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

Conglong Li [*]
Carnegie Mellon University
conglonl@cs.cmu.edu

Alan L. Cox
Rice University
alc@rice.edu

Abstract

Memory-based key-value stores, such as Memcached and Redis, are often used to speed up web applications. Specifically, they are used to cache the results of computations, such as database queries and dynamically generated web pages, so that a future request to the web application may not have to repeat the same computation. Currently, when memory-based key-value stores reach their capacity limits, they use replacement policies, like LRU and random, that are oblivious to differences among the cached results in their recomputation costs. However, this paper shows that if the costs of recomputing cached results vary significantly, as in the RUBiS and TPC-W benchmarks, then a cost-aware replacement policy will not only reduce the web application's total recomputation cost but also reduce its average response time.

To this end, this paper introduces *GD-Wheel*, which is an amortized constant-time implementation of the *GreedyDual* replacement algorithm that supports a limited range of costs. In effect, GD-Wheel integrates recency of access and cost of recomputation in an efficient manner. Moreover, this paper describes an implementation of GD-Wheel in Memcached, including the modifications to Memcached's interface so that web applications can include cost information with each key-value pair, and a new cost-aware slab rebalancing policy for Memcached's slab-based memory allocator. An evaluation of this implementation using the *Yahoo! Cloud Serving Benchmark* shows that GD-Wheel, when compared to LRU, reduces the total recomputation cost by as much as 90%. Moreover, GD-Wheel reduces the web application's average and 99th percentile latency to obtain the computed results by as much as 56% and 85%, respectively.

[*] This work was performed by Conglong Li while he was an M.S. student at Rice University.

EuroSys'15, April 21–24, 2015, Bordeaux, France.
Copyright © 2015 ACM 978-1-4503-3238-5/15/04…$15.00.
http://dx.doi.org/10.1145/2741948.2741956

1. Introduction

Low-latency access to large volumes of data has become critical for many web services. Given the latency of performing database queries and dynamic web page renderings, it is desirable to cache information in memory for faster reuse. Memory-based key-value stores allow for combining the distributed memory of different machines into a single, large pool. To support low-latency access to data, these memory-based key-value stores are used to implement distributed caches by many large-scale web applications. For instance, Memcached [15] is used at Facebook, Twitter and Zynga; and Redis [3] is used at GitHub, Flickr and Stack Overflow.

Memory-based key-value stores support two basic operations: GET for retrieving the value associated with the given key in the store, and SET for inserting a new key-value pair into the store. Since the underlying operations on current key-value store data structures that are used to implement GET and SET have constant time complexity, doing GET or SET also takes constant time regardless of the number of cached pairs. As a result, GET or SET have very low latencies. However, memory-based key-value stores cannot process an incoming SET request when the store is full unless a replacement is made.

Because of their fast response and finite size, memory-based key-value stores are usually used as database query caches, web page caches, or in general caches for any kind of computation results. By caching the results of computations in key-value stores, applications can avoid recomputing the same results and thereby reduce their read access latencies. Since these computations may have different purposes and even come from different sources, the results of computations cached in key-value stores may have different computation costs. In other words, it may take different amounts of time to recompute the different results cached in key-value stores. In fact, studies of real-world key-value store deployments [5, 24] and several representative web application benchmarks [10] provide evidence that significant cost variations do exist.

Because of computation cost variations, it could be desirable to retain key-value pairs with higher rather than lower recomputation costs. However, to take these costs into consideration when replacing key-value pairs, there are two problems that have to be addressed. The first is that key-value stores don't have the cost information available. They don't have the information available because this cost can only be defined by clients and measured outside the cache. Moreover, the current SET operation protocol does not provide an option for clients to include the cost information with a key-value pair. Thus, we need to provide a protocol that allows clients to include the cost information.

The second problem is that current replacement polices used by key-value stores don't take cost into consideration. For example, Memcached uses separate *Least Recently Used (LRU)* queues for objects of different sizes [1]. To increase space efficiency and concurrency in Memcached, Fan *et al.* have recently proposed a *CLOCK*-based approximation to LRU [14]. In contrast, Redis provides options for using either random or LRU-based replacement [2]. In summary, all of these replacement policies take constant time for making replacement decisions, but none of them take cost into consideration.

Instead of these existing polices, we need a replacement policy that takes cost into consideration and retains the key-value pairs with higher rather than lower cost. As a possible solution, the *GreedyDual* algorithm generalizes the LRU algorithm to address the weighted caching problem, which is the problem of making replacement decisions for objects with non-uniform costs [31]. To solve this problem, the GreedyDual algorithm integrates recency of access and cost of cached objects when making replacement decisions. In a nutshell, GreedyDual maintains a dynamic priority on each cached object that reflects its cost and recency of access, and replaces the object with the lowest priority. However, there is still a problem with using the GreedyDual algorithm in key-value stores. For either inserting or simply accessing an object, the time complexity of the state-of-the-art implementation of GreedyDual is logarithmic in the number of objects [11]. As key-value stores otherwise use constant-time algorithms to achieve fast GET and SET response times, a logarithmic complexity replacement policy is undesirable. In other words, we need a more efficient implementation of the GreedyDual algorithm.

To start, this paper argues that it can be beneficial for key-value stores to provide the option for clients to include cost information with SET requests, and take such cost variations into consideration when making replacement decisions. As a demonstration, this paper presents a new cost-aware replacement policy, *GD-Wheel*, which is an efficient implementation of the GreedyDual algorithm. GD-Wheel uses a data structure that we call *Hierarchical Cost Wheels* that is inspired by Varghese and Lauck's *Hierarchical Timing Wheels* [30]. In essense, through the use of Hierarchical Cost Wheels our GreedyDual implementation has time complexity that is a function of the number of priorities rather than cached objects. Moreover, when the number of priorities are limited to a predefined range, GD-Wheel can be shown to achieve amortized constant-time complexity per insertion or access operation. And, in fact, while the computation costs for key-value pairs in the TPC-W and RUBiS workloads vary, the maximum difference is only about a factor of twenty. Consequently, the computation costs can be effectively mapped onto a limited range of integers.

In addition, this paper describes a full implementation of GD-Wheel in Memcached, including our changes to the slab-based memory allocator and the SET request protocol. Under Memcached's slab-based memory allocator, key-value pairs of different sizes are stored in different *slab classes*, and each of these classes performs LRU replacement independently. Thus, we replaced each slab class's implementation of LRU replacement with GD-Wheel. To balance the memory allocated to different slab classes, Memcached performs periodic rebalancing that moves slabs between classes based on differences in the eviction rates among the slab classes. As an alternative to this policy, we also implemented a new cost-aware rebalancing policy.

