GD-Wheel: A Cost-Aware Replacement Policy for Key-Value Stores

Conglong Li
Rice University
cli@rice.edu

Alan L. Cox
Rice University
alc@rice.edu

***Abstract*—Various memory-based key-value stores, such as *Memcached* and *Redis*, are used to speed up dynamic web applications. Specifically, they are used to cache the results of computations, like database queries. Currently, these key-value stores use either *LRU* or an *LRU* approximation as the replacement policy for choosing a key-value pair to be evicted from the store. However, if the cost of recomputing cached values varies a lot, like in the RUBiS and TPC-W benchmarks, then none of these replacement policies are the best choice. Instead, it can be advantageous to take the cost of recomputation into consideration. To that end, this paper proposes a new replacement policy, *GD-Wheel*, which seamlessly integrates recency of access and cost of recomputation. This paper applies GD-Wheel to Memcached and evaluates its performance using the *Yahoo! Cloud Serving Benchmark*. The evaluation shows that GD-Wheel, when compared to LRU, greatly reduces the total recomputation cost, as well as the average and 99th-percentile read access latency for the application.**

I. Introduction

Memory-based key-value stores are used by many large-scale web applications. For instance, *Memcached* [10] is used at Facebook, Twitter and Zynga; and *Redis* [1] is used at GitHub, Flickr and Stack Overflow. As an intermediate layer in the application's data-management hierarchy, these key-value stores play an important role in speeding up dynamic web applications. By caching the results of computations, like database queries, key-value stores reduce the application read access latency.

Figure 1 illustrates how a key-value store is used to cache the results of computations. After receiving the HTTP request, the web application first tries to retrieve the value by sending a GET request with the appropriate key to the key-value store. If the value is returned at step 3, then the application skips to step 6. We refer this as a GET hit. In this case, the application's read access latency is only the sum of steps 2 and 3. If, however, a *not found* error is returned at step 3, then the application needs to request the lower layer in the data-management hierarchy, e.g., a database, to compute the value. We refer to this as a GET miss. In this case, the application's read access latency is the sum of steps 2 through 5. After the computation, the application usually sends a SET request with the computed value to the key-value store in order to retain the key-value pair for later use. However, due to the limited capacity of the key-value store, key-value pairs may be evicted from the store. This results in GET misses and recomputations of values.

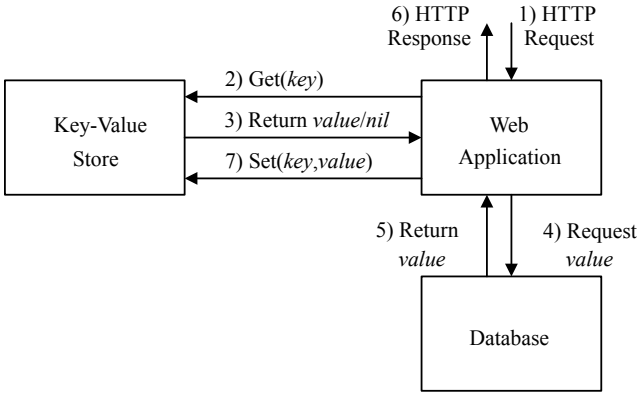

Fig. 1: Using a Key-Value Store as a Cache.

Currently, key-value stores use *LRU* or an approximation to LRU as the replacement policy for choosing a key-value pair to be evicted from the store. For example, *MemC3* recently proposed a *CLOCK*-based approximation to LRU for use in Memcached in order to increase space efficiency and concurrency [9]. However, neither LRU nor an LRU approximation is necessarily the best choice if the cost of recomputing cached values varies a lot. (In Figure 1, this recomputation cost is the sum of steps 4 and 5.) Then, it may be advantageous to take the cost of recomputation into account when

making replacement decisions. In fact, real-world key-value store deployments [14] and several representative web application benchmarks [5] provide evidence that significant cost variations do exist.

