CAMP: A Cost Adaptive Multi-Queue Eviction Policy for Key-Value Stores *†

Shahram Ghandeharizadeh, Sandy Irani, Jenny Lam, Jason Yap, Hieu Nguyen

Database Laboratory Technical Report 2015-05

Computer Science Department, USC

Los Angeles, California 90089-0781

May 20, 2015

Abstract

Cost Adaptive Multi-queue eviction Policy (CAMP) is an algorithm for a general purpose key-value store (KVS) that manages key-value pairs computed by applications with different access patterns, key-value sizes, and varying costs for each key-value pair. CAMP is an approximation of the Greedy Dual Size (GDS) algorithm that can be implemented as efficiently as LRU. In particular, CAMP's eviction policies are as effective as those of GDS but require only a small fraction of the updates to an internal data structure in order to make those decisions. Similar to an implementation of LRU using queues, it adapts to changing workload patterns based on the history of requests for different key-value pairs. It is superior to LRU because it considers both the size and cost of key-value pairs to maximize the utility of the available memory across competing applications. We compare CAMP with both LRU and an alternative that requires human intervention to partition memory into pools and assign grouping of key-value pairs to different pools. The results demonstrate CAMP is as fast as LRU while outperforming both LRU and the pooled alternative. We also present results from an implementation of CAMP using Twitter's version of memcached.

1 Introduction

Applications with a high read-to-write ratio augment their persistent infrastructure with an in-memory key-value store (KVS) to enhance performance. An example is memcached in use by popular Internet destinations such as Facebook, Twitter, and Wikipedia. Using a general purpose caching layer requires workloads to share infrastructure despite different access patterns, key-value sizes, and time required to compute a key-value pair [24]. An algorithm that considers only one factor may cause different application workloads to impact one another negatively, decreasing the overall effectiveness of the caching layer.

*Sandy Irani and Jenny Lam are with the University of California, Irvine. Their research is supported in part by the NSF grant CCF-0916181.

†A shorter version of this paper appeared in the Proceedings of the ACM/IFIP/USENIX Middleware Conference, Bordeaux, France, December 2014.

As an example, consider two different applications of a social networking site: one shows the profile of members while a second determines the displayed advertisements. There may exist millions of key-value pairs corresponding to different member profiles, each computed using a simple database look-up that executed in a few milliseconds. The second application may consist of thousands of key-value pairs computed using a machine-learning algorithm that processed Terabytes of data and required hours of execution. This processing time is one definition of the *cost* of a key-value pair. With a limited memory size and a high frequency of access for member profile key-value pairs, a simple algorithm that manages memory using recency of references (LRU) may evict most of the key-value pairs of the second application, increasing the incurred cost.

In general, reducing the incurred cost translates into a faster system that processes a larger number of requests per unit of time and may provide a better quality of service. The latter is due to availability of data (*e.g.*, cache hit for a key-value computed using the machine learning algorithm) that enables the application to provide a user with more relevant content than content selected randomly. A possible approach is for a human expert to partition the available memory into disjoint pools with each pool managed using LRU. Next, the expert groups key-value pairs with similar costs together and assigns each group to a different pool [21]. With our example, the expert would construct two pools. One for the key-value pairs corresponding to members profiles and a second corresponding to advertisements. The primary limitation[1] of this approach is that it requires a human familiar with the different classes of applications to identify the pools, construct grouping of key-value pairs, and assign each group to a pool. Over time, the service provider may either introduce a new application or discontinue an existing one. This means the human expert must again become involved to identify the pool for the key-value pairs of the new application and possibly rebalance memory across the pools once an application is discontinued.

This paper introduces a novel caching method called Cost Adaptive Multi-queue eviction Policy (CAMP), that manages the available memory without partitioning it. CAMP is an approximation of the Greedy Dual Size (GDS) algorithm [4] that processes cache hits and misses more efficiently using queues. Hence, it is significantly faster than GDS and as fast as LRU. It is novel and different from LRU in that it constructs multiple LRU queues dynamically based on the size and cost of key-value pairs. The number of constructed LRU queues depends on the distribution of costs and sizes of the key-value pairs. CAMP manages these LRU queues without partitioning memory. Thus, there is no need for human involvement to construct groups of key-value pairs, dictate assignment of groups to the pools, or configure and adjust the memory pool characteristics. CAMP is robust enough to prevent an aged expensive key-value pair from occupying memory indefinitely. Such a key-value pair is evicted by CAMP as competing applications issue more requests.

CAMP is parameterized by a variable that controls its precision. At the highest precision, CAMP's eviction decisions are essentially equivalent to those made by GDS. Our empirical results show that CAMP does not suffer any degradation in the quality of its eviction decisions at lower precisions. Moreover, it is able to make those decisions much more efficiently than GDS. GDS requires an internal priority queue to determine a key-value pair to evict

[1]Partitioning is known to reduce the utilization of resources by resulting in formation of hot spots and bottlenecks. One may address this limitation by over-provisioning resources.

from the cache. The time to maintain its data structures consistent in a thread-safe manner is expensive because it requires synchronization primitives [15] with multiple threads performing caching decisions. Moreover, CAMP performs a significantly fewer updates of its internal data structures than GDS, reducing the number of times it executes the thread-safe software dramatically. This is specially true when CAMP is extended with an admission control technique.

The rest of this paper is organized as follows. Section 2 starts with a description of GDS to motivate CAMP and details its design decisions. Section 3 presents a simulation study of CAMP and compares it with LRU and the pooled approach that partitions resources, demonstrating its superiority. Section 4 describes an implementation of CAMP using a variant of Twemcache and compares this implementation with the original that uses LRU. Obtained results demonstrate that CAMP is as fast as LRU and provides superior performance as it considers, in addition to recency of requests, the size and the cost of the key-value pairs. Section 5 describes related work. Section 6 provides brief words of conclusions and future research directions.

2 CAMP

The algorithm Greedy Dual Size (GDS) was developed in the context of replacement policies for web proxy caches. It captures many of the benefits of LRU and considers the fact that data objects on the web have varying sizes and incur varying time delays to retrieve depending on their location and network traffic [4]. The same principles apply in the context of maintaining the identity of key-value pairs occupying the memory of a KVS, although the cost of an object in the KVS setting may denote computation time (or some other quantity) instead of retrieval time. Even though the algorithm is applicable to a wide variety of settings, we adopt the terminology used for cache augmented database management systems [11]. The GDS algorithm is based on an algorithm called Greedy Dual, developed by Neal Young [26], that handles objects of varying cost but uniform size. GDS assigns a value $H(p)$ to each key-value pair p in the KVS. $H(p)$ is computed from a global parameter, L, as well as from size(p), the size of p and cost(p), the cost of p. The value of $H(p)$ approximates the benefit of having that key-value in the KVS. When there is insufficient memory to accommodate an incoming key-value pair, the algorithm continually evicts the key-value pair with the lowest value until there is room in the KVS to store the incoming key-value pair.

The pseudocode for GDS is given in Algorithm 1. The following proposition is useful in understanding how the algorithm works:

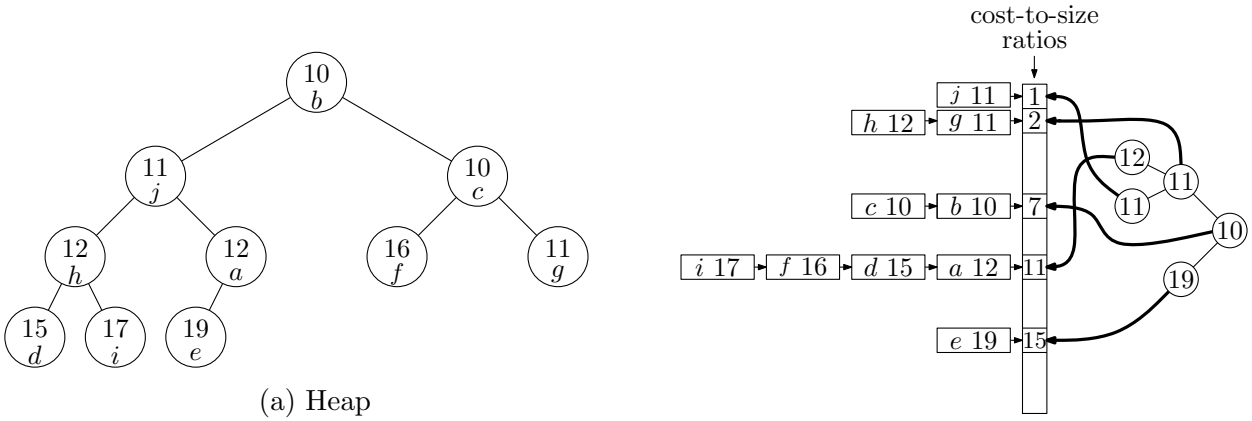
Proposition 1.