We evaluated the performance of Memcached with GD-Wheel and the cost-aware rebalancing policy using the *Yahoo! Cloud Serving Benchmark (YCSB)* [12]. This evaluation consists of two parts. In the first part, we directly compare GD-Wheel and LRU using workloads with different cost distributions but key-value pairs that are of the same size. Consequently, all of the key-value pairs belong to the same slab class, so we avoid any side effects from the rebalancing policy. In contrast, in the second part of the evaluation, we use workloads where key-value pairs of different sizes have different costs, enabling us to compare Memcached's original rebalancing policy with our cost-aware policy. Overall, the results show that our approach, when compared to LRU and the original rebalancing policy, reduces the total recomputation cost by as much as 90%. In addition, our approach reduces the average and 99th percentile application read access latency by as much as 56% and 85%, respectively. Finally, we show that, when compared to the original GreedyDual implementation, GD-Wheel also reduces latencies and CPU overhead in Memcached.

The rest of this paper is organized as follows. Section 2 provides deeper motivation for our work. Section 3 presents the GD-Wheel replacement policy. Section 4 describes the implementation of GD-Wheel in the Memcached key-value store. Section 5 describes the original rebalancing policy in Memcached and presents our alternative cost-aware rebalancing policy. Section 6 describes our experimental methodology and presents our results. Finally, Section 7 discusses related work, and Section 8 summarizes our conclusions.

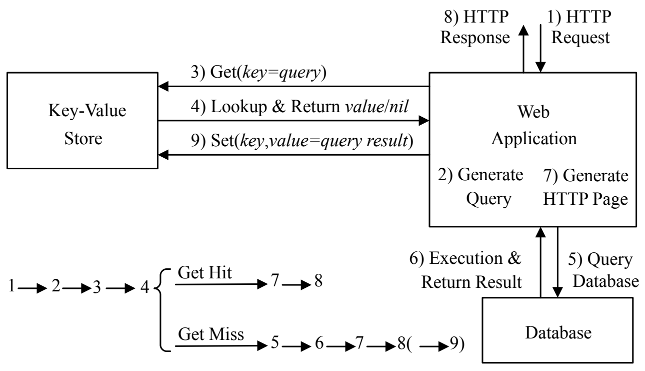


Figure 1: Using a Key-Value Store as a Database Query Cache.

2. Motivation

This section first explains how memory-based key-value stores are used by web applications. Then, it discusses the different sources of cost variation in the recomputation of key-value pairs.

2.1 How Key-Value Stores are Used

A web application, such as an online bookstore, may receive HTTP requests asking for book details, new books, best sellers, etc. The application then generates the necessary database queries to satisfy these requests and issues these queries to the database. After receiving the query results, the application generates the web page and sends the HTTP response. As described previously, there is a spectrum of what could be cached in a key-value store: it could be as low level as database query results; it could be as high level as web pages; or it could be both low and high level computations cached in the same store.

Figure 1 illustrates how a key-value store is used as a database query cache. After receiving a HTTP request (Step 1), the application generates the necessary database query (Step 2) and asks the key-value store for the query result by sending a GET request with the query as the key (Step 3). After receiving the GET request, the key-value store performs a lookup for the requested key and returns either the cached query result or a *not found* error (Step 4). If the query result is returned, the application skips to step 7 wherein it generates the web page. Finally, the application sends the HTTP response (Step 8). We refer to this case as a GET hit. If, however, a *not found* error is returned, then the application needs to request the database to execute the query (Steps 5&6). We refer to this case as a GET miss. After the execution in the database, the application usually sends a SET request to retain the query result for later use (Step 9).

As another use case, a key-value store could be used as a cache for dynamically generated web pages. After receiving a HTTP request, the application first asks the key-value store for the web page. On a GET hit, the application simply returns the cached web page. However, on a GET miss, the application needs to generate the web page, which may involve extensive computation, including database queries.

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 recomputation of previously cached values. For database query results, the recomputation cost is the sum of steps 5 and 6 in Figure 1. For web pages, the recomputation cost is the sum of step 2 and steps 5 through 7 in Figure 1. Current key-value stores don't provide the option for clients to include such recomputation cost information in SET requests.

2.2 Cost Variations in Recomputations

There exist two common sources of variation in the computation costs of key-value pairs. The first source is simultaneously caching different types of objects from different levels in the application stack. As described above, a key-value store could cache both low level query results and high level web pages. These different types of objects will likely have different recomputation times, and thus different costs.

As a real-world example, Twemcache, a version of Memcached deployed by Twitter, has two use cases: First, Twemcache stores recently and frequently accessed Tweets to reduce the frequency of database accesses. Second, Twemcache also stores recently rendered Tweets, which includes computed metadata, such as the number of retweets and favorites [5].

Similarly, Facebook uses Memcached for multiple purposes [24]. For example, Facebook uses Memcached as a query cache to lighten the read load on databases. More generally, Facebook leverages Memcached for caching various computation results, such as the results of sophisticated machine learning algorithms that are used by a variety of applications. To accommodate the cache miss cost variations, Facebook partitions a cluster's Memcached servers into separate pools. There is a default general pool and some smaller pools for key-value pairs that are accessed frequently but have an inexpensive cache miss cost. Although this approach of separating key-value pairs with different costs doesn't require any changes to Memcached, it does require prior or dynamic usage analysis to determine the exact size of each pool. If the workload characteristics change over time, such partitioning may result in inefficient usage of memory. It could be more efficient to maintain a single pool and make replacement decisions based on the recomputation cost variations.

The second source of cost variations comes from objects of the same type or at the same level in the application stack. Previous work on some web application benchmarks has shown that even objects at the same level can have widely varying costs. Bouchenak *et al.* implemented a web page cache above two web application benchmarks: *RUBiS* and *TPC-W* [10].

RUBiS implements the core functionality of an auction site modelled after eBay [6]. It defines various interac-

	RUBiS	TPC-W
Low Cost (Proportion)	10 *ms* (17%)	10 - 25 *ms* (48%)
Mid Cost (Proportion)	60 - 95 *ms* (79%)	45 - 150 *ms* (25%)
High Cost (Proportion)	240 *ms* (4%)	210 - 300 *ms* (27%)

Table 1: Extra Response Times on Cache Misses.

tions, including browsing, bidding, buying or selling items. Bouchenak *et al.* measured the extra response time on cache misses for different interactions. For simple interactions, such as browsing items, the extra response time is as low as 10*ms*. On the other hand, the cache miss cost is as high as 240*ms* for complicated interactions, such as showing user information, which includes buying and selling history.

TPC-W is a web server and database performance benchmark that simulates an online bookstore [4]. It defines interactions including listing new products and best sellers, updating the shopping cart, ordering, *etc.* Bouchenak *et al.* measured the extra response time on cache misses. For some simple interactions, such as displaying orders and showing product detail, the extra response time is as low as 10*ms* to 25*ms*. On the other hand, the cache miss cost is as high as 210*ms* to 300*ms* for complicated interactions such as showing best sellers and executing searches.

To summarize, Table 1 categorizes the cache misses in RUBiS and TPC-W based upon the extra time they took to respond. Specifically, we categorize the cache misses into three groups, where the ratio of response times between the groups is roughly 1:7.5:20. On one hand, this analysis shows that there do exist substantial variations in the execution times for different interactions. On the other hand, the variations aren't so large that they couldn't be mapped onto a limited range of cost values.