To that end, this paper proposes a new replacement policy, *GD-Wheel*, which is an efficient implementation of the *GreedyDual* algorithm [17] using the *Hierarchical Cost Wheels* data structure. The GD-Wheel algorithm seamlessly integrates recency of access and cost of recomputation in making replacement decisions. Using the Hierarchical Cost Wheels data structure for a given cost range, GD-Wheel yields an efficient implementation of the GreedyDual algorithm with constant time complexity, just like LRU.

This paper also describes the application of GD-Wheel to Memcached. This includes changes to the SET request protocol so that clients can optionally provide cost information with each key-value pair. This paper then evaluates the performance of Memcached with GD-Wheel using the *Yahoo! Cloud Serving Benchmark* (YCSB) [7]. This evaluation shows that GD-Wheel, when compared to LRU, greatly reduces the total recomputation cost as well as the average and 99th-percentile read access latency for the data. Specifically, the total recomputation cost is reduced by as much as 91.05%, and the average and 99th-percentile latencies are reduced by as much as 53.95% and 94.65%, respectively.

The rest of this paper is organized as follows. Section II provides deeper motivation for our work. Section III presents the GD-Wheel algorithm and describes the implementation of GD-Wheel in the Memcached key-value store. Section IV describes our experimental methodology, and Section V presents our experimental results. Finally, Section VI discusses related work, and Section VII summarizes our conclusions.

II. Background

This section discusses different sources of cost variations in recomputations of key-value pairs as well as some relevant characteristics of real-world workloads on key-value stores.

A. Cost Variations of Recomputations

There exist two common sources of cost variations in recomputations. The first source is different kinds of simultaneous cache usage, in other words, simultaneously caching different types of objects from different levels in the application stack. These different types of objects will likely have different recomputation times, and thus different costs. For example, Facebook uses Memcached as both a query cache to lighten the load on databases and a generic computation cache to store the results of different applications [14].

The second source is the cost variations among objects of the same type or at the same level in the application stack. Previous work on some dynamic web application benchmarks has shown that even objects at the same level could have widely varying costs. Bouchenak *et al.* studied two benchmarks: *RUBiS* and *TPC-W* [5]. RUBiS models an auction site that supports selling, browsing and bidding [3]. For its read-only database queries, the extra response time that Bouchenak *et al.* measured in the case of a cache miss was: 10*ms* for 15% of the requests; 60*ms* to 95*ms* for 70% of the requests; and 240*ms* for 4% of the requests. TPC-W is a web server and database performance benchmark that simulates an online bookstore [2]. For its read-only database queries, the extra response time that Bouchenak *et al.* measured in the case of a cache miss was: 10*ms* to 25*ms* for 42% of the requests; 45*ms* to 150*ms* for 22% of the requests; and 210*ms* to 300*ms* for 23% of the requests. Thus, for RUBiS and TPC-W there exists large variation in the execution times for database queries.

B. Real-World Workloads of Key-Value Stores

Recently, Atikoglu *et al.* [4] and Nishtala *et al.* [14] have provided a detailed picture of how Facebook uses Memcached.

1) Key/Value Sizes: Small keys and values dominate in all workloads. However, there is a very large variation in the sizes of the cached items. Atikoglu *et al.* reported that most keys are smaller than 32 bytes and most values are no more than a few hundred bytes. Nonetheless, there are a few very large values (around 1 MB). Nishtala *et al.* reported that the responses to Memcached requests have a median size of 135 bytes and a mean size of 954 bytes.

2) GET-to-SET Ratio: All workloads are read intensive. Atikoglu *et al.* reported that the GET-to-SET ratio was 30:1 for the workload which is the most representative of general cache usage. Since each GET miss is usually followed by a SET to update the cache, the GET-to-SET ratio is dependent on the GET miss rate.

3) Miss Rate: The miss rate is not negligible. Atikoglu *et al.* reported that the mean GET hit rate

ranged from 81.4% to 98.7% across the different workloads. Considering the latency differences between GET hits and misses, the extra time required in the case of a GET miss is a considerable part of the total application read access latency.