1. *L is non-decreasing in time.*
2. *If p is in the KVS, then $L \leq H(p) \leq L + \text{cost}(p)/\text{size}(p)$.*

Proof: We prove the claim by induction on the number of requests. At the beginning of the request sequence, there are no key-value pairs in the KVS, so both claims are true. In lines 2 and 6, the value of L is set to be the smallest H-value among all the key-value pairs in the KVS. Since, by induction, for every key-value pair p in the cache, $L \leq H(p)$, L can only increase or stay the same after the change. Lines 2 and lines 6 are the only time that L changes, so the first claim must hold.

Algorithm 1 GreedyDualSize.

Initialize $L \leftarrow 0$.
Process each request for key-value pairs in turn.
The current request is for key-value pair p:
(1) **if** p is already in memory (denoted by M),
(2) $\quad L \leftarrow \min_{q \in M \setminus \{p\}} H(q)$.
(3) **else**,
(4) $\quad$ **while** there is not enough room in memory for p,
(5) $\quad\quad$ Evict the q with the smallest $H(q)$.
(6) $\quad\quad L \leftarrow \min_{q \in M} H(q)$.
(7) $\quad$ Bring p into memory.
(8) $H(p) \leftarrow L + \text{cost}(p)/\text{size}(p)$.



(a) Heap

(b) CAMP's priority queue

Figure 1: A heap used in a straightforward manner (1a) contains many more nodes than the heap in CAMP's LRU-heap hybrid (1b) when there are only a few distinct cost-to-size ratios.

For claim 2, L will not exceed $H(p)$ for any p in the cache because its new value (assigned in line 2 or 6) is the smallest $H(p)$ among all the key-value pairs in the cache. Any eviction performed in line 5 only makes it easier to satisfy claim 2. Finally, when a key-value pair is brought into the KVS, it's H value is set to $L + cost(p)/size(p)$ so the claim 2 still holds by definition. ■

Consider a key-value pair p in the KVS. As more key-value pairs are referenced (*i.e.*, p becomes requested less recently), the value of L increases and $H(p)$ becomes smaller relative to the other key-value pairs in the KVS. If p is requested while it is in the KVS, then $H(p)$ increases to $L + \text{cost}(p)/\text{size}(p)$ which has the effect of delaying its eviction. All other things being equal, if a key-value pair has a *cost-to-size ratio*, *i.e.*, the quantity $\text{cost}(p)/\text{size}(p)$, that is c times that of another key-value pair, it will reside in the KVS roughly c times longer. GDS exhibits good performance under a variety of different load conditions because it considers both varying costs and sizes without resorting to *ad hoc* categorization of key-value pairs.

An implementation of GDS must maintain a data structure to identify and delete the key-value pair with the minimum priority efficiently. Typically, key-value pairs are maintained in a data structure that can retrieve and delete the key-value pair with the minimum priority. Normally this is accomplished by an implementation of a priority queue like a Fibonacci heap [5]. The worst-case performance of any priority queue is a $\log n$ cost per operation, where n is the number of key-value pairs in the priority queue. For extremely large KVSs, such as those in use by Facebook and Twitter, overhead that scales even logarithmically as a function of the number of key-value pairs in the KVS results in a significant cost that could potentially be avoided.

The starting point for this work is the observation that the H value assigned to each key-value pair in the KVS is merely an approximation for the value of storing that key-value pair in the KVS in the absence of information about when that key-value pair will be requested next in the future relative to the other key-value pairs in the KVS. Insisting that the priority queue evict the key-value pair with the absolute minimum value is likely overkill. It seems reasonable that a similar KVS hit performance can be achieved if we only require that the data structure evict a key-value pair whose priority is only approximately smallest. An approximate priority queue could potentially be more efficient than one that is required to return the true minimum.

With GDS, the *priority* or H value (the two terms will be used interchangeably) of every key-value pair in the KVS is the sum of two values: the global non-decreasing variable L and the cost-to-size ratio of the key-value pair. CAMP rounds the priority for every key-value pair by rounding the cost-to-size ratio before adding it to L. The rounding scheme results in a smaller set of possible cost-to-size values for key-value pairs stored in the KVS. CAMP takes advantage of the rounding by grouping the key-value pairs in its data structure according to the cost-to-size ratio instead of by priority value. The eviction decisions between CAMP and GDS differ slightly for two reasons. First, CAMP rounds the cost-to-size ratio in determining the H value of a key-value pair. Second, in evicting the key-value pair with smallest priority, CAMP breaks ties according to LRU, whereas GDS breaks ties arbitrarily.

Figure 1 shows the storage schemes for GDS and CAMP, with each circle denoting the H value of a key-value pair. Figure 1a shows a typical priority-queue-based implementation of GDS in which a set of key-value pairs are stored in a heap[2] based on their priority value. Figure 1b shows CAMP's data structure in which key-value pairs are grouped into queues according to their cost-to-size ratio. Key-value pairs in a queue are ordered according to their priority which is their H value. CAMP maintains a heap containing the priority of the data item at the head of every queue. Thus, to identify a candidate key-value pair to evict from the KVS, CAMP locates the key-value pair with the smallest priority among the heads of each of the queues.

CAMP's implementation is efficient based on the following key observation. If the key-value pairs within each queue are stored according to LRU, then the key-value pairs are automatically ordered according to their priority. For this reason, the queues maintained by CAMP are termed LRU queues. To understand why this observation holds, recall that all the items within an LRU queue have the same cost-to-size ratio. Furthermore, the H

[2]A heap is a tree-based implementation of a priority queue which maintains the property that the priority of any node in the tree is at most the priority of its children.

value of a key-value pair is the value of L at the time of its last request plus its cost-to-size ratio. Since L increases over time, a key-value pair that is requested earlier on will have a smaller H value and appear towards the front of the queue, whereas a key-value pair that is referenced more recently will have a larger H value and appear towards the end of the queue. In particular, the first key-value pair in each queue has the smallest priority.

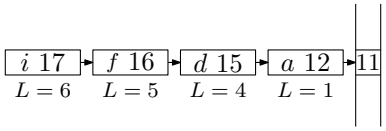


Figure 2: An LRU queue within CAMP.

As an example, consider the LRU queue in Figure 2 which is one of the five queues shown in Figure 1b. It points to the array value 11, containing all the key-value pairs that have cost-to-size ratio 11. Each node shows the key-value pair's H value, and the shown L value is the value of L at the time of its last request. Since L increases over time and key-value pairs are inserted at the tail of the queue, key-value pairs are ordered by the value of L at the time of the request. Since all these key-value pairs have similar cost-to-size values, they are also ordered by their H value. Specifically, a is the least recently requested key-value pair in its queue.

With CAMP, the complexity of processing a KVS hit for a referenced key-value pair is the complexity to update the LRU queue ($O(1)$) plus the complexity to update the heap. The worst case for the latter is logarithmic in the number of non-empty queues instead of the number of key-value pairs in the KVS since the nodes in the heap tree structure of CAMP identify LRU queues, see Figure 1b. With our implementation of CAMP, we chose to use an 8-ary implicit heap as suggested by the recent study [16] on priority queues. Here, 8-ary means with branching factor at most 8. Moreover, a heap is implicit if it uses the usual array implementation rather than with pointers.

To illustrate the processing of a KVS hit using the same running example of Figure 1, consider a new reference for g. The KVS locates g using a hash table (Figure 3a), moves it to the end of its LRU queue (see Figure 3b), and updates its value to $10 + 2 = 12$ where 10 is the minimum priority (L value) and 2 is the cost-to-size ratio of g. Now, the new head of the queue has priority 12. CAMP updates the value of the heap node pointing to this queue (Figure 3c), causing the heap to be updated as shown in Figure 3d.

One way to improve performance is by limiting the number of LRU queues. We can do so by assigning key-value pairs with "similar" cost-to-size values to the same queue. Similarity in this context has a specific meaning, in that values that have different orders of magnitude should remain distinct. Therefore, a rounding scheme that simply truncates a fixed number of bits will not work. Instead, CAMP uses the integer rounding scheme described in [18]. Given a number x, let b be the order of its highest non-zero bit. To round x to precision p, zero out the $b-p$ lower order bits or, in other words, preserve only the p most significant bits starting with b. If $b \leq p$, then x is not rounded. Table 1 illustrates the difference between these rounding schemes with examples. With regular rounding, too much information is kept for large values and too little information is kept for small values. Since we don't know the range of values *a priori*, we don't know how to select p to balance the two extremes.