More generally, the following intuitive argument can be made for the sufficiency of a limited range of cost values for web applications. At the low end, as the computation cost shrinks, it's no longer worth caching the value; recomputing it is cheaper than requesting it from the key-value store. At the high end, there are limits to how long web application developers will want to make users wait, even in the worst case. So, arbitrarily long computations will not be performed in the course of an interaction.

3. GD-Wheel Replacement Policy

Based on the observed cost variations in key-value store workloads, this paper proposes the GD-Wheel replacement policy, which is an efficient implementation of the GreedyDual algorithm. This section first introduces the GreedyDual algorithm. Then, it describes the implementation of the GD-Wheel replacement policy using the Hierarchical Cost Wheels structure. Finally, it discusses GD-Wheel's time complexity.

Algorithm 1 The GreedyDual Algorithm under Cao *et al.*'s implementation

```
Initialize $M \leftarrow \varnothing$ and $L \leftarrow 0$
For each requested object $p$
if $p$ is already in memory
    $H(p) \leftarrow L + c(p)$
    Update the position of $p$ in priority queue
if $p$ is not in memory
    while there is not enough room in memory for $p$
        $M \leftarrow \{a \mid a \in memory \text{ and } H(a) = min\{H(b) \mid b \in memory\}\}$
        Evict the least recently used object $q$ in $M$
        Let $L \leftarrow H(q)$
    $H(p) \leftarrow L + c(p)$
    Insert $p$ into priority queue
end
```

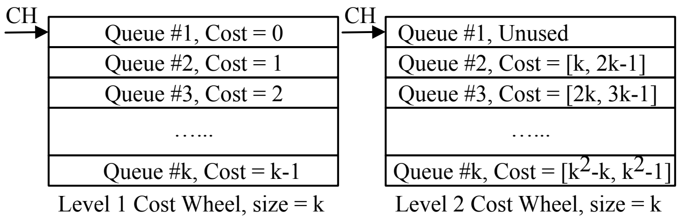
3.1 GreedyDual Algorithm

Young *et al.* introduced the GreedyDual algorithm as a primal-dual strategy for solving the weighted caching problem, which is the problem of making replacement decisions for items with non-uniform costs [31]. This algorithm is of practical interest because it generalizes the LRU algorithm, integrating recency of access with the cost of cached objects when making replacement decisions.

The original implementation of the GreedyDual algorithm associates a value, H, with each cached object. On insertion or reuse of an object p, $H(p)$ is set to be the cost of bringing the object into the cache $c(p)$. On eviction, the object q with the lowest H value $H(q)$ is evicted. Then all cached objects reduce their H values by $H(q)$. Since objects with higher cost as well as recently inserted or reused objects have higher H values, the GreedyDual algorithm seamlessly integrates recency of access and cost of cached objects in making replacement decisions. Since, as described, an eviction requires a subtraction on every cached object, this implementation requires $O(n)$ time complexity for each eviction, where n is the total number of cached objects.

A few years later, Cao *et al.* introduced a new implementation with reduced time complexity [11]. Described in Algorithm 1, their implementation uses a single priority queue to store the priority of each cached object. As before, each cached object has an associated priority value H. However, instead of doing subtractions on all cached objects on evictions, the priority queue uses a global inflation value L. On insertion or reuse of an object p, $H(p)$ is set to $L + c(p)$ where L is the current inflation value and $c(p)$ is the cost of p. On eviction, the object with the lowest H in the queue is evicted. If multiple objects have the lowest H value, the least recently used one will be evicted. Then the global inflation value L is updated to the H value of the evicted object.

By introducing the global inflation value, Cao *et al.*'s implementation of the algorithm takes only $O(\log n)$ time for handling an insertion, a reuse, or an eviction. However, when the global inflation value is about to overflow, a scan of the priority queue that takes $O(n)$ time is required to reduce the inflation value. Although Cao *et al.*'s implemen-



CH
Queue #1, Cost = 0
Queue #2, Cost = 1
Queue #3, Cost = 2
……
Queue #k, Cost = k-1
Level 1 Cost Wheel, size = k
CH
Queue #1, Unused
Queue #2, Cost = [k, 2k-1]
Queue #3, Cost = [2k, 3k-1]
……
Queue #k, Cost = [k2-k, k2-1]
Level 2 Cost Wheel, size = k

Figure 2: A Two-Level Hierarchical Cost Wheels. CH = Clock Hand.

tation reduces the complexity to $O(\log n)$ time per operation, this is still more costly than the constant-time policies currently used by key-value stores. Thus, if we are to maintain the constant-time operation that key-value stores currently achieve for GET and SET requests, then we need a more efficient implementation of the GreedyDual algorithm with constant time complexity per operation.

3.2 GD-Wheel Replacement Policy

3.2.1 Hierarchical Cost Wheels

Since the GreedyDual algorithm requires a priority for each object, it seems impossible to build an implementation with constant complexity. However, if we can restrict the priority range, constant time complexity is achievable.

To reduce the time complexity, GD-Wheel uses a data structure that we call *Hierarchical Cost Wheels* that are inspired by Varghese and Lauck's *Hierarchical Timing Wheels* [30]. As shown in Figure 2, this structure is made up of a series of *Cost Wheels*. Each Cost Wheel is basically an array of queues with a clock hand pointing to one of its queues. By using separate queues for objects with different priorities, this structure changes the time complexity from being a function of the number of objects to being a function of the number of priorities. The time complexity would be logarithmic if not for the fact that we limit the range of priorities supported by the Hierarchical Cost Wheels. By using a fixed number of Cost Wheels in a hierarchy and a fixed number of queues in each Cost Wheel, the Hierarchical Cost Wheels structure can support a limited yet sufficient priority range to express the cost variations found in key-value store workloads.

A Single Cost Wheel A single Cost Wheel (Level 1 Cost Wheel in Figure 2) has the same functionality as a priority queue. The only difference is that a Cost Wheel only supports k different priorities, where k is the number of queues in each Cost Wheel configured at initialization. Acting like the global inflation value in Cao *et al.*'s implementation, the clock hand advances when searching for eviction candidates. Instead of storing all objects in a single priority queue, objects are stored in different queues. The queue is selected for each inserted object by adding the object's cost to the current position of the clock hand. For an object with cost c and the clock hand pointing to the x^{th} queue, the object will be inserted into the $((c+x) \bmod k)^{\text{th}}$ queue. As a result, objects with different priorities are stored in different queues.

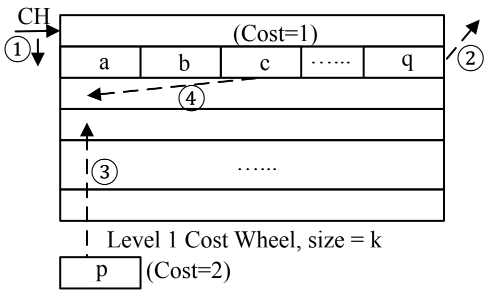
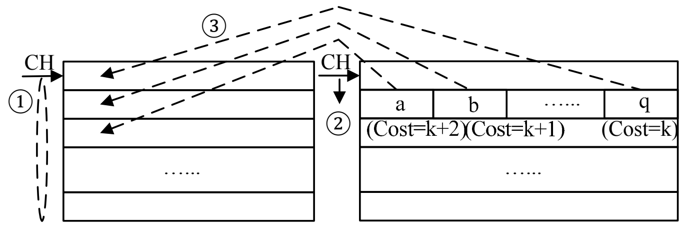


Figure 3: Handling an Eviction(object q), an Insertion(object p), and a Reuse(object c) in a Single Cost Wheel.