III. GD-Wheel Replacement Policy

A. The GreedyDual Algorithm

The GD-Wheel replacement policy is derived from the GreedyDual algorithm [17]. GreedyDual is actually a family of algorithms, but we only focus on one version which is a generalization of LRU. Cao *et al.* introduced the best known implementation of GreedyDual [6]. Their implementation associates a value, H, with each cached object and uses an inflation value L for aging. On retrieval or on a hit on an object p, $H(p)$ is set to $L+c(p)$ where $c(p)$ is the cost of p. On eviction, L is set to the smallest H in the cache and the object with $H=L$ is evicted. Just as H combines the cost and age inflation values, GreedyDual seamlessly integrates recency of access and cost of recomputation in making eviction decisions. Cao *et al.* implemented the GreedyDual algorithm by maintaining a priority queue based on the H values. Consequently, handling a hit or an eviction requires $O(logn)$ time, which is slower than the LRU algorithm.

B. Hierarchical Cost Wheels

In order to reduce the time complexity, we implemented the GreedyDual algorithm using a data structure that we call Hierarchical Cost Wheels. This data structure is inspired by Varghese and Lauck's *Hierarchical Timing Wheels* [16]. The Hierarchical Cost Wheels structure is made up of a series of *Cost Wheels*. A Cost Wheel is basically an array of queues. Instead of storing all objects in a single priority queue, key-value pairs with different costs are stored in different Cost Wheel queues. Since handling a key-value pair only needs to deal with a single queue inside the Hierarchical Cost Wheels, GD-Wheel requires only $O(1)$ time for handling a hit or an eviction, just like LRU.

Since each Cost Wheel's size N is fixed, the Hierarchical Cost Wheels can only support a limited range of costs. To ensure that this limited range of costs is large enough to support any reasonable cost variation of key-value pairs, we used multiple Cost Wheels in a hierarchy such that each higher level Cost Wheel supports a larger range of cost. An object p with $H(p)=a+bN+cN^2+...+xN^y$ $(0<a,b,c,...,x<N)$ will reside in the $(y+1)$th level Cost Wheel's xth queue.

C. The GD-Wheel Replacement Policy

Algorithm 1 The GreedyDual Algorithm

```
Let M be the set of all key-value pairs in the store.
Initialize L ← 0.
For each requested key-value pair p:
if p is already in the store,
    Dequeue p.
    H(p) ← L + c(p).
    Enqueue p.
if p is not in the store,
    if there is not enough room in the store for p,
        Let L ← minH(q), (q ∈ M).
        Evict q such that H(q) = L.
        If L reaches a multiple of the Cost Wheel
size N, do migration.
    H(p) ← L + c(p).
    Enqueue p.
end
```

Algorithm 1 describes the GreedyDual algorithm as implemented in GD-Wheel. It's slightly modified from the original algorithm because of the different data structure. If the requested key-value pair p is already in the key-value store, $H(p)$ will be updated based on the change of L, and p will be dequeued from its current queue and enqueued to the queue corresponding to the new $H(p)$.

If p is not in the store and there is not enough room for p, L is updated to the minimum H inside the store. Then a key-value pair q that has $H(q)=L$ will be evicted. In addition, a *migration* is performed when L reaches a multiple of the Cost Wheel's size N: all key-value pairs having H value between L and $L+N-1$ will be migrated to a lower-level Cost Wheel. A key-value pair with $H=a+bN$ $(0<a,b<N)$ will be migrated from the $2nd$ level Cost Wheel's bth queue to the $1st$ level Cost Wheel's ath queue when L reaches bN. When there is enough room for p, it will be enqueued with $H(p)=L+c(p)$.

D. The Memcached Key-Value Store

Memcached uses a slab allocator for memory allocation. This slab allocator uses different slab classes to store key-value pairs of different size ranges. In more detail, the primary storage is broken up into pages of the same size. When a page is assigned to a slab class, it is

cut into chunks of a specific size for that slab class. By default, the chunk sizes differ by a factor of 1.25 from one slab class to the next larger class. Each slab class has its own LRU queue for replacement. When storing key-value pairs, they are allocated from the slab class that is the nearest fit. If there are no free chunks, and no free pages for that slab class, Memcached will look at the end of the slab class's LRU queue for an item to reclaim. It will first search the last few items for one which has already expired and thus is free for reuse. If it cannot find an expired item, it will then evict the least recently used item. Memcached also balances the eviction rates across slab classes by periodically rebalancing slab assignments of pages.

