), Most Frequently Used (LFU) as well as cost-aware algorithms including Landlord and Minimal Cost First (MCF). Landlord is an optimal on-line cost-aware caching algorithm [9, 6]. Upon each replacement, it chooses the block with the minimal residual cost as victim, and decreases each resident block's residual cost by this

minimal value. Then upon each hit, the block's residual cost is updated by a value that is between current value and its original cost. It is proved in [6] that Landlord is a k-competitive algorithm, hence an optimal on-line replacement algorithm. It is also a generalization of the GreedyDual algorithm [10], which is studied in WWW-proxy cache management.

The traces used in our experiments are production storage I/O traces from Storage Performance Council (SPC) [5] – a vendor-neutral standards body. They include both OLTP application I/O and search engine I/O. The OLTP traces are with strong temporal locality, i.e. repeated accesses to the same block, if any, are usually separated by a small (compared with cache size in blocks) count of accesses to other blocks. And the OLTP traces also include a significant portions of concurrent sequential accesses due to both the nature of server workloads and OLTP itself. The search engine traces comprise mostly random accesses, which are mostly non-sequential accesses with weak temporal locality, i.e. repeated accesses to the same block, if any, are usually separated by a large (compared with cache size in blocks) count of accesses to other blocks. Due to the resource-intensive nature of the OPT algorithm, which makes replaying the complete trace computationally intractable on our system, we split an entire trace into smaller traces by the ASU (Application Specific Unit) field of each request, so that logically related requests are grouped in the same trace file. We randomly generate the cost for each block based on two cost distributions. One cost distribution spans a wide range with differences as large as 70,000 times, the other spans a small range with differences at most 3 times. The details of the cost distribution is listed in table 1 and 2 (see Appendix).



(a)Small Cost Distribution (b)Large Cost Distribution

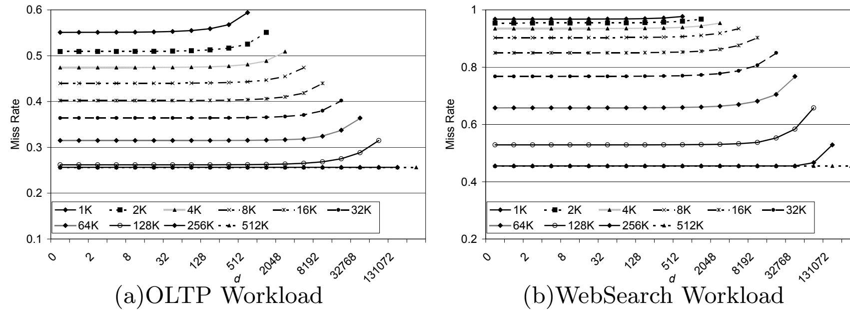
Fig. 4. Total cost comparison of three workloads between OPT and different algorithms

Impact of Cost-Aware Replacement: We compare OPT with both existing and our proposed replacement algorithms to evaluate the impact of weighted cache on replacement performance. In this experiment, we set one eighth of the working set size as the cache size.

Figure 4 shows the performance ratio of various algorithms over OPT for three small workloads using two different cost distributions. Overall, a significant performance degradation is observed using non-cost-aware replacement algorithms. For example, the optimal non-cost-aware algorithm MIN is 182% worse than OPT for OLTP2 using large cost distribution, so is LRU 537% worse. In contrast, the cost-aware algorithms including Landlord, MIN-d, and MIN-cod perform better almost in all these cases. The only exception is for the small cost distribution, MIN performs closer to OPT than Landlord, since in these scenarios, miss count is more important in the trade-off for overall performance.

The results also show that the extent of the performance degradation is related with the workload itself. As we can see, OLTP2 is much more sensitive to the cost-awareness of the algorithms than the other two, because the other two traces have very few reused blocks.

Miss Rate of MIN-d: Figure 5 show the miss rate impact of MIN-d on both OLTP and WebSearch workloads. The OLTP workload has a working set of around $300K$ blocks, while the WebSearch workload has a working set of around $480K$. The results show that the miss rate does not increase noticeably until d is larger than 6% of the cache size. [5] Based on this empirically result, in the following experiments, we set d to be 1/16 of the cache size.