Figure 3 illustrates how evictions, insertions, and reuse are handled in a single Cost Wheel. Suppose object p is being inserted. If there is not enough free memory to hold p, then one or more objects must be evicted. To select these objects, the clock hand advances until a non-empty queue is found (Step 1). Then the object q at the tail of that queue is evicted (Step 2). Once enough objects have been evicted to allow p to be instantiated, p will be inserted at the head of the queue selected by adding p's cost to the current position of the clock hand. As shown in Figure 3, because the clock hand is currently pointing to the 2nd queue, the object p with cost 2 will be inserted in the 4th queue (Step 3). When an object is reused, the position of the object is updated based on the current position of the clock hand. As shown in Figure 3, the object c with cost 1 is reused. Then the object c is removed from the 2nd queue and inserted at the head of the 3rd queue, since the clock hand is currently pointing to the 2nd queue (Step 4). In the case when the clock hand's position is unchanged, the reused object will be removed from the queue and inserted at the head of the same queue.

Hierarchical Cost Wheels Since a single Cost Wheel's size k is fixed at run-time, a single Cost Wheel can only support up to k costs. To extend the range of costs in an efficient manner, we use a fixed but configurable number of Cost Wheels in a hierarchy such that each higher level Cost Wheel supports a larger range of costs. As shown in Figure 2, each queue in the level 1 Cost Wheel only supports a single cost, while each queue in the level 2 Cost Wheel supports k different costs. In general, each queue in a level x Cost Wheel will support k^{x-1} different costs, where k is the number of queues in the Cost Wheel. Thus, a key-value store using GD-Wheel will be configured at initialization time to utilize a sufficient number of Cost Wheels to support the desired cost range.

Hierarchical Cost Wheels act the same as a single Cost Wheel on insertions and reuse. Objects are inserted or moved to the appropriate queue by adding the object's cost to the current position of the clock hands. Objects will be evicted from the lowest level Cost Wheel, not from the higher level



Level 1 Cost Wheel, size = k   Level 2 Cost Wheel, size = k

Figure 4: A Migration in the Higher Level Cost Wheel.

Cost Wheels, because by definition objects stored in the higher level Cost Wheels have higher priorities. The clock hand in the lowest level Cost Wheel keeps advancing until a non-empty queue is found. On the other hand, the clock hands in the higher level Cost Wheels advance to the next queue when the next lower level Cost Wheel's clock hand has completed a whole round. This is similar to the relationship between second, minute, and hour hands of an analog clock.

After a clock hand in a higher level Cost Wheel advances, a migration is performed between that Cost Wheel and the next lower level Cost Wheel. During this migration, objects stored in the queue pointed to by the higher level clock hand will be migrated to the corresponding queues in the next lower level Cost Wheel. In effect, a migration realizes the reduction of priorities for these objects. An object with $H = a + bk$ ($0 < a, b < k$, where k = number of queues in each Cost Wheel) will be migrated to the level 1 Cost Wheel's $(a+1)^{\text{th}}$ queue when the clock hand in the level 1 Cost Wheel has completed b rounds. The object is migrated to the $(a+1)^{\text{th}}$ queue because the clock hand in the lower level Cost Wheel will always point to the 1st queue when migration happens.

Figure 4 illustrates how migration is performed. After the clock hand in the level 1 Cost Wheel has completed a whole round (Step 1), the clock hand in the level 2 Cost Wheel advances to the next queue (Step 2). Then the objects pointed to by the level 2 Cost Wheel's clock hand will be migrated to the corresponding queues in the level 1 Cost Wheel (Step 3): object a with cost $k+2$ is moved to the 3rd queue; object b with cost $k+1$ is moved to the 2nd queue; and object q with cost k is moved to the 1st queue in the level 1 Cost Wheel. If the migrated object is reused, it will "jump" back to a higher level Cost Wheel since it has higher priority again.

3.2.2 GD-Wheel Time Complexity Analysis

Algorithm 2 summarizes GD-Wheel's approach to the implementation of the GreedyDual algorithm. It's different from the original algorithm because of the Hierarchical Cost Wheels data structure. In this section, we will argue that this implementation achieves amortized constant-time complexity per operation if the priority range is limited.

Algorithm 2 The GreedyDual Algorithm under GD-Wheel's implementation

```
Let $NW \leftarrow$ number of Cost Wheels
Let $NQ \leftarrow$ number of queues in each Cost Wheel
Let $C[NW] \leftarrow$ array of ones        //Clock hands start from the 1st queues
For each requested object $p$
if $p$ is already in memory
    Remove $p$
    $W \leftarrow max\{i \mid 0 < i \le NW$ and $round(c(p)/NQ^{(i-1)}) > 0\}$
    $Q \leftarrow (round(c(p)/NQ^{(W-1)}) + C[W]) \bmod NQ$
    Insert $p$ to the head of $Q^{\text{th}}$ queue in the level $W$ Cost Wheel
if $p$ is not memory
    while there is not enough room in memory for $p$
        $C[1] \leftarrow$ index of next non-empty queue in level 1 Cost Wheel
        Evict $q$ at the tail of the $C[1]^{\text{th}}$ queue in level 1 Cost Wheel
        if $C[1]$ has advanced a whole round back to 1, call migration(2)
    $W \leftarrow max\{i \mid 0 < i \le NW$ and $round(c(p)/NQ^{(i-1)}) > 0\}$
    $Q \leftarrow (round(c(p)/NQ^{(W-1)}) + C[W]) \bmod NQ$
    Insert $p$ to the head of $Q^{\text{th}}$ queue in the level $W$ Cost Wheel
end

Function migration(idx)
$C[idx] \leftarrow (C[idx] + 1) \bmod NQ$
if $C[idx]$ has advanced a whole round back to 1, call migration(idx+1)
For each object $p$ in the $C[idx]^{\text{th}}$ queue in the level $idx$ Cost Wheel
    Remove $p$
    $Cost\_Remainder \leftarrow c(p) \bmod NQ^{(idx-1)}$
    $Q \leftarrow (round(Cost\_Remainder/NQ^{(idx-2)}) + C[idx-1]) \bmod NQ$
    Insert $p$ to the head of $Q^{\text{th}}$ queue in the level $(idx-1)$ Cost Wheel
end
```

In general, an object is first inserted into the cache, then reused zero or more times, and finally evicted. Since each queue is implemented by a doubly linked list, removing or inserting a given node requires only constant time. An insertion of a given object requires one list insertion. A reuse of a given object requires one list removal and one list insertion. Thus handling either an insertion or a reuse of an object takes only $O(1)$ time in the worst case.