Our implementation replaces each slab class's LRU replacement policy with GD-Wheel. However, Memcached's rebalancing policy is unchanged. To the metadata of each key-value pair, we add additional fields including the cost of the key-value pair and the indices required by the Hierarchical Cost Wheels. These additional fields introduce an 8-byte metadata overhead to each stored key-value pair. The SET request protocol is modified in order to let clients send the additional cost information with each key-value pair to the Memcached server.

IV. Methodology

All experiments were run on two machines connected to the same 1 Gbps network switch. Each machine had two Quad-Core AMD Opteron 2393 SE processors and 16 GB of DRAM. One machine acts as the Memcached server and the other acts as its client. We configured Memcached with a 2 GB cache, 8 threads, and one of the two replacement policies. On the client machine, we use the YCSB Benchmark to generate GET and SET requests [7]. When the GD-Wheel replacement policy is used, the client additionally provides the cost of the key-value pair with the SET requests to Memcached.

The YCSB Benchmark operates in two phases: the first, the warmup phase, loads the key-value store by sending SET requests for a certain number of different key-value pairs; and the second, the measurement phase, executes the desired workload. Since the pool of key-value pairs is shared by the two phases, the number of SET requests in the warmup phase will directly affect the hit rate in the measurement phase. Thus, we controlled the number of SET requests in the warmup phase in order to keep the hit rate during the measurement phase at about 95% for LRU. During the measurement phase, each workload sends 10 million GET requests following a *zipf* distribution on the requested keys. During this phase, when a GET request fails, or misses, a subsequent SET request will be sent for the same key. As a result, each workload's measurement phase will send 10 million GET requests and about 500,000 SET requests, for a GET-to-SET ratio of about 20:1.

Workload	Key/Value Size (bytes)	Cost Distribution
1	16 / 256	10-30(80%);120-180(15%);350-450(5%)
2	16 / 256	10-30(20%);120-180(75%);350-450(5%)
3	16 / 256	10-30(50%);120-180(25%);350-450(25%)
4	16 / 256	10(100%)
5	16 / 256	20-400(100%)
6	16 / 64	10-30(80%);120-180(15%);350-450(5%)
7	16 / 128	10-30(80%);120-180(15%);350-450(5%)
8	16 / 2048	10-30(80%);120-180(15%);350-450(5%)
9	16 / 4096	10-30(80%);120-180(15%);350-450(5%)

TABLE I: Workload Configurations.

Workload	LRU	GD-Wheel	Reduction (%)
1	28918201	5826803	79.85
2	67413318	8930531	86.75
3	72663505	6503548	91.05
4	4938670	4938200	0.01
5	103395684	24708191	76.10
6	29201904	23873144	18.25
7	29196819	5718156	80.42
8	29210183	8330009	71.48
9	28934337	9109587	68.52

TABLE II: Total Recomputation Costs for LRU and GD-Wheel.

Table I shows our different workloads. All workloads use 16-byte keys. Workload 1 is our baseline with 256-byte values, three groups of costs based on the cost variations in RUBiS and TPC-W, and an exponential distribution for the proportion of each cost group. Workloads 2 and 3 use the cost proportions from RUBiS and TPC-W, respectively. Workload 4 uses the same cost for all objects. Workload 5 adopts a totally random cost distribution. Workloads 6 to 9 test with different value sizes compared to the baseline.