(a)OLTP Workload (b)WebSearch Workload

Fig. 5. The impact of d on miss rate for two different workloads. The experiments are run with 10 different cache sizes. For each size, we vary the value of d to half the cache size to evaluate the impact on miss rate. The results shows the impact is very small up to 1/16 of the cache size

Off-line Algorithm Results: Figure 6 shows the experiment results comparing off-line algorithms (MIN-d and MIN-cod) with existing off-line algorithm MIN.

[5] Please note this results include cold miss, therefore when cache size is large than working set, there are still significant miss rate for the workloads.

We also include the comparison with on-line algorithms: MCF and Landlord for two reasons: a) MIN and MCF represent two ends on the trade-off between miss count and cost. MIN considers only misses in replacement decisions, while MCF considers only cost; b)since Landlord is the state-of-the-art cost aware replacement algorithm, the inclusion of it gives us an idea on the benefit of complete request knowledge to variable-cost cache replacement.

Overall, MIN-cod consistently performs the best for all workloads. It reduces up to 62% of the total cost of MIN and up to 50% of Landlord. Other algorithms perform differently depending on the workloads. For example, as expected, MIN-d performs close to MIN-cod for the small cost distribution; however it is significantly worse than MIN-cod for the large distribution due to its scope limitation of victim candidates. Compared with Landlord, MCF performs poorly for the OLTP workloads, while it performs better for the Websearch workloads. Using the lower bound measured, it also shows that when the cache size is 50% of the workload, in seven out of the eight cases MIN-cod performs the same as OPT, while for one of the WebSearch workloads, MIN-cod has a total cost 20% larger than the lower bound of OPT. In all cases, using the bound reported, we can guarantee that MIN-cod's performance is at most four times of optimal.

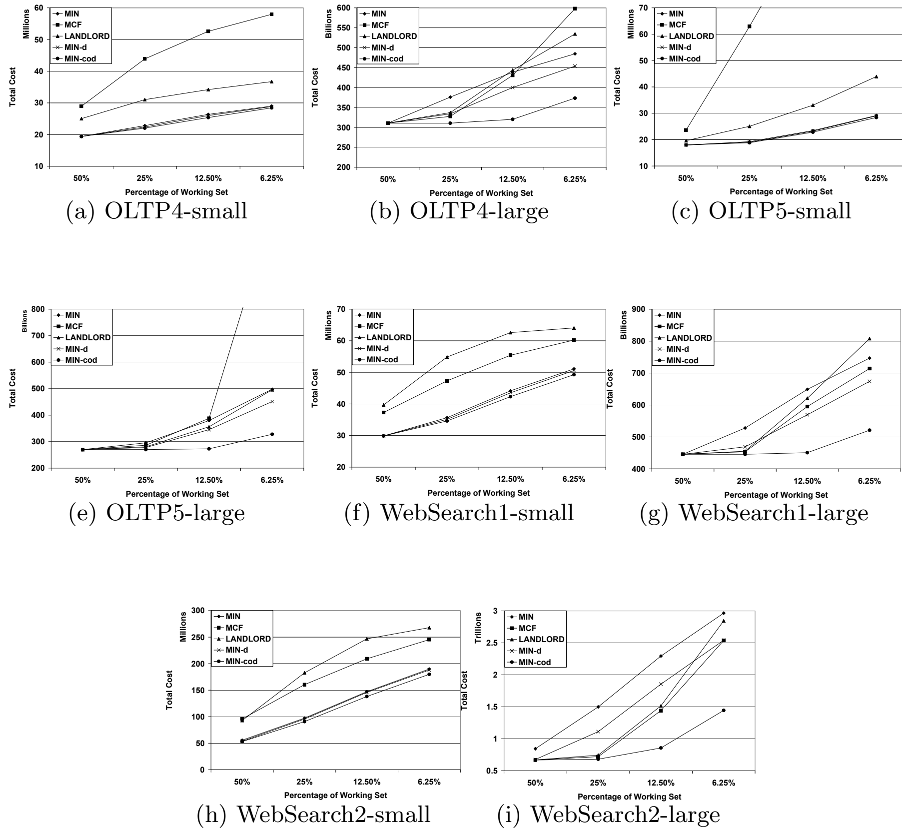
The above results show that the Cod heuristic works well with all the workloads tested. Since MIN-cod balances the cost of evicted blocks and the number of misses the replacement decision can cause, it is effective for a cost-aware replacement algorithm.