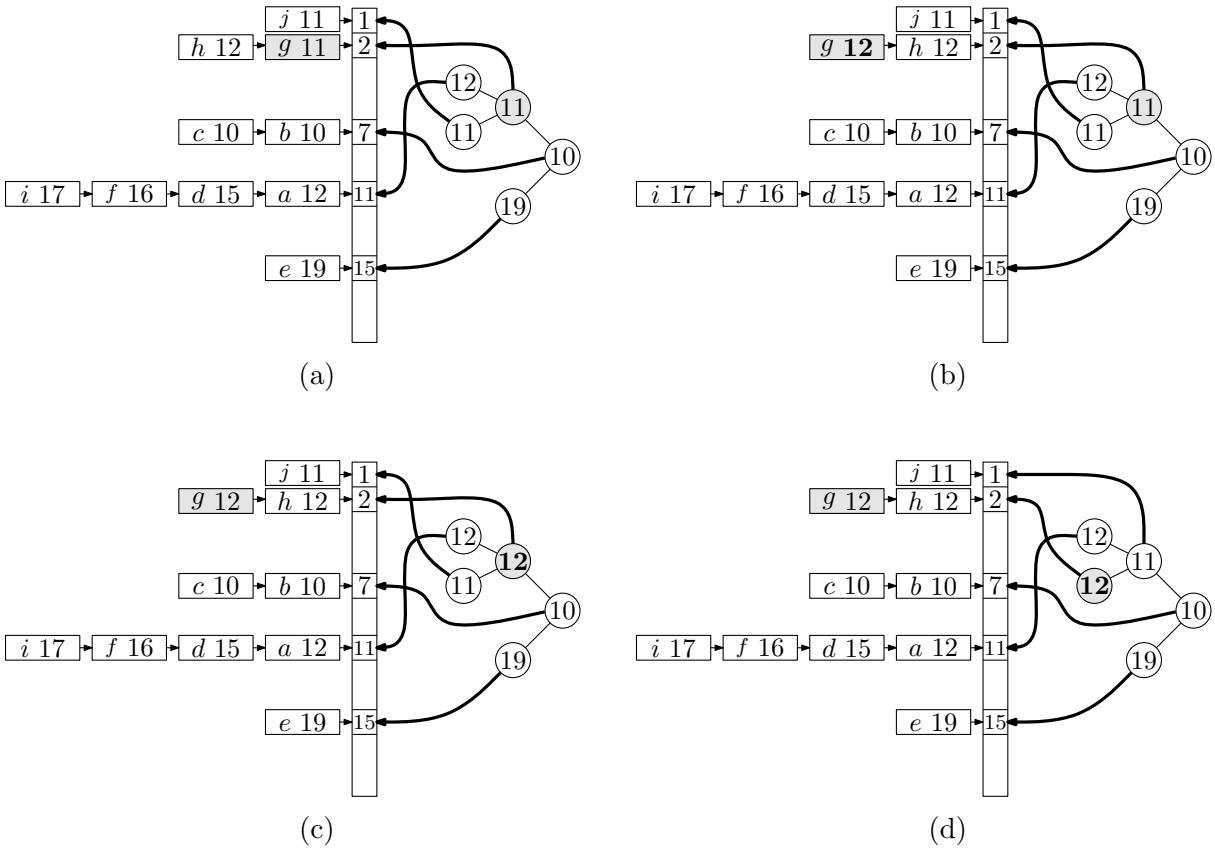


Figure 3: CAMP update on a KVS hit. If g is requested (3a), it is moved to the back of its queue (3b), and the heap is updated accordingly (3c and 3d).

Therefore, we prefer the amount of rounding to be proportional to the size of the value itself (right column).

regular rounding	CAMP's rounding
10110**1011** → 10110**0000**	1011**01011** → 1011**00000**
00101**0011** → 00101**0000**	001010**011** → 00101**0000**
00000**1010** → 00000**0000**	00000<u>1010</u> → 00000**1010**
00000**0111** → 00000**0000**	00000<u>111</u> → 00000**111**

Table 1: Rounding with (binary) precision 4

The following proposition gives a bound on the number of distinct rounded values for the cost-to-size ratio which in turn is an upper bound on the number queues maintained by CAMP:

Proposition 2. *If the original values for the cost-to-size ratio are integers in the range* $1, \ldots, U$*, then the number of distinct rounded values is at most* $(\lceil \log_2(U+1) \rceil - p + 1)2^p$ *where* p *is the selected precision.*

Proof: Let x denote the value to be rounded. Since x is in the range 1 through U, the binary representation of x uses at most $\lceil \log_2(U+1) \rceil$ bits. Bit locations are numbered 1 through $\lceil \log_2(U+1) \rceil$ with 1 being the lowest order bit. Let b be maximum of p and the location of the highest order non-zero bit in the binary representation of x. The scheme zeroes out all bits except those in locations $b, b-1, \ldots, b-p+1$. There are at most $\lceil \log_2(U+1) \rceil - p + 1$ possible values for b. For each value of b, there are 2^p possible rounded values encoded in bits $b, b-1, .., b-p+1$. Thus, the total number of distinct rounded values is $(\lceil \log_2(U+1) \rceil - p + 1)2^p$. ■

The competitive ratio of GDS is k which means that on every sequence of requests, the overall cost of GDS is within a factor of k of the optimal algorithm that knows the entire request sequence in advance. The proposition below shows that CAMP with precision p approximates the behavior of GDS by a factor of $1+\epsilon$ where $\epsilon = 2^{-p+1}$ in the sense that CAMP obtains a competitive ratio of $(1+\epsilon)k$. Thus, for sufficiently small ϵ, the data structure would always evict the key-value pair with the true minimum priority.

Proposition 3. *The competitive ratio of CAMP is* $(1+\epsilon)k$*, where* $\epsilon = 2^{-p+1}$*.*

Proof: Consider an unrounded integer x and denote its rounded value by $\bar{x}$. We know that $\bar{x} \leq x$ because rounding only involves changing 1's to 0's. Let b be the location of the highest order bit in x. Then $\bar{x} \geq 2^{b-1}$. Bits 1 through $b-p$ are set to zero when x is rounded. In the worst case, the cleared bits are all 1. The amount that is subtracted from x to get $\bar{x}$ is at most 2^{b-p}. Therefore, $(x-\bar{x})/\bar{x} \leq 2^{b-p}/2^{b-1} = 2^{-p+1}$ and $x \leq (1+\epsilon)\bar{x}$, where $\epsilon = 2^{-p+1}$.

Now let σ be a sequence of requests and let $\bar{\sigma}$ be the same request sequence but with rounded cost-to-size ratios. Define $CAMP(\sigma)$ to be the total cost of $CAMP$ on input σ and let $OPT(\sigma)$ be the total cost of the optimal offline algorithm on input σ. CAMP makes the same eviction decisions on σ as it does on $\bar{\sigma}$ because it rounds the cost-to-size ratios in σ. However, it pays potentially a factor of $(1+\epsilon)$ more on each cache miss. Therefore $CAMP(\sigma)/(1+\epsilon) \leq CAMP(\bar{\sigma})$. CAMP makes the same decisions as GDS on $\bar{\sigma}$ because the values are already rounded, so $CAMP(\bar{\sigma}) = GDS(\bar{\sigma})$. Since we know that GDS is k-competitive, $GDS(\bar{\sigma}) \leq kOPT(\bar{\sigma})$. Finally, OPT will have at least as large an overall cost

on σ as it will on $\bar{\sigma}$ because all the cost-to-size ratios are at least as large which means that $OPT(\bar{\sigma}) \leq kOPT(\sigma)$. Putting it all together, we get

$$\begin{aligned} \frac{CAMP(\sigma)}{1+\epsilon} &\leq CAMP(\bar{\sigma}) = GDS(\bar{\sigma}) \\ &\leq k \cdot OPT(\bar{\sigma}) \leq k \cdot OPT(\sigma), \end{aligned}$$

and CAMP is $(1+\epsilon)k$-competitive. ■

To use the integer rounding scheme we have described, we must first convert the cost-to-size ratio from a fraction to an integer. We cannot simply round since we may lose information regarding the relative order of magnitude among values that are less than 1. We can solve this problem by first dividing the cost-to-size ratio by a lower bound estimate on the smallest cost-to-size ratio that can ever occur. Then we perform the actual rounding to the nearest integer. The cost of each key-value pair is a non-negative integer, so 1 divided by the maximum size of any key-value pair can serve as a lower bound for the cost-to-size ratio. Thus, we are effectively multiplying each cost-to-size ratio by the size of the largest key-value pair. Although we do not know the maximum size of a key-value pair *a priori*, we can determine it adaptively. (The next paragraph describes why we do not simply use a large number such as the cache size.) A variable is used to hold the current maximum size observed so far. The variable is updated as soon as a referenced key-value pair is larger than the current maximum. For the sake of efficiency, we do not update the rounded priorities of all the key-value pairs in the KVS when a new lower bound on the cost-to-size ratio is determined. However, the new value is used for all future rounding.

The rounding scheme makes no a priori assumptions as to the values of the cost-to-size ratios other than assuming that the ratio of the smallest to largest cost/size ratio is bounded by U. Converting all values to an integer is just a mathematically convenient way of expressing that assumption. The goal is to use the range 1 to U as effectively as possible in expressing the range of cost-to-size ratios. The larger the value of U, the more space that must be set aside for potential LRU queues. The conversion from fractional values to integers is achieved by multiplying all values by a fixed multiplier and rounding to the nearest integer. Selecting a large number for the multiplier would result in large rounded values and would require a large upper bound U. This explains why we do not simply use the cache size for the multiplier.