After initialization, the number of Cost Wheels NW and the size of each Cost Wheel NQ are fixed. An eviction of an object from the lowest level Cost Wheel requires advancing the clock hand to the next non-empty queue and removing the object at the tail of the queue. Advancing the clock hand takes constant time since we have a fixed number of queues in a Cost Wheel. Thus an eviction takes constant time for objects in the lowest level Cost Wheel. However, a migration from the next level Cost Wheel will be performed if the clock hand has completed a whole round. It's true that migrating the objects in a queue would take $O(n)$ time in the worst case if all the cached objects are stored in that queue. However, we show that if each migrated object is charged for the cost of its migrations, the overall algorithm achieves an amortized constant-time complexity over a sequence of operations.

A migration of an object requires its removal from the queue in the higher level Cost Wheel and an insertion to the queue in the lower level Cost Wheel. Thus one migration of one object takes constant time. Considering a sequence of

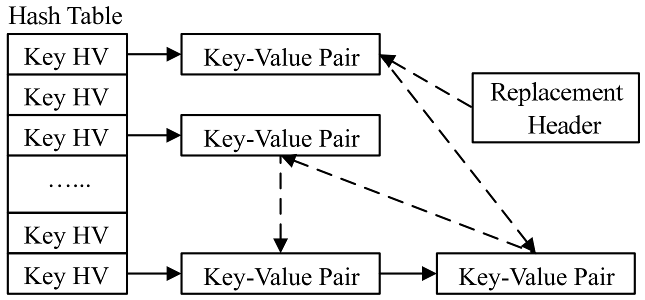


Figure 5: Basic Components of Memory-Based Key-Value Stores. HV = Hash Value.

operations for an arbitrary object, the cost per operation, including the cost of migrations between operations, is amortized constant time. Because a sequence of operations on an object always starts from an insertion and ends with an eviction, there are two cases where migrations could happen: between an insertion/reuse and a reuse; or between an insertion/reuse and an eviction.

After an insertion or a reuse of an object with high cost, the object will be inserted into the higher level Cost Wheel. A migration can only move objects in one direction, from the higher level Cost Wheels to the lower level Cost Wheels. Since there are a fixed number of Cost Wheels at run-time, there will be a constant number of migrations of an object until an operation removes the object from the lower level Cost Wheel. Until the next reuse or the eviction of the object, there could only be a constant number of migrations. Thus the migrations between an insertion/reuse and a reuse/eviction have constant cost. Any further migrations after the second reuse will be charged to that reuse operation.

4. Implementation

This section first introduces the relevant components of memory-based key-value stores. Then it provides a more detailed introduction to the Memcached key-value store. Finally it describes the implementation of GD-Wheel in Memcached.

4.1 Relevant Components of Memory-Based Key-Value Stores

Figure 5 illustrates the relevant components of memory-based key-value stores: an index of the key-value pairs and the replacement data structure. The index consists of a hash table which maps the hash value of a requested key to the location where the key-value pair is stored in memory. The replacement data structure is used to make eviction decisions when there is not enough room for the new key-value pair being inserted.

Each cached key-value pair has metadata which includes the key/value size, expiration time, and linked list pointers which point to the previous and next key-value pair's metadata in the hash table and the replacement data structure.

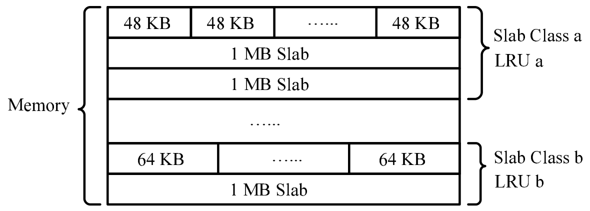


Figure 6: Memcached's Memory Allocation and Replacement

Thus the actual memory allocated for each key-value pair is based on the total size of the key, value, and metadata. The hash table maps each hash value to a chain of key-value pairs' metadata. The replacement data structure uses a linked list to link all key-value pairs and to determine the eviction priority for each key-value pair.

When a GET request is sent to the key-value store, a hash value is computed based on the requested key. Then the hash table is used to map this hash value to a chain of key-value pairs that might hold the requested key. After locating the requested key-value pair in that chain, the key-value store will first return the value and later update the position of the key-value pair inside the replacement data structure.

When a SET request is sent to the key-value store, the memory allocator first checks if there is enough memory to store the key-value pair. If not, the replacement data structure will be used to decide which key-value pair to evict. After the memory used by that key-value pair is freed, the new key-value pair is allocated and inserted into the hash table and the replacement data structure.

4.2 Memcached Key-Value Store

Memcached is a high-performance distributed-memory object caching system. Its simple design promotes quick and easy deployment. In addition to the basic operations GET and SET, Memcached also supports operations like DELETE and REPLACE.

Figure 6 illustrates Memcached's memory allocation and replacement designs. Memcached uses a slab allocator for memory allocation. This allocator divides the available memory into 1 MB slabs and uses different slab classes to store key-value pairs belonging to different size ranges. 1 MB slabs are distributed to different slab classes based on the number of key-value pairs stored in each class. Each slab class has an unique chunk size. If a slab class stores objects of sizes from x to y ($x \leq y$), the slabs belonging to the class are divided into chunks of size y. The chunk sizes differ by a growth factor, which is by default 1.25.

Each slab class has its own LRU queue for replacement. In other words, only the objects stored in the same slab class as the object being inserted will be considered for replacement. If there are no free chunks, and no free slabs for that slab class, Memcached will look at the tail of the slab class's LRU queue for an object to reclaim. In Memcached, objects can have an expiration time. Before evicting an unexpired object from the tail of the LRU queue, Memcached will first look at a fixed number of objects near the tail of the LRU queue to see if an expired object can be reclaimed instead.

4.3 Implementation in Memcached

Our implementation replaces each slab class's LRU mechanism with GD-Wheel. By default, it uses two level of Hierarchical Cost Wheels, where each Cost Wheel has 256 queues. Thus our GD-Wheel implementation supports 256^2+256 = 65792 different costs by default, which is more than enough to support the cost variations found in RUBiS and TPC-W. To the metadata of each key-value pair, we add an additional field for the cost information. This additional field increases the metadata size by 2 bytes for each stored key-value pair. However, since the allocated metadata size is rounded up to an 8-byte boundary to avoid fragmentation, this additional field doesn't increase the allocated size of each metadata. The SET request protocol is modified so that clients are able to optionally send cost information with each key-value pair to the Memcached server.

5. Rebalancing Policy in Memcached

This section first describes how the original rebalancing policy in Memcached works and discusses its deficiencies. Then we present our alternative cost-aware rebalancing policy.