On-line Algorithm Results Figure 7 compares HD-cod with online algorithms: LRU, LFU, MRU, MCF, and Landlord. Overall, HD-cod performs very close to the best algorithm in each test scenario. Specifically, for the OLTP workload, HD-cod performs comparably as Landlord, yet reduces up to 68% of the total cost of MCF; for the WebSearch workload, HD-cod performs similarly as MCF, yet reduces up to 15% of the total cost of Landlord.

The above results show that for on-line cost-aware replacement algorithms where the forward distance of a block is not known, the balance between the two metrics (cost and forward distance) needs to be conducted adaptively based on the workloads. Since the Landlord algorithm only considers future accesses as with LRU-like locality, it does not perform well for workloads with the non-LRU-like locality. Since HD-cod detects workload characteristics by tracing the hit density, it adapts itself to behave more like MCF when temporal locality is weak and to behave more like Landlord when temporal locality is strong. Therefore, HD-cod performs close to the best algorithm in all test cases.

5 Related Work

Previous work on cost-aware cache replacement includes both theoretical results and system studies. Young [11] studied the weighted caching problem and proposed an on-line algorithm – GreedyDual. Cao *et al.* [10] studied WWW-proxy caching and proposed GreedyDual-size to incorporate file size into replacement



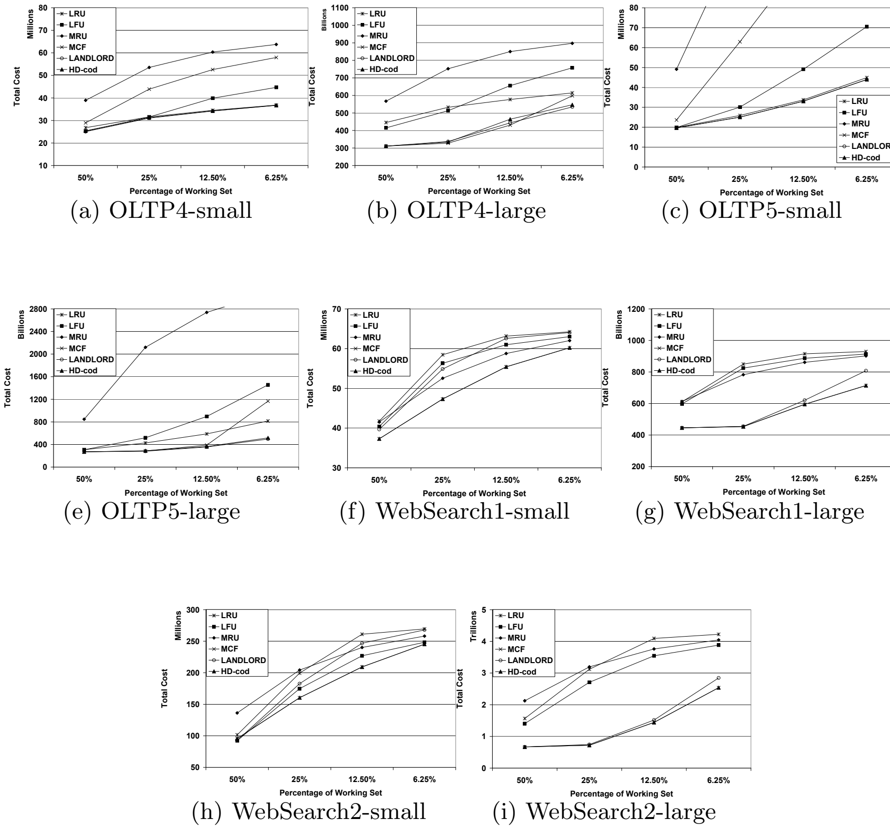
(a) OLTP4-small (b) OLTP4-large (c) OLTP5-small