In short, CAMP computes the H value of a key-value pair in three steps. As the initial step, it converts the cost-to-size ratio to an integer. It then rounds the result by the pre-specified precision to an approximate value c. Finally, it assigns the key-value pair to the LRU queue associated with the value c. The key-value pair is assigned an H value of $c + L$, where L is the offset parameter used by GDS.

Figure 4 compares the number of visited heap nodes in a heap-based implementation of GDS and in CAMP when run using the trace-driven simulation of Section 3. This quantity is an indication of the amount of runtime overhead of each implementation: in the case of GDS, this is the number of nodes that are visited when the heap is updated due to an insertion or deletion. In the case of CAMP, insertions and deletions from the queue comprise a constant time update to an LRU queue as well as the occasional update to the heap when the head of an LRU queue changes.

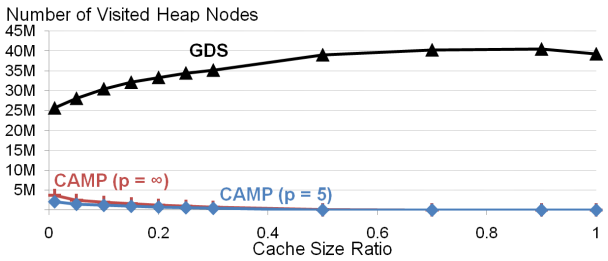


Figure 4: Number of visited heap nodes as a function of the cache size ratio.

There are two contributing factors to CAMP's significantly smaller number of node visits. First, the number of nodes in GDS's heap is equal to the number of key-value pairs in cache whereas the number of nodes in CAMP's heap is equal to the number of non-empty LRU queues, which is very small as noted in Figure 5b. Since the number of node visits required by a heap update grows logarithmically with the number of nodes in the heap, GDS's heap updates can take longer than CAMP's. The second contributing factor is that GDS makes more heap updates than CAMP does. In particular, GDS updates its heap every time the priority of a key-value pair is updated. On the other hand, CAMP only does so whenever the priority value of the head node of an LRU queue changes, or when an LRU queue is created or deleted.

Figure 4 shows the number of visited nodes by GDS increases as a function of the memory size. This trend is reversed with CAMP. The GDS curve increases because the number of nodes in the heap is equal to the number of items in the KVS and there are many more data items with a larger memory size. In contrast, the CAMP curve decreases because the number of heap nodes, which is equal to the number of non-empty LRU queues, remains constant as a function of cache size. But since more items can be stored when the cache size increases, there are fewer updates to the cache. Hence, the decreasing curve.

3 Evaluation

We used a social networking benchmark named BG [1, 2, 7, 8] to generate traces of key-value references from a cache augmented database management system [11]. BG emulates members of a social networking site viewing one another's profile, listing their friends, and other interactive actions. The benchmark is configured to reference keys using a skewed pattern of access with approximately 70% of requests referencing 20% of keys. A trace file consists of approximately 4 million rows. Each row identifies a referenced key-value pair, its size, and cost. Cost is either the time required to compute the key-value pair by issuing queries to the RDBMS or a synthetic value selected from {1, 100, 10K}[3]. With the latter, each key-value pair is assigned one of the three possible values with equal probability. Once a cost is assigned to a key-value pair, it remains in effect for the entire trace.

We implemented a simulator that consists of a KVS and a request generator to read a trace file and issue requests to the KVS. The KVS manages a fixed-size memory that implements

[3]The values {1, 100, 10K} were chosen to simulate widely varying costs between key-value pairs.

either the LRU or the CAMP algorithm. Every time the request generator references a key and the KVS reports a miss for its value, the request generator inserts the missing key-value pair in the KVS. This results in evictions when the size of the incoming key-value pair is larger than the available free space.

The simulator quantifies two key metrics: miss rate and cost-miss ratio. *Miss rate* is the total number of requests that result in a KVS miss divided by the total number of requests in the sequence. *Cost-miss ratio* is obtained by summing the costs for each request that results in a KVS miss divided by the sum of the costs for all the requests. With both, the first request to a particular key-value pair in the trace (called a *cold* request) is not counted because any algorithm will fault on such requests. Since CAMP is tuned to minimize the total cost of serving the requests in the sequence, the cost-miss ratio is the primary metric used to quantify performance. In the following, we report these metrics as a function of either the precision used by the CAMP algorithm or the *cache size ratio*. The latter is the size of the KVS memory divided by the total size of the unique objects in the trace file.

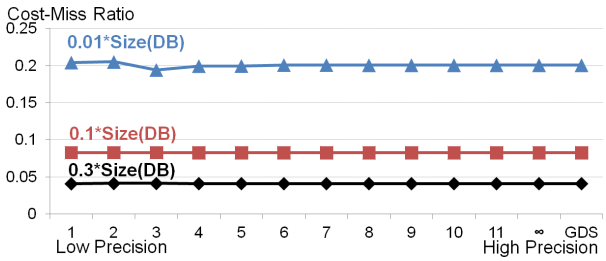
The precision of CAMP is a parameter that can be tuned for performance optimization. Small values of precision result in fewer LRU queues. With larger values of precision, replacement decisions are more finely tuned to differences in the cost-to-size ratio for each key-value pair. The graph in Figure 5a shows the cost-miss ratio for CAMP as a function of the precision value. The three curves show the results for three different cache sizes. For the version labeled ∞, no rounding is done after the initial cost-to-size ratio is rounded to an integer. In other words, this version corresponds to the standard GDS algorithm. Figure 5a shows that there is almost no variation in cost-miss ratios for different precisions. More importantly, there is almost no difference between the cost-miss ratios of CAMP and standard GDS.

Figure 5b shows the number of distinct LRU queues maintained by CAMP as a function of precision. The maximum number of queues possible is the number of distinct possible values for the cost-to-size ratio of the key-value pairs for a particular precision value. However, the KVS may not hold a key-value pair with a particular cost to size value, so at any given point in time, many of the queues are empty. Figure 5b shows the actual number of non-empty queues at the end of the trace. Even for a very low level of precision, CAMP has at least five non-empty queues and outperforms LRU that has only one queue, see Figure 5c.

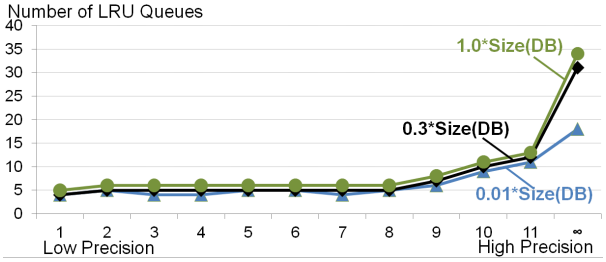
In Figure 5c, Pooled LRU is the partitioned-memory scheme described in [21]. This approach partitions the available memory into distinct pools. Each pool employs LRU to manage its memory. Those key-value pairs with similar costs are grouped together according to their cost. Different groups are assigned to different pools so that cheap and expensive key-value pairs do not compete with one another for the same memory. This is not the same as CAMP, which adjusts the amount of memory used by each queue automatically, as demand fluctuates.

To give Pooled LRU the greatest advantage, the amount of memory for each queue is computed in advance using the frequency of references to the different key-value pairs over the entire trace. We experimented with different ways to partition the memory. In the first way, memory is allocated uniformly between the three queues. In the second way, the fraction of the total available memory assigned to each queue is proportional to the total cost of requests in the trace that belong to a particular pool.

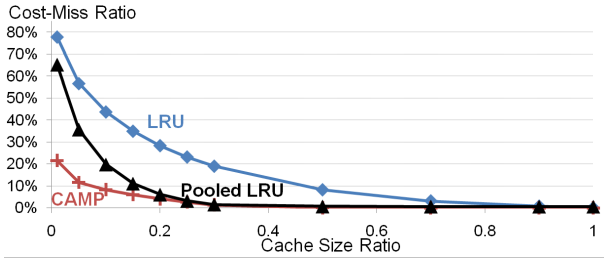
With the BG benchmark-generated trace using synthetic cost values selected from {1,



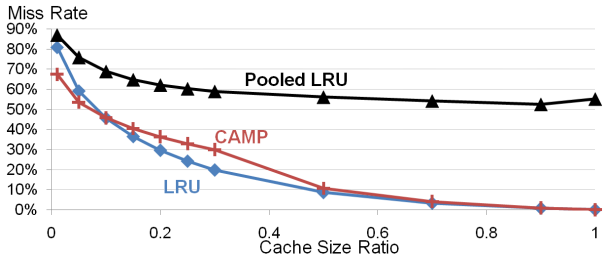
(a) Cost-miss ratio as a function of precision.



(b) Number of LRU queues maintained by CAMP with different precisions.



(c) Cost-miss ratio as a function of the cache size ratio.