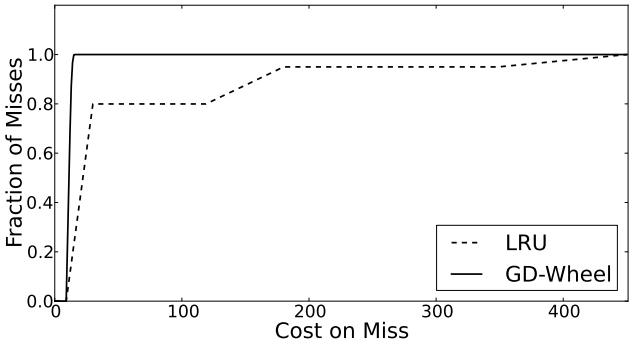

Fig. 2: CDF of Recomputation Costs.

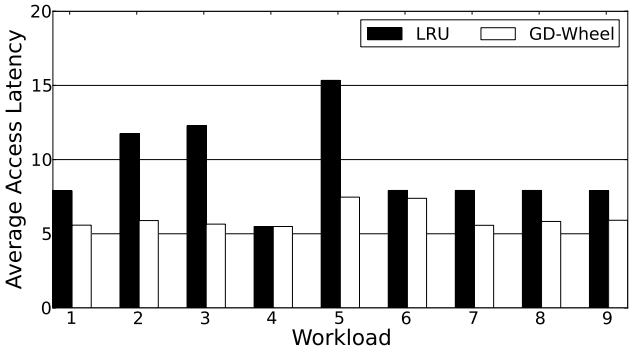

Fig. 3: Average Application Read Access Latencies.

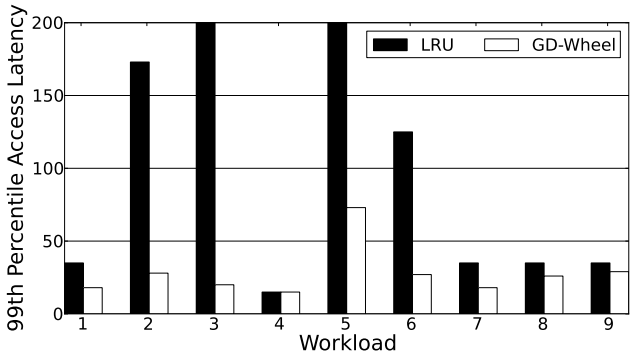

Fig. 4: 99th Percentile Application Read Access Latencies. LRU's Workload 3(374) and Workload 5(378) are cut off due to the space.

V. Experimental Results

A. Baseline Comparison

For workload 1 the hit rates for the two replacement policies are nearly the same, 95.064% for LRU and 95.055% for GD-Wheel. Overall, the hit rates achieved by LRU and GD-Wheel differ by no more than 0.07% under all workloads except workload 6 which will be discussed later. Table II shows the total recomputation costs for all workloads. For workload 1, GD-Wheel reduces 79.85% of the total recomputation cost compared to LRU. Figure 2 shows the CDF of recomputation costs for workload 1: all of the recomputations for GD-Wheel have a cost less than 16, while the recomputation costs for LRU span the cost distribution. The average response latency for GET and SET requests isn't affected by the replacement policy. In fact, the GET and SET request latency is almost identical under GD-Wheel (268.23 *us*/224.77 *us*) and LRU (267.4 *us*/221.72 *us*).

Figure 3 shows an estimation of the average application read access latency for each workload, including the extra latency for recomputations. In the figure, we assume each GET request has a response latency of 5 when accessing Memcached and use the recomputation costs as additional miss latency. The results show that GD-Wheel greatly reduces the average application read access latency, ranging from 0% to 53.95%. Figure 4 shows an estimate of the 99th-percentile application read access latency that also assumes a response latency of 5 in Memcached. The results show that GD-Wheel greatly reduces the 99th-percentile application read access latency, ranging from 0% to 94.65%, which is critical to large-scale Web services [8].

B. Different Cost Distributions

For workloads 2 and 3, GD-Wheel reduces the total recomputation cost by 86.75% and 91.05% compared to LRU. This shows that GD-Wheel is beneficial under the cost distributions of RUBiS and TPC-W. As would be expected, the two replacement policies have the same performance in workload 4, where all key-value pairs have the same cost. For workload 5 where the cost is totally random, GD-Wheel reduces the total recomputation cost by 76.10% compared to LRU.