(e) OLTP5-large (f) WebSearch1-small (g) WebSearch1-large

(h) WebSearch2-small (i) WebSearch2-large

Fig. 6. Comparison of off-line cost aware algorithms using different cache sizes

decision. Landlord [6] is a generalization of both GreedyDual and GreedyDual-size. Chrobak *et al.* [7] proposed on-line algorithms: Rotate and Balance. All these algorithms are proved to be k-competitive, thus theoretically optimal. GreedyDual-size has also been demonstrated to be effective experimentally using simulation experiments on Web proxy traces. Although competitive analysis provides a bound for approximation algorithms, the bound is usually too loose to be attractive. For example, k in the above results is the cache size in blocks, which is at the magnitude of millions with current technology; and it keeps increasing as the technology evolves. Although experimental results are provided to demonstrate the effective of GreedyDual-size [3], it focuses on the variance of document size rather than the access cost of uniform-sized block in storage system. Therefore, they compare with popular replacement algorithms such as LRU, LFU and other size-aware algorithms. Forney *et al.* [9] studied partition-based cache management scheme for heterogeneous storage devices and proposed



(a) OLTP4-small (b) OLTP4-large (c) OLTP5-small

(e) OLTP5-large (f) WebSearch1-small (g) WebSearch1-large

(h) WebSearch2-small (i) WebSearch2-large

Fig. 7. Comparison of on-line cost aware algorithms using different cache sizes

to use equal device wait time as a metric for dynamic cache allocation, which is orthogonal to our studies.

Compared with previous work, our work demonstrates that the upper-bound of performance degradation of any replacement algorithm over OPT can be determined by keeping track of the costs of evicted blocks, which is more practical and meaningful than the k-competitive result. We also point out that the key design issue for efficient cost-aware replacement algorithms is to make an effective trade-off between victim blocks' cost and the total number of evictions. Our Cod heuristic which is based on the principle is demonstrated to outperform previous cost-aware and cost-unaware algorithms in real storage server traces simulations.

6 Conclusions

We have proposed both off-line and on-line algorithms that have performance comparable to the optimal replacement algorithm (OPT) measured by the total cost, yet are much faster to run in practice. The algorithms' design is guided by the following findings of ours. The performance of any replacement algorithm is deviated from OPT by at most the cost of evicted blocks, such that the key to design cost-aware replacement algorithm is to trade-off the number of evictions and the cost of victim blocks. Our work provides analytical bases for buffer cache management in distributed storage systems. We will further understand the implications of our study in our experimental system research.

References

1. Jiang, S., Zhang, X.: Making LRU friendly to weak locality workloads: A novel replacement algorithm to improve buffer cache performance. IEEE Trans. on Comp. **54**(8) (2005) 939–952
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press (2001)
3. Goldberg, A.V.: An efficient implementation of a scaling minimum-cost flow algorithm. J. Algorithms **22**(1) (1997) 1–29
4. Belady, L.: A study of replacement algorithms for virtual storage computers. IBM Sys. J. **5** (1966) 78–101
5. Storage Performance Council: SPC I/O Traces. Available at http://www.storageperformance.org.
6. Young, N.E.: On-line file caching. In: Proc. 9th Annu. ACM-SIAM sympos. Discrete algorithms. (1998) 82–86
7. Chrobak, M., Karloff, H., Payne, T., Vishwanathan, S.: New results on server problems. SIAM J. Discret. Math. **4**(2) (1991) 172–181
8. Patterson, D.A., Hennessy, J.L.: Computer architecture: a quantitative approach. Morgan Kaufmann Publishers Inc. (1990)
9. Forney, B., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Storage-aware caching: Revisiting caching for heterogeneous storage systems. In: Proc. 1st USENIX Sympos. File and Storage Tech. (2002) 61–74
10. Cao, P., Irani, S.: Cost-aware WWW proxy caching algorithms. In: Proc. 1997 Usenix Sympos. on Internet Tech. Sys. (1997)
11. Young, N.E.: The k-server dual and loose competitiveness for paging. Algorithmica **11**(6) (1994) 525–541