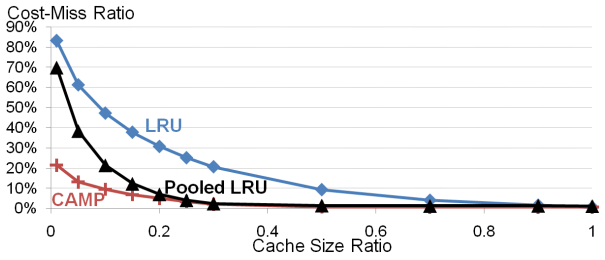
(d) Miss rate as a function of the cache size ratio.

Figure 5: Simulation results with one trace and cost values selected from 1, 100, 10K. With 5c and 5d, the precision of CAMP is set to 5.

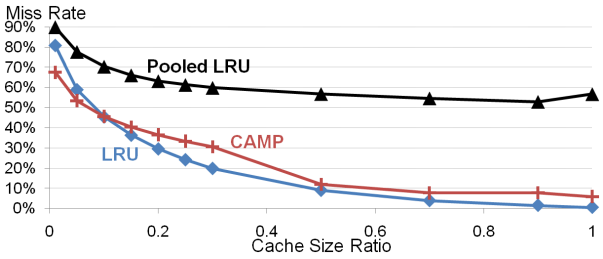
100, 10K}, Pooled LRU constructs three pools. These pools have approximately the same number of key-value pairs, frequency and size, where the frequency of a pool is the number of references made to key-value pairs in that pool and the size is the amount of memory needed to store all key-value pairs in that pool. With a uniform partitioning of memory, Pooled LRU has both a cost-miss ratio and miss rate similar to LRU. This is a consequence of the key-value pairs assigned to each pool having the same frequency and size. The performance is so close, that we only display the cost-miss ratio for LRU in Figure 5c. When memory is partitioned using cost, Pooled LRU improves over LRU's cost-miss ratio. Furthermore, it is able to match CAMP's cost-miss ratio when given a large enough cache size by assigning practically all of it to the most expensive pool. However, this improvement is at the expense of a significantly worse miss rate (see Figure 5d), since, even with a large cache size, the cheapest pool has nearly a 100% miss rate and the second pool has a miss rate of 65%.

3.1 Evolving Access Patterns

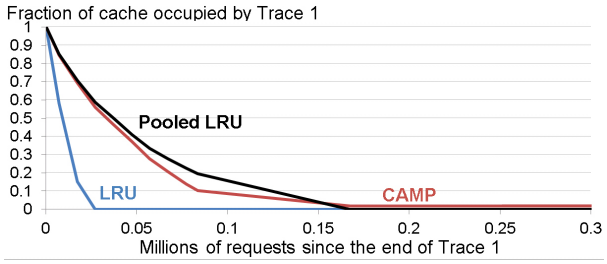
A key feature of CAMP is its ability to adapt to evolving access patterns by evicting those key-value pairs that were hot in some distant past. This includes expensive key-value pairs. Obtained results show CAMP adapts effectively when compared to LRU and Pooled LRU. Moreover, the overall cost-miss ratio and miss rate trends remain the same as the results of Figure 5.



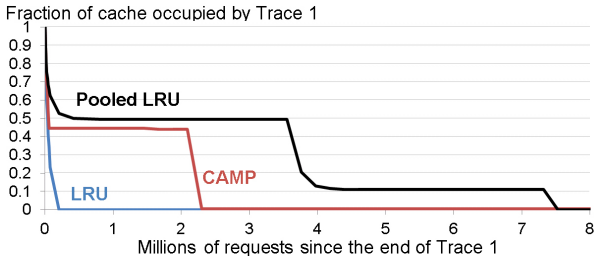
(a) Cost-miss ratio as a function of cache size ratio.



(b) Miss rate as a function of cache size ratio.



(c) Fraction of cache occupied by Trace File 1 items, with cache size ratio set at 0.25.



(d) Fraction of cache occupied by Trace File 1 items, with cache size ratio set at 0.75.

Figure 6: Simulation results with changing access patterns.

In this experiment, we used ten different traces back to back. Each trace file consists of 4 million key-value references. Moreover, requests from different traces are given distinct identification, so any request from a given trace file will never be requested again after that trace. The trace files are identified as Trace File 1, TF1, Trace File 2, TF2, etc. They are adversarial to CAMP because each trace file is generated using a skewed distribution of access as described at the beginning of this section. This means once the simulator switches from TF1 to TF2, none of the objects referenced by TF1 are referenced again. The same holds true for all other trace files, emulating a sudden shift in access patterns where expensive objects that were referenced frequently are never referenced again.

We conducted the above experiment for a variety of cache sizes and observe CAMP adapts across the different trace files to provide a cost-miss ratio and miss rate similar to those observed in the previous section, see Figure 6a and 6b. Figures 6c and 6d show the fraction of KVS memory occupied by the key-values of TF1 for two different KVS memory size ratios, 0.25 and 0.75. These two figures show how well the different techniques adapt to the sudden change in access patterns. The x-axis of these two figures it the number of key-value references issued relative to the start of TF2 in millions of requests. The transition to a different TF is at the 4 million tick mark of the x-axis. The y-axis shows the fraction of KVS memory size occupied by key-value pairs of TF1.

With a small cache size (see Figure 6c with a cache size ratio of 0.25), all three algorithms evict key-value pairs referenced by TF1 quickly. LRU is the quickest, evicting all key-value pairs of TF1 after 21,000 references of TF2. It is followed by Pooled LRU with 131,000

references of TF2. CAMP evicts most of TF1 key-value pairs quicker than Pooled LRU and slower than LRU. It does not evict all TF1 key-value pairs (those with the highest cost-to-size ratio) until 7.7 million references, close to the end of TF3. However, these items occupy less than 2% of the total cache size.

With a larger cache size (see Figure 6d with a cache size ratio of 0.75), both CAMP and Pooled LRU behave in a step function with CAMP evicting a majority of TF1 key-value pairs faster than Pooled LRU. Once again, LRU is quickest as it considers recency of references. Pooled LRU evicts all key-value pairs referenced by TF1 after 7.3 million requests are issued, close to the end of TF3. CAMP maintains a few of the most expensive key-value pairs of TF1 even after 40 million requests are issued. However, these occupy less than 0.6% of the available KVS memory.

Let us analyze the behavior of each algorithm. LRU evicts the key-value pairs requested in TF1 when the total size of newer key-value pairs is greater than the cache size, which occurs before the transition to TF3 regardless of cache size. The jump in eviction time at cache size ratio 1 corresponds to the fact that the key-value pair that causes the total size of requested key-value pairs to exceed the cache size is the first key-value pair requested in TF3.

The sudden eviction of large portions of TF1 key-value pairs by Pooled LRU at large cache sizes (see Figure 6c) correspond to the introduction of new key-value pairs at the beginning of TF3 and TF4, which occur at around the 4 millionth and 8 millionth mark. As in the first experiment, Pooled LRU pools key-value pairs by cost, and for a fixed cache size, each pool is allotted a portion of the cache proportional to the cost value. Since the only occurring cost values are 1, 100 and 10K, 99% of the cache is dedicated to the pool of expensive key-value pairs. On the other hand, the expensive key-value pairs from a single TF only occupy a third of the maximum cache size, so that a cache size ratio of 2/3 can store all expensive key-values from two TFs, and a cache size ratio of 1 can store those of 3 separate TFs. Now the point at which all TF1 key-value pairs are evicted occurs when the total size of the expensive key-value pairs requested in a subsequent TF exceeds the cache size. When the cache size ratio is 1/3 or less, this occurs before the end of TF2 (4 million requests). At a cache size ratio of 2/3 or higher, the eviction time occurs during TF4 (roughly between 8 and 12 million requests).

Finally, CAMP maintains an LRU queue for each cost-to-size ratio, and unlike Pooled LRU, these queues can be resized dynamically. Because key-value pairs requested at a later time can have higher priority of being evicted than those requested earlier, CAMP only guarantees that those requested after TF1 that have the highest cost-to-size ratio will have lower priority than any TF1 request. This observation yields the loose guarantee that all TF1 key-value pairs will be evicted by the time the total size of all newer requested items with the highest cost-to-size ratio reaches the cache size. According to Figure 6d, for a cache size ratio of 0.75, there are cache-resident still key-value pairs from TF1 at the end of the simulation. This is explained by the fact that the highest cost-to-size key-values contribute less than 1/20th of the maximum cache size per TF. Together over the whole trace, these key-value pairs will fit in caches with cache size ratios greater than 0.5. Hence, the condition guaranteeing the eviction of all TF1 items is satisfied with the .25 cache size of Figure 6c and not satisfied with the .75 cache size of Figure 6d.

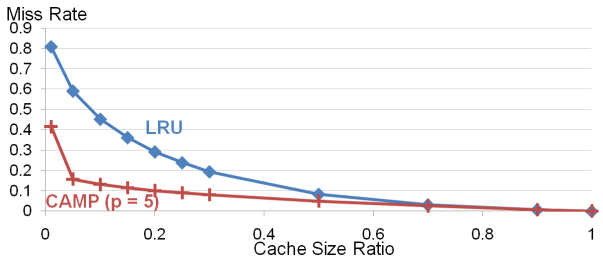


Figure 7: Miss rate as a function of cache size with variable sized key-value pairs and constant cost.