5.1 The Original Rebalancing Policy in Memcached

Since memory is distributed to different slab classes and each slab class has its own replacement structure, the eviction rates between different slab classes can vary significantly. To balance the memory allocated to each slab class, Memcached has a rebalancing policy that periodically moves slabs between slab classes based on the eviction rates for the different slab classes. This policy checks the eviction rate of each slab class 3 times per 30 seconds. If a slab class has the highest eviction rate when all 3 checks were performed, it will take the least recently used slab from a slab class which had zero evictions in the last 30 seconds.

The original rebalancing policy in Memcached aims to balance the memory allocated to each slab class based on the eviction rates between slab classes. However, this policy is very conservative and in many cases ineffective. First, the policy only takes a slab from a slab class which has had zero evictions in the last 30 seconds. Suppose there is a slab class with more chunks and a lower eviction rate, and a slab class with fewer chunks and a higher eviction rate. It might be beneficial to move slabs from the class with the lower eviction rate to the class with more evictions. Second, this policy only moves at most one slab per 30 seconds. If the chunk size is large, moving a single slab is far less than needed. In addition, making periodic decisions might be too ”lazy” considering the fast response times of GET and SET.

Given recomputation cost variations, we argue that a rebalancing policy should also take cost information into consideration. Different slab classes may store key-value pairs with much different recomputation costs. If a slab class stores key-value pairs with higher average cost than any other slab classes, it could be preferable to avoid evictions from this slab class first.

5.2 The Cost-Aware Rebalancing Policy

As an alternative solution, we implemented a new cost-aware rebalancing policy in Memcached. Each slab class maintains an average cost per byte. Our rebalancing policy moves slabs from the class with the lowest average cost to the classes with higher average costs. We kept a field in Memcached which remembers the slab class id that has the lowest average cost. This field is updated when the average cost information is changed in any slab class. This update takes constant time since there are fixed number of slab classes in Memcached.

Instead of making periodic decisions, our rebalancing policy reacts immediately on evictions. When an eviction occurs in a slab class with higher average cost, our rebalancing policy will move a certain number of least recently used slabs from the class with lowest cost to the slab class suffering the eviction. The number of slabs moved is determined by the size of the evicted key-value pair. More slabs will be moved if the evicted key-value pair is large and vice versa. By possibly preventing further evictions in slab classes with higher average cost, our rebalancing policy complements the cost-awareness inside each slab class.

In addition to the implementation of the GD-Wheel replacement policy in Memcached, we replaced the original rebalancing policy with our cost-aware rebalancing policy. Since the cost-aware rebalancing policy requires the cost information for each key-value pair, it cannot collaborate with LRU. In the following evaluation, we will compare GD-Wheel combined with the cost-aware rebalancing policy with LRU combined with the original rebalancing policy.

6. Evaluation

This section first discusses the real-world key-value store workload characteristics on which our experimental workloads are based. Then we describe the evaluation environment and the YCSB Benchmark. This is followed by a description of our workloads and our experimental results.

Our evaluation has two primary goals. First, we want to directly compare GD-Wheel with the LRU replacement policy. We fulfill this goal with our study using single-size workloads, i.e., workloads that use a single size for all key-value pairs. Since only a single size is used, all key-value pairs are stored in one slab class. This avoids side effects from the rebalancing policy. Our second goal is to evaluate our cost-aware rebalancing policy in Memcached. We fulfill this goal with our study using multiple-size workloads, i.e., workloads that use different sizes and therefore different slab classes for key-value pairs with different costs. We show that

GD-Wheel together with cost-aware rebalancing can greatly improve the performance of the application.

To show that logarithmic time complexity in the replacement data structure would, in fact, affect GET/SET request latencies, we also reproduced Cao *et al.*'s implementation of the GreedyDual algorithm. We call this implementation *GD-PQ*. It maintains all key-value pairs for a slab class in a single priority queue. We will compare GD-PQ with GD-Wheel and LRU in terms of GET/SET latencies in Memcached.

6.1 Real-World Workloads of Key-Value Stores

Recently, Atikoglu *et al.* [8] and Nishtala *et al.* [24] have provided a detailed picture of how Facebook uses Memcached. Atikoglu *et al.* reported the characteristics of five Memcached pools sampled at Facebook. Nishtala *et al.* recorded all Memcached operations for a small percentage of user requests at Facebook. We based the workloads used in the latter part of our evaluation on these characteristics.

Key/Value Sizes Small keys and values dominate in all workloads. However, there exist large size variations among 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 returned values from Memcached GET requests have a median size of 135 bytes and a mean size of 954 bytes.

GET-to-SET Ratio All workloads are GET intensive. Atikoglu *et al.* reported that most of the Memcached pools at Facebook are GET-dominated and the GET-to-SET ratio was 30:1 for the pool 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 affected by the GET miss rate.

Miss Rate There are two kinds of GET misses in key-value stores: cold misses and capacity misses. Cold misses are inevitable at warmup. On the other hand, GD-Wheel helps to avoid the capacity misses with higher recomputation costs. Atikoglu *et al.* reported that the mean GET hit rate over the entire trace ranged from 81.4% to 98.7% across different Memcached pools. For the pool most representative of general cache usage, the hit rate was only 81.4%. Among those misses, 22% were capacity misses. This means that 4.1% of GET requests resulted in capacity misses. Considering the approximately 55,000 request/sec rate of accesses to that pool, the costs of these capacity misses greatly affect the read access latencies.

Requests Distribution The requested keys on GET requests follow a *Zipf* distribution. Atikoglu *et al.* reported that most properties of the user requests can be modeled using power-law distributions (Zipf's law) for the Memcached pool most representative of general cache usage. About 50% of key-value pairs were accessed in only 1% of requests.

6.2 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 32 GB of DRAM. One machine acts as the Memcached server and the other acts as its client. We configured Memcached with different cache sizes ranging from 10 GB to 25 GB, and we used the 25 GB cache size for most of the experiments. We also configured Memcached with 8 threads, and one of the LRU, GD-Wheel, and GD-PQ replacement policies. On the client machine, we use the YCSB Benchmark to generate GET and SET requests [12]. When the GD-Wheel or GD-PQ replacement policy is used, the client includes the cost of the key-value pair in the SET requests to Memcached.

The YCSB Benchmark is a load-generating tool that generates GET and SET requests based on a specified workload configuration. We divide each experiment into 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 to keep the hit rate during the measurement phase at about 95% for LRU. Then we use the same number of SET requests in the warmup phase for GD-Wheel for a fair comparison. Since the cold misses in the warmup phase are not included in the measurements, our evaluation focuses on the benefits of GD-Wheel on capacity misses. We aimed for 5% capacity misses, which is comparable to the rate of capacity misses in Facebook's Memcached servers.

During the measurement phase, each workload generates 100 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 LRU workload's measurement phase will send 100 million GET requests and about 5 million SET requests, for a GET-to-SET ratio of about 20:1.