7 Appendix

Cost Distribution of Traces

Table 1. Large Cost Distribution of Different Traces

Cost	OLTP1	OLTP2	OLTP3	OLTP4	OLTP5	WebSearch1	WebSearch2
30	32.2%	0.4%	12.3%	16.0%	14.8%	15.0%	12.7%
800	2.3%	0.0%	0.0%	5.6%	0.8%	2.1%	4.8%
1000	4.1%	0.2%	17.3%	11.3%	10.4%	5.9%	6.3%
70700	2.9%	0.0%	0.0%	7.3%	1.4%	2.1%	4.7%
1044	4.8%	0.0%	21.1%	7.8%	9.5%	14.8%	8.1%
5000000	4.9%	0.0%	0.0%	2.5%	1.0%	2.1%	4.7%
80054	5.6%	0.2%	10.7%	3.3%	13.3%	5.9%	6.5%
623	4.9%	25.0%	0.0%	3.0%	1.0%	2.1%	4.7%
2345879	3.9%	1.3%	9.3%	11.9%	18.5%	14.8%	7.9%
17456	3.0%	14.1%	0.0%	3.8%	0.9%	2.2%	4.8%
8567	2.3%	6.9%	12.2%	4.3%	6.7%	5.9%	6.4%
37859	4.1%	3.0%	0.0%	3.0%	0.7%	2.1%	4.9%
77392	5.0%	4.8%	7.2%	9.9%	9.7%	14.9%	8.0%
351238	8.0%	28.3%	0.0%	3.1%	0.8%	2.2%	4.7%
6345982	6.0%	15.2%	9.9%	3.9%	9.6%	5.8%	6.2%
2345907	6.1%	0.5%	0.0%	3.4%	1.0%	2.1%	4.8%
Total:	6489	10136	9693	1235633	3046112	1055236	4578819

Table 2. Small Cost Distribution of Different Traces

Cost	OLTP1	OLTP2	OLTP3	OLTP4	OLTP5	WebSearch1	WebSearch2
534	36.1%	1.6%	5.7%	31.4%	6.1%	11.9%	19.7%
87	0.0%	0.0%	7.3%	1.2%	7.7%	4.4%	3.3%
112	9.7%	28.8%	10.2%	3.2%	8.9%	6.5%	5.9%
77	0.0%	0.0%	8.1%	1.5%	5.9%	4.5%	3.4%
104	8.0%	8.5%	6.6%	4.3%	5.2%	8.4%	9.9%
56	0.0%	0.0%	7.7%	1.9%	5.9%	4.7%	3.3%
67	9.5%	14.6%	2.4%	5.5%	7.0%	6.5%	6.0%
23	0.0%	0.0%	4.7%	2.6%	4.0%	4.7%	3.3%
58	9.8%	4.8%	6.6%	28.6%	8.9%	8.4%	9.9%
46	0.0%	0.0%	4.2%	1.5%	7.1%	4.5%	3.4%
93	6.8%	25.0%	9.5%	4.2%	4.4%	6.6%	5.9%
39	0.0%	0.0%	4.9%	1.4%	3.1%	4.5%	3.3%
47	10.0%	14.0%	2.7%	4.9%	9.7%	8.7%	10.1%
38	0.0%	0.0%	6.5%	1.9%	7.7%	4.6%	3.4%
62	10.2%	2.6%	6.2%	4.5%	4.3%	6.6%	5.9%
37	0.0%	0.0%	6.8%	1.4%	3.9%	4.5%	3.3%
Total:	6489	10136	9693	1235633	3046112	1055236	4578819