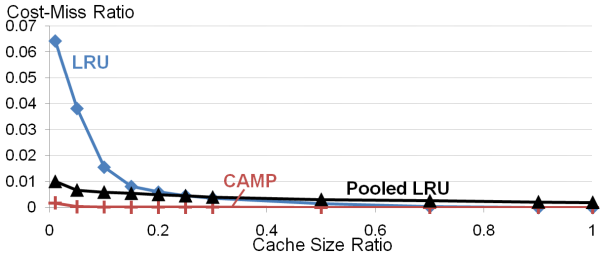
3.2 Other Traces

The trends reported in Subsection 3.1 hold true with other traces. The most insightful results are obtained with the two possible extremes, namely, variable sized key-value pairs with almost similar costs and equi-sized key-value pairs with varying costs. We describe these in turn.

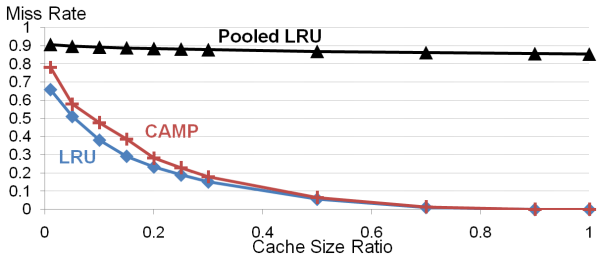
With variable sized key-value pairs whose cost is identical, CAMP renders small key-value pairs KVS resident, providing a lower miss rate when compared with LRU, see Figure 7. Pooled-LRU constructs one pool and behaves the same as LRU. Because the cost of the key-value pairs is set to 1, the cost-miss ratio of a technique is equal to its miss rate. Hence Figure 7 can also be interpreted as the cost-miss ratio of LRU and CAMP as a function of cache size ratio.

With equi-sized key-value pairs that incur a different cost, CAMP continues to provide a superior cost-miss ratio to both LRU and Pooled-LRU, see Figure 8a. With a limited amount of memory, CAMP's miss rate is slightly worse than that of LRU as it favors high cost key-value pairs, see Figure 8b. With Pooled-LRU, we were challenged in determining the size of different pools as there was no clear way to partition the range of different costs. In the original trace where key-value pairs could only have one of three different cost values, items were pooled by their cost value. Here, we opted to pool items by range of cost values. Specifically, the ranges were 1 to 100, 100 to 10,000 and 10,000 and beyond. The available memory was then divided among the three pools in such a way that each pool received an amount proportional to the lowest cost value in its range. With this assignment, Pooled-LRU results in a superior cost-miss ratio with small cache-size ratios. With larger cache size ratios, its partitioning of space makes it inferior to both LRU and CAMP.

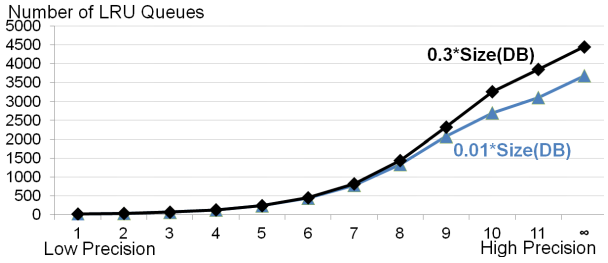
In comparison to the trace with three distinct cost values (Figure 5b), the trace with equi-sized key-value pairs has key-value pairs with many more distinct cost values. Therefore, for this trace, CAMP creates a larger number of LRU queues when no rounding takes place, see Figure 8c. This increase is due to the fact that there is a greater number of cost-to-size ratios as a result of the considerably larger set of cost values. With additional rounding, *i.e.* at lower precision, the number of LRU queues in the two traces decrease significantly and converge without any sizable performance degradation.



(a) Cost-miss ratio as a function of cache size ratio.



(b) Miss rate as a function of cache size ratio.



(c) Number of queues as a function of precision.

Figure 8: Simulation results with equi-sized key-value pairs and variable costs.

3.3 Admission control and comparison with an offline algorithm

This section compares CAMP with a static offline algorithm that assigns key-value pairs across the available storage based on advanced knowledge of their access frequency. The offline algorithm has very good performance in terms of its cost-miss ratio. Although the offline algorithm is impractical to implement, it motivates the introduction of admission control to temper the rate at which CAMP brings key-value pairs into the cache. CAMP with admission control has improved cost-miss ratio and requires substantially less data to be written to the cache than CAMP without admission control.

If each request to a key-value pair is generated independently according to a known probability distribution, then a static algorithm that always keeps the same set of objects in the cache achieves the optimal expected cost-miss ratio. Define the benefit of key-value pair p to be $ben(p) = prob(p)cost(p)$, where $prob(p)$ is the probability that p is requested on each request. The expected cost-miss-ratio of any static algorithm whose cache contents do not change is the sum of $ben(p)$ over all key-value pairs not included in the cache. The optimal

static algorithm then is the algorithm that maximizes the total benefit of key-value pairs included in the cache. Finding the optimal static algorithm is the same as the well-known Knapsack problem which is NP-complete [6]. However, if ϵ is defined to be the ratio of the size of the largest key-value pair to the size of the cache, a simple greedy algorithm can be shown to have an expected benefit that is within ϵ of the optimal. The greedy algorithm sorts the key-value pairs according to $ben(p)/size(p)$ in decreasing order. Next, it adds key-value pairs to the cache in order and stops just before the capacity of the cache is exceeded. We call this algorithm StaticGreedy.

The analysis for StaticGreedy is a standard analysis of the greedy algorithm for the Knapsack problem. We include it here for completeness. Let M be the total capacity of the memory. For any set of key-value pairs P, $size(P)$ is the sum of the sizes of key-value pairs in P and $ben(P)$ is the sum of the benefits of the key-value pairs in P. Define P_{SG} be the set of key-value pairs selected by the StaticGreedy algorithm. Clearly P_{SG} depends on M, but we will assumed a fixed M and not include M as a parameter in the notation.

Theorem 4. *For any set of key-value pairs P such that $size(P) \leq M$,*

$$ben(P)(1-\epsilon) \leq ben(P_{SG}),$$

where $\epsilon = \max_p size(p)/M$.

Proof: Let $\overline{p}$ be the last key-value pair considered by the StaticGreedy which does not fit in the cache. Define $m = M - size(P_{SG})$ to be the amount of unfilled capacity in StaticGreedy's cache. Then $m < size(\overline{p})$.

Define $\alpha = ben(\overline{p})/size(\overline{p})$. Due to the greedy nature of the StaticGreedy algorithm, for any $p \in P_{SG}$, $ben(p)/size(p) \geq \alpha$. Also, for any $p \notin P_{SG}$, $ben(p)/size(p) \leq \alpha$.

Because the key-value pairs in P do not exceed the capacity of the cache, we know that $size(P - P_{SG}) \leq size(P_{SG} - P) + m$. Since the benefit to size ratio of all the key-value pairs in $P - P_{SG}$ is at most α, we know that $ben(P - P_{SG}) \leq \alpha(size(P_{SG} - P) + m)$. Also since the benefit to size ratio of all key-value pairs in P_{SG} is at least α, $ben(P_{SG} - P) \geq \alpha \cdot size(P_{SG} - P)$. Putting it together, we get that:

$$\begin{aligned} & ben(P) - ben(P_{SG}) \\ = \; & ben(P - P_{SG}) - ben(P_{SG} - P) \\ \leq \; & \alpha(size(P_{SG} - P) + m) - \alpha \cdot size(P_{SG} - P) \\ = \; & \alpha m \end{aligned}$$

The overall benefit to size ratio of the set P_{SG} is at least α, so $ben(P_{SG})/(M - m) \geq \alpha$. Thus, we have that

$$ben(P) - ben(P_{SG}) \leq m \left(\frac{ben(P_{SG})}{M - m} \right)$$

Rearranging:

$$ben(P) \left(1 - \frac{m}{M}\right) \leq ben(P_{SG}).$$

Since $m < size(\overline{p})$, $m/M \leq \epsilon$, and we have that $ben(P)(1-\epsilon) \leq ben(P_{SG})$.