6.3 Workloads

Table 2 shows our single-size 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 both the cost groups and 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 are derived from the baseline but use different value sizes. Workload 10 deviates from the baseline by using a coarser cost distribution where all costs

Workload	Key/Value Size (bytes)	Cost Distribution
1. Baseline	16 / 256	10-30(80%);120-180(15%);350-450(5%)
2. RUBiS	16 / 256	10-30(20%);120-180(75%);350-450(5%)
3. TPC-W	16 / 256	10-30(50%);120-180(25%);350-450(25%)
4. Same	16 / 256	10(100%)
5. Random	16 / 256	20-400(100%)
6. Small_1	16 / 64	10-30(80%);120-180(15%);350-450(5%)
7. Small_2	16 / 128	10-30(80%);120-180(15%);350-450(5%)
8. Big_1	16 / 2048	10-30(80%);120-180(15%);350-450(5%)
9. Big_2	16 / 4096	10-30(80%);120-180(15%);350-450(5%)
10. Coarse	16 / 256	(1-3)*10(80%);(12-18)*10(15%);(35-45)*10(5%)

Table 2: Single Size Workload Configurations.

Workload	Key/Value Size (bytes)	Cost Distribution
1. Baseline	16 / (192/256/320)	10-30(80%);120-180(15%);350-450(5%)
2. RUBiS	16 / (192/256/320)	10-30(20%);120-180(75%);350-450(5%)
3. TPC-W	16 / (192/256/320)	10-30(50%);120-180(25%);350-450(25%)

Table 3: Multiple Size Workload Configurations.

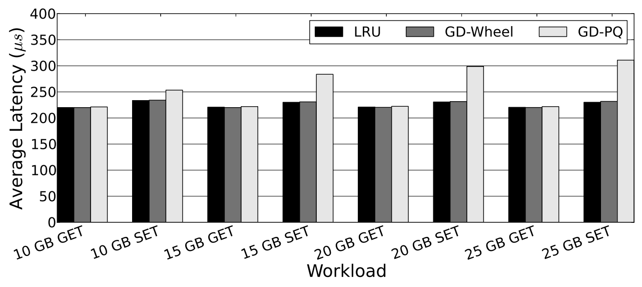


Figure 7: Average GET/SET Request Latencies (μs) for the Baseline Single Size Workload.

are multiples of 10. This workload evaluates the sensitivity of GD-Wheel's results to the precision of cost distribution.

Table 3 shows our multiple-size workloads. As summarized in the table, we use the same cost variations as the first three single-size workloads. The difference is that now we are using different value sizes for the three cost groups. The higher the cost, the larger the value size. We select these three value sizes so that the key-value pairs in different cost groups will fall into different slab classes.

6.4 Results

6.4.1 Single Size Workload Results

Average GET/SET Request Latencies and Overall Throughput in Memcached Figure 7 shows the average GET/SET request latencies for the baseline workload with different replacement policies and different Memcached cache sizes. The average GET request latencies are all about 220 μs. When dealing with GET requests, Memcached will send the return value right after the hash table lookup. Changing the

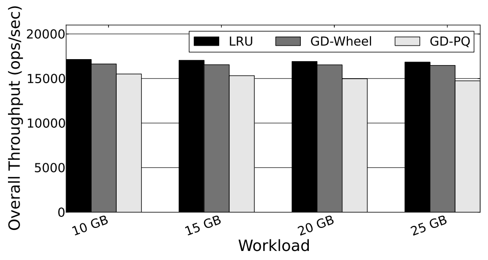


Figure 8: Overall throughput (ops/sec) for the Baseline Single Size Workload.

priority of the requested key-value pair in the replacement data structure happens after sending the GET response. Thus the complexity of the replacement policy won't affect the GET request latency. However, the complexity will still affect the CPU usage in the Memcached server. On the other hand, the average SET request latencies for GD-PQ keep increasing as the Memcached cache size increases. In contrast, the average SET latencies for LRU and GD-Wheel are the same under different cache sizes. This difference is because the complexity of the replacement policy affects the SET request latency. Compared to the constant time complexity of LRU and GD-Wheel, GD-PQ's logarithmic time complexity makes the SET request latency increase as cache size increases.

Figure 8 shows the overall throughput of the Memcached server for the baseline workload with different replacement policies and cache sizes. Compared to LRU, GD-Wheel introduces a throughput reduction of about 2% that is constant across the different cache sizes. This overhead is mainly introduced by the extra data structure manipulation for the Hierarchical Cost Wheels. On the other hand, GD-PQ exhibits throughput reduction that keeps increasing as cache size increases (from 9.5% to 12.5%). This indicates that GD-PQ introduces increasing CPU overhead on both GET and SET requests. As the above results show that LRU and GD-Wheel outperform GD-PQ, we will only present results for LRU and GD-Wheel in the rest of this evaluation. Also, we will only use the 25 GB cache size. We did, however, perform all of the following experiments on GD-PQ and the replacement decisions made by GD-PQ were exactly the same as GD-Wheel. It's just the latency and throughput in Memcached that are different.

Average Read Access Latency Since there is no database layer in our experiments, we calculate the overall application read access latency as follows. We use the average GET request latency (220 μs) measured by YCSB as the cache hit latency, and we use the recomputation cost as the additional miss latency. For the smallest recomputation cost, 10, we represent it as twice the hit latency (440 μs). As a result, each unit of cost represents 44 μs and we represent other recomputation costs based on this rate.

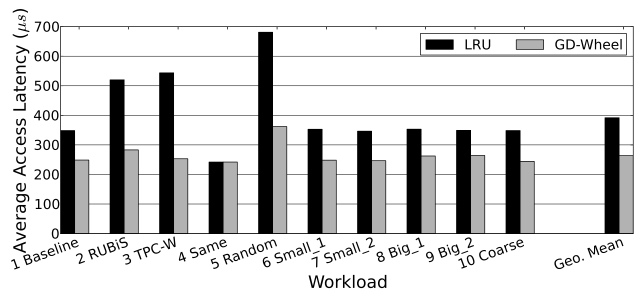


Figure 9: Average Application Read Access Latencies (μs) for the Single Size Workloads.

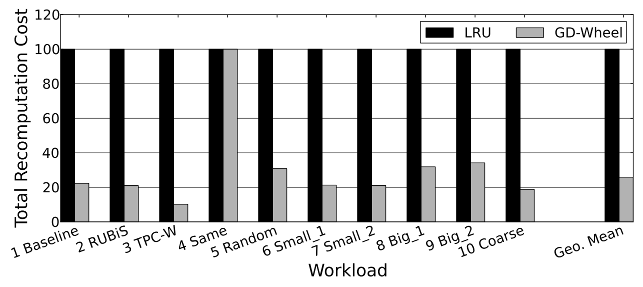


Figure 10: Normalized Total Recomputation Cost for the Single Size Workloads.

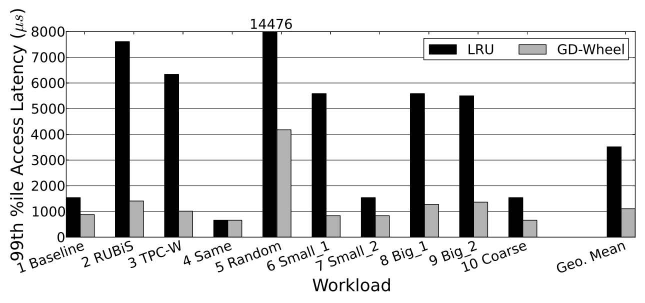


Figure 11: 99th Percentile Application Read Access Latencies (μs) for the Single Size Workloads.