C. Different Value Sizes

For workloads 6 to 9, GD-Wheel reduces the total recomputation cost by 18.25%, 80.42%, 71.48% and 68.52% compared to LRU. For workload 6, the hit rates for GD-Wheel (85.09%) and LRU (96.01%) differ a lot. This is because the 8-byte metadata overhead for GD-Wheel increases the size of each key-value pair, and this overhead forces GD-Wheel to use a slab class with fewer, larger chunks than LRU in workload 6. Consequently, in this case, fewer objects can be stored in the cache under GD-Wheel. By default, the chunk sizes differ by

a factor of 1.25 from one slab class to the next larger class. If we change this factor to 1.125, the difference between the hit rates for GD-Wheel (88.45%) and LRU (95%) is reduced. If, however, we change the factor to 1.375, the two replacement policies use the same slab class again and the hit rates for GD-Wheel (92.57%) and LRU (92.58%) are nearly the same. This shows that this performance inversion depends on both the key-value pair size and the chunk sizes. In addition, this performance inversion is more likely to happen with small key-value pair sizes. However, even with the performance inversion, GD-Wheel reduces the average and 99th percentile application read access latency for workload 6.

VI. Related Work

Many efforts have been made to improve key-value stores in terms of throughput and space efficiency. Masstree used a variation of B^+ trees to support range queries and applied optimizations for cache locality and optimistic concurrency control [12]. MemC3 used optimistic cuckoo hashing and CLOCK-based cache management to achieve high concurrency and space-efficiency [9]. To improve the throughput and reduce the CPU overhead of key-value stores, several works implement Memcached over RDMA on Infiniband [11], [13] or soft-iWARP [15]. To the best of our knowledge, our work is the first to propose a new replacement policy that takes cost variations into consideration.

VII. Conclusions

This paper has argued that a key-value store's replacement policy should take cost variations into consideration. Moreover, as a demonstration, it has introduced a new cost-aware replacement policy, GD-Wheel, which is an implementation of the GreedyDual algorithm with constant time complexity. In all of our experiments, GD-Wheel greatly reduced the total recomputation cost and improved the average application read access latency as compared to LRU. It also reduced the 99th percentile application read access latency which is critical in modern web applications. In future work, we plan to introduce cost-awareness into the rebalancing policy of Memcached.

References

[1] Redis. http://redis.io/.

[2] Tpc-w: a transactional web e-commerce benchmark. Transation Processing Performance Council. http://www.tpc.org/tpcw/.

[3] C. Amza, A. Chanda, A. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite. Specification and implementation of dynamic web site benchmarks. In *WWC-5*, pages 3–13, 2002.

[4] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS '12*, pages 53–64, 2012.

[5] S. Bouchenak, A. Cox, S. Dropsho, S. Mittal, and W. Zwaenepoel. Caching dynamic web content: designing and analysing an aspect-oriented solution. In *Middleware '06*, pages 1–21, 2006.

[6] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *USITS '97*, pages 18–18, 1997.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC '10*, pages 143–154, 2010.

[8] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[9] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI '13*, pages 371–384, 2013.

[10] B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, Aug. 2004.

[11] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur Rahman, H. Wang, S. Narravula, and D. K. Panda. Scalable memcached design for infiniband clusters using hybrid transports. In *CCGrid '12*, pages 236–243, 2012.

[12] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys '12*, pages 183–196, 2012.

[13] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efcient key-value store. In *USENIX ATC '13*, 2013.

[14] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *NSDI '13*, pages 385–398, 2013.

[15] P. Stuedi, A. Trivedi, and B. Metzler. Wimpy nodes with 10gbe: Leveraging one-sided operations in soft-rdma to boost memcached. In *USENIX ATC '12*, 2012.

[16] G. Varghese and T. Lauck. Hashed and hierarchical timing wheels: data structures for the efficient implementation of a timer facility. In *SOSP '87*, pages 25–38, 1987.

[17] N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, 1994.