■

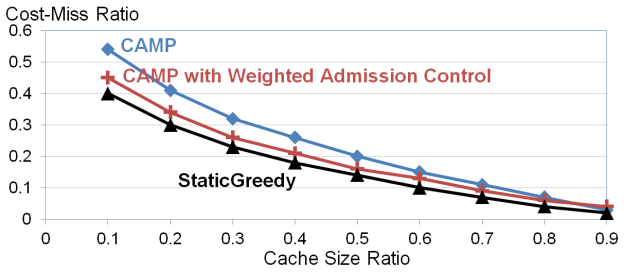


Figure 9

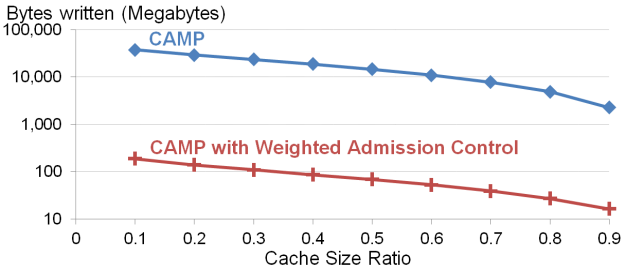


Figure 10

StaticGreedy performs very well on the BG trace files because the requests are generated independently according to a fixed distribution. StaticGreedy estimates the probability that a key-value pair p is requested by post-processing the trace file and determining the frequency of requests to key-value pair p. While the cost-miss ratio of StaticGreedy is very close to optimal for request sequences generated according to a fixed distribution, it is impractical for several reasons. First, it is costly to compute the frequencies over time and determine the set of key-value pairs to include in the cache. Secondly, the popularity of key-value pairs will fluctuate over time and StaticGreedy has no way to adapt unless frequencies are recomputed and a new set of key-value pairs to include in the cache is determined. Furthermore, in our experiments StaticGreedy has the unfair advantage that we measure its performance on the same traces that we use to estimate the frequency. Nonetheless, StaticGreedy serves as a useful yardstick to evaluate CAMP and to develop improvements.

StaticGreedy shows lower cost-miss ratios than CAMP, especially with smaller cache size ratios for which space in the cache is more critical. The superior performance of StaticGreedy motivates augmenting CAMP with admission control. Since CAMP brings every requested key-value pair into the cache, there is less room for key-value pairs of high value (i.e., whose benefit-to-size ratio value is high). The success of StaticGreedy suggests that it is better to keep such costly key-value pairs in the cache instead of recently requested key-value pairs. We implemented an admission control policy that admitted a key-value pair based on its cost-to-size ratio (CSR). On a request to a key-value pair p that results in a cache miss, p is brought into the cache with probability CSR_p/CSR_{max}, where CSR_{max} is the largest CSR for any key-value pair requested up to that point. The results are given in Figure 9. CAMP with admission gets much closer to the cost-miss ratio of StaticGreedy. Furthermore, the admission control saves CAMP the cost of writing many requested key-value pairs into the cache. The graph in Figure 10 shows the number of bytes written into the cache. Admission

control saves CAMP roughly a factor of 100 in the total number of bytes written to the cache.

Experiments were performed with an unweighted version of admission control with each requested key-value pair brought into the cache with some fixed probability. The weighted version of admission control whose probability of admission depends on the cost-to-size ratio performed better in most situations and has the advantage that it does not require tuning the admission probability parameter which could depend on trace characteristics. The experiments reported in the above graphs were also performed on a variety of different traces files with different levels of skew in the distribution over key-value pairs. The same trends are born out on all the traces tested.

4 An Implementation

We implemented CAMP in the IQ Twemcache, a modified version of the Twemcache v2.5.3 [23] that implements the IQ framework [12]. This implementation computes the cost of a key-value pair by noting the timestamp of a miss observed by a get (iqget) and the subsequent insertion of the computed value using a set (iqset). The difference between these two timestamps is used as the cost of the key-value pair.

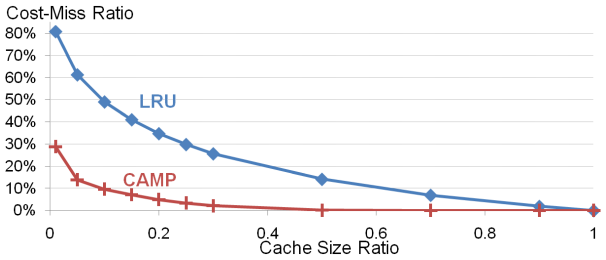
The approach taken to provide recomputation time is to use the service time to compute a key-value pair (and piggybacked as a part of the KVS put). Application provided hints are another possibility.

CAMP does not need to address the issue of malicious applications with misleading costs, because it is intended for use in a middleware deployed in a trusted environment (data center) with no adversary.

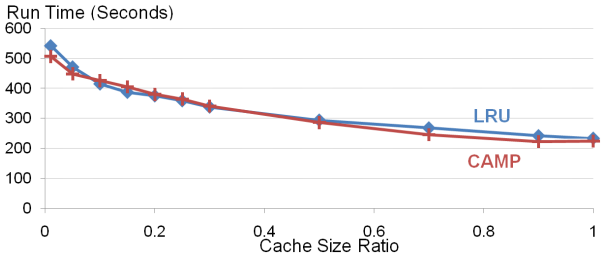
We developed an application that implements the request generator of Section 3 by reading a trace file and issuing requests to the KVS using Whalin client version 2.6.1 [25]. We used the trace file with synthetic costs of {1, 100, 10K} per discussions of Section 3. Figure 11a shows the observed cost-miss ratio with LRU and CAMP as a function of different cache size ratios. CAMP incurs a significantly lower cost for the missing keys with smaller cache sizes. This difference becomes smaller with larger cache sizes because the KVS miss rate drops. These results are consistent with those reported in Section 3.

Figure 11b shows the amount of time required to run the trace with both LRU and CAMP. It includes the following: (1) the time for either LRU or CAMP to process a cache hit and to make replacement decisions when the memory is exhausted; (2) the time to transmit a key-value pair across the network (with both a cache hit and a cache insert); and (3) the time to copy a key-value pair across the different software layers. The results show that CAMP provides response times comparable to those of LRU. If the cost was included in the reported response times, CAMP would have been significantly faster than LRU, resembling the results reported in Section 3. Here, we wanted to show results that demonstrate that an implementation of CAMP is as fast as LRU while ignoring the cost associated with keys.

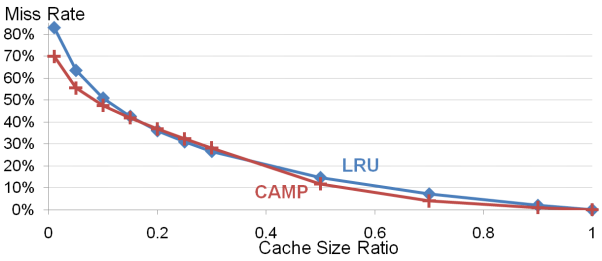
With both CAMP and LRU, the run time decreases as a function of cache size. The explanation for this is as follows. A get(k_i) that observes a miss is followed by a set(k_i,v_i). With a small cache size that is full, the set operation must evict one or more existing key-value pairs and write the new k_i-v_i in the cache. This write operation requires copying of



(a) Cost-miss ratio as a function of the cache size ratio.



(b) Run time as a function of the cache size ratio.



(c) Miss rate as a function of the cache size ratio.

Figure 11: Implementation results, where the precision of CAMP is set to 5.

the k_i-v_i from the network buffer into the memory of the cache manager. A larger cache size reduces the number of such operations because a higher percentage of get(k_i) operations observe a cache hit, see Figure 11c. This explains why the run time improves with both LRU and CAMP using a larger cache size. This also explains why CAMP provides a faster response time than LRU for those cache sizes that provide a lower miss rate, *i.e.*, cache ratio of 0.01.

5 Related Work

Management of KVS memory space must address two key questions: First, how should memory be assigned to a key-value pair? Second, what key-value pairs should occupy the available memory? In the context of online algorithms that manage memory, the second question must identify what key-value pair should be evicted when there is insufficient space for an incoming key-value pair. CAMP provides an answer to this question. Below, we survey the state of the art and describe how CAMP is different.

LRU-K [22], 2Q [14], and ARC [19] are adaptive replacement techniques that balance between the recency and the frequency features of a workload continuously, improving cache hit rate for fix sized disk pages with a constant cost. CAMP is different in that it considers both the size of a key-value pair and its cost. CAMP is an efficient implementation of the GDS algorithm [4], visiting a significantly fewer number of heap nodes than GDS, see the discussion of Figure 4 in Section 2.

Similar to CAMP, GD-Wheel [17] strives to enhance the efficiency of GDS. There are, however, some significant differences in approach. GD-Wheel rounds the overall priority for

each key-value pair instead of the cost-to-size ratio which makes it difficult to evaluate the cost of approximation. The GD-Wheel study does not give a direct comparison between GD-Wheel and GDS. With CAMP we are able to give well-defined guarantees on the competitive ratio of CAMP relative to GDS. Moreover, GD-Wheel does not address how to select their precision parameter N or give an empirical characterization of performance as a function of precision. Finally, GD-Wheel must implement occasional migration procedures wherein all the key-value stores within a GD-Wheel are migrated to the next level. CAMP does not require such a migration step as it uses the cost-to-size ratio as the basis of the rounding scheme (which does not change while a key-value pair is in the cache).

While deciding how space should be assigned (answer to the first question) is a different topic, there are implementations that strive to answer this question in combination with a replacement technique. For example, the memory used by a Twemcache server instance is managed internally using a slab allocation system [3] in combination with LRU. Below, we describe this technique and how it is different than CAMP.

Twemcache divides memory into fix sized slabs (chunks of memory), the default size being 1 Megabyte. Each slab is then assigned a slab class and further sub-divided into smaller chunks based on its slab class. For example, a slab class of 1, the smallest size, would have a chunk size of 120 bytes. This means that a single slab of class 1 can fit 8737 (1 MB / 120 byte) chunks. Every subsequent higher slab class uses chunk sizes that are approximately a factor of 1.25 larger. So, a slab class of 2 accommodates 6898 chunks each 152 bytes in size. This slab class stores key-value pairs whose size is between 120 and 152 bytes. The largest slab class uses a chunk size that accommodates the entire slab.

When storing a key-value pair, k_i-v_i, the server identifies the size required to store k_i-v_i along with some meta-data header information. The memory allocation attempts the following steps, proceeding in order until it is able to satisfy the allocation request:

1. Replace an expired key-value of the smallest slab class that can accommodate the entire k_i-v_i.
2. Find a free chunk within allocated slabs of that slab class.
3. Allocate a new slab to the matching slab class and assign one of the chunks.
4. Evict an existing key-value pair using LRU and replace its contents.

A limitation of the slab allocation system is that, once a slab has been allocated to a particular slab class, it will forever maintain its class assignment. The consequence of this rigid assignment is that it may prevent future requests from storing key-value pairs in the KVS. For example, a certain workload may assign all slabs to the slab class 1 (120 bytes). Subsequently, the workload may change and require chunks of slab class 5 (304 bytes). Since all slabs were already assigned to slab class 1, all requests to store key-value pairs with a slab class of 5 fail. This phenomenon is termed slab calcification and causes a KVS using slab based allocation to under-utilize its available memory.

Twemcache attempts to resolve this calcification limitation by randomly evicting a slab from another class if it is unable to allocate an item. This approach may evict potentially hotly accessed items, impacting the cache hit rate adversely. Additionally, the random slab eviction does not deal with the case when a disproportionately small number of slabs are assigned to the needed slab class. To illustrate, assume only one slab was assigned to the slab class 5 (0.1% of total memory) and the workload changes such that key-value pairs corresponding to slab class 5 are referenced much more frequently. All the requests have

to compete for chunks in the single slab whereas the remaining cache space is now under-utilized. Since key-value pairs can still be allocated, the random slab eviction does not activate to free up more slabs.

In [13], we propose a simple algorithm that controls the physical layout of key-value pairs in conjunction with CAMP's eviction policy. The new algorithm implements each queue of CAMP using First in First Out (FIFO) instead of LRU since FIFO provides a means of filling holes without moving items within the cache [9]. To dynamically allocate memory between the queues, the algorithm of [13] divides memory into same-size blocks. Starting from an empty block, key-value pairs are placed within a block next to each other in the order in which they entered the KVS. Each FIFO queue owns a set of blocks. While the blocks in each queue are logically ordered in FIFO order, they are physically scattered across the memory. Thus, a block can be emptied and reassigned to another queue as dictated by CAMP's allocation. If there is no more room for an incoming key-value pair in the block at the tail of a queue, the algorithm selects a block to flush and adds it to the tail of the queue. Due to the imposed page limit, we decided to maintain our focus on CAMP as a replacement policy and not include the memory allocation algorithm. This presentation of CAMP is more complete than [10] by including (1) an admission control and its comprehensive evaluation and (2) the offline algorithm, StaticGreedy, as a measuring yardstick for CAMP and future online algorithms.

6 Conclusion and Future Research

We have presented a new efficient implementation of GDS called CAMP. CAMP takes advantage of rounded priority values so that the underlying data structure selects a key-value pair for eviction very efficiently. Moreover, we have shown that on typical access patterns, there is no degradation in performance due to loss in precision of the priority values. CAMP outperforms LRU as well as Pooled LRU in the overall cost of caching key-value pairs in a sequence of requests in which the cost to access different key-value pairs vary dramatically. The implementation of CAMP in IQ Twemcache shows that the overhead in implementing replacement decisions is as efficient as LRU.

Our short term research plans include examining the performance of CAMP in a wider variety of settings. It would be particularly interesting to test the performance of CAMP on real trace data and in realistic deployments. Another important direction to explore is the use of admission control policies in conjunction with CAMP that also considers variations in key-value sizes and costs. This should enhance the performance of CAMP by not inserting unpopular key-value pairs that are evicted before their next request.

More longer term, we are extending CAMP for use with a hierarchical cache (using SSD, hard disk, or both). This includes an investigation of environments that require CAMP to manage both key-value pairs that pertain to the result of computations and fix sized disk pages used by the computations. We are also investigating a decentralized CAMP in the context of a cooperative caching framework such as KOSAR [20]. A challenge here is how to maintain a last replica of a cached key-value pair without allowing those that are never accessed again to occupy the KVS indefinitely.

References

[1] Barahmand, S., and Ghandeharizadeh, S. BG: A Benchmark to Evaluate Interactive Social Networking Actions. *CIDR* (January 2013).

[2] Barahmand, S., Ghandeharizadeh, S., and Yap, J. A Comparison of Two Physical Data Designs for Interactive Social Networking Actions. *CIKM* (2013).

[3] Bonwick, J. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer* (1994), pp. 87–98.

[4] Cao, P., and Irani, S. Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)* (1997).

[5] Fredman, M. L., and Tarjan, R. E. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. ACM 34*, 3 (July 1987), 596–615.

[6] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[7] Ghandeharizadeh, S., and Barahmand, S. A Mid-Flight Synopsis of the BG Social Networking Benchmark. *Fourth Workshop on Big Data Benchmarking* (October 2013).

[8] Ghandeharizadeh, S., Boghrati, R., and Barahmand, S. An Evaluation of Alternative Physical Graph Data Designs for Processing Interactive Social Networking Actions. *TPC Technology Conference* (September 2014).

[9] Ghandeharizadeh, S., Ierardi, D., and Zimmermann, R. An Algorithm for Disk Space Management to Minimize Seeks. *Information Processing Letters 57* (1996), 75–81.

[10] Ghandeharizadeh, S., Irani, S., Lam, J., and Yap, J. CAMP: A Cost Adaptive Multi-Queue Eviction Policy for Key-Value Stores. In *Proceedings of the 15th International Middleware Conference, Bordeaux, France, December 8-12, 2014* (2014), pp. 289–300.

[11] Ghandeharizadeh, S., and Yap, J. Cache Augmented Database Management Systems. In *ACM SIGMOD DBSocial Workshop* (June 2013).

[12] Ghandeharizadeh, S., Yap, J., and Nguyen, H. Strong Consistency in Cache Augmented SQL Systems. *ACM/IFIP/USENIX Middleware* (December 2014).

[13] Irani, S., Lam, J., and Ghandeharizadeh, S. Cache Replacement with Memory Allocation. In *Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments, ALENEX 2015, San Diego, CA, USA, January 5, 2015* (2015), pp. 1–9.

[14] Johnson, T., and Shasha, D. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB* (1994), pp. 439–450.

[15] Jung, H., Han, H., Fekete, A., Heiser, G., and Yeom, H. A Scalable Lock Manager for Multicores. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), SIGMOD '13, pp. 73–84.

[16] Larkin, D., Sen, S., and Tarjan, R. E. A Back-to-Basics Empirical Study of Priority Queues. In *ALENEX* (2014), pp. 61–72.

[17] Li, C., and Cox, A. L. GD-Wheel: A Cost-Aware Replacement Policy for Key-Value Stores. In *7th Workshop on Large-Scale Distributed Systems and Middleware* (2013).

[18] Matias, Y., Sahinalp, S. C., and Young, N. E. Performance Evaluation of Ap-

proximate Priority Queues. In *Proceedings of Fifth DIMACS Implementation Challenge* (1996).

[19] Megiddo, N., and Modha, D. S. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *FAST* (2003), USENIX.

[20] Mitra LLC. KOSAR, http://kosarsql.com 2014.

[21] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling Memcache at Facebook. In *NSDI* (Berkeley, CA, 2013), USENIX, pp. 385–398.

[22] O'Neil, E. J., O'Neil, P. E., and Weikum, G. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *ACM SIGMOD* (1993).

[23] Rajashekhar, M., and Yue, Y. Twitter memcached (Twemcache) is version 2.5.3, https://github.com/twitter/twemcache/releases/tag/v2.5.3.

[24] Ugander, J., Karrer, B., Backstrom, L., and Marlow, C. The Anatomy of the Facebook Social Graph. *CoRR abs/1111.4503* (2011).

[25] Whalin, G., Wang, X., and Li, M. Whalin memcached Client Version 2.6.1, http://github.com/gwhalin/Memcached-Java-Client/releases/tag/release_2.6.1.

[26] Young, N. E. On-line Caching as Cache Size Varies. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)* (1991).