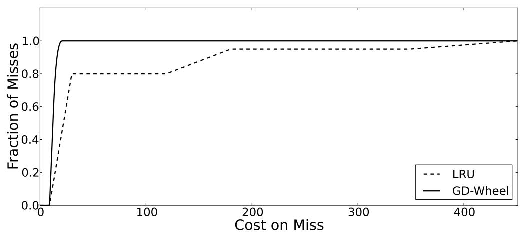


Figure 12: CDF of Recomputation Costs for the Single Size Workload 1 Baseline.

Figure 9 shows the average application read access latency for each workload, including the extra latency for recomputations. The results show that GD-Wheel greatly reduces the average application read access latency, by an average of 33% and as much as 53%. Results for workloads 2 and 3 show that GD-Wheel is beneficial under the cost variations of RUBiS and TPC-W. As would be expected, the two replacement policies have the same latency for workload 4, where all key-value pairs have the same cost. Results for workloads 6 to 9 show that changing the key-value pair sizes does not affect the performance of GD-Wheel. GD-Wheel provides similar improvements in workloads 1 and 10. This shows that changing the precision within the same cost group won't affect the performance of GD-Wheel. The variations between different clustering cost groups are more important to GD-Wheel's performance.

Reduction of Total Recomputation Cost The reason for the average read access latency improvement is that GD-Wheel greatly reduces the total recomputation cost, in other words the extra access latency on cache misses. Figure 10 shows the normalized total recomputation cost for LRU and GD-Wheel. All the numbers for LRU are set to 100 and the numbers for GD-Wheel are normalized to the total recomputation cost for LRU. The results show that GD-Wheel greatly reduces the total recomputation cost, by an average of 74% and as much as 90%. For workload 4 where all objects have the same cost, both policies have the same total recomputation cost. For all the other workloads, GD-Wheel reduces the total recomputation cost by at least 66%.

Tail Read Access Latency Tail read access latency is critical to large-scale Web services [13]. Figure 11 shows the 99th percentile application read access latency. The results show that GD-Wheel greatly reduces the 99th percentile application read access latency, by an average of 69% and as much as 85%. For workload 5 with random cost, GD-Wheel keeps the 99th percentile latency as low as 4136 μs, while LRU's 99th percentile latency is as high as 14476 μs. For all the other workloads, GD-Wheel keeps the tail latencies no larger than 1364 μs, while LRU's tail latencies have a huge variation among different workloads.

CDF of Recomputation Costs The reason for the tail read access latency improvement is that GD-Wheel avoids large recomputation costs. Figure 12 show the cumulative distribution function (CDF) of recomputation costs for workload 1. All of GD-Wheel's misses belong to the lowest cost group, while LRU have misses that fall into all three cost groups.

GET Hit Rate We recorded the GET hit rate for both LRU and GD-Wheel. Overall, the hit rates achieved by LRU and GD-Wheel differ by no more than 0.18% among all workloads. This shows that under the Zipf request distribution, GD-Wheel achieves a similar hit rate as LRU.

6.4.2 Multiple Size Workload Results

Average Read Access Latency Figure 13 shows the average application read access latency for each workload,

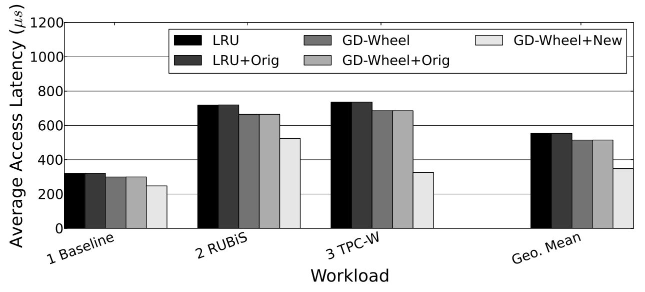


Figure 13: Average Application Read Access Latencies (μs) for the Multiple Size Workloads.

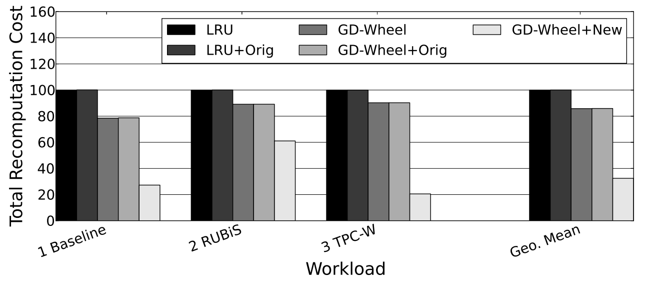


Figure 14: Normalized Total Recomputation Cost for the Multiple Size Workloads.

including the extra latency for recomputations. In all three workloads with both LRU and GD-Wheel, the original rebalancing policy didn't move any slabs, since there is no slab class with zero evictions (LRU+Orig and GD-Wheel+Orig in the figure). Nonetheless, there is still some improvement achieved by GD-Wheel alone. This is because there exists a small cost variation in each of the three cost groups (10-30, 120-180, and 350-450). However, our cost-aware rebalancing policy combined with GD-Wheel achieves a much greater improvement than using GD-Wheel alone. Compared to LRU with the original rebalancing policy, GD-Wheel with the cost-aware rebalancing policy (GD-Wheel+New in the figure) greatly reduces the average application read access latency, by an average of 37% and as much as 56%.

Reduction of Total Recomputation Cost Figure 14 shows the normalized total recomputation cost for both LRU and GD-Wheel, with the original and cost-aware rebalancing policies. All the numbers for LRU are set to 100 and the other numbers are normalized to the total recomputation cost for LRU. As explained before, the original rebalancing policy didn't migrate any slabs in all the three workloads. Thus the total recomputation cost is unchanged. GD-Wheel alone takes the cost information in each slab class into consideration and achieves reasonable reductions. In addition, GD-Wheel with the cost-aware rebalancing policy, compared to LRU, greatly reduces the total recomputation cost by an average of 68% and as much as 79%.

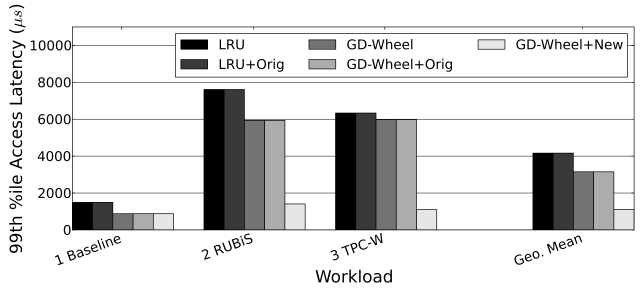


Figure 15: 99th Percentile Application Read Access Latencies (μs) for the Multiple Size Workloads.

Reduction	Avg. Read Latency	Tail Read Latency	Total Recomputation Cost
Single Avg.	33%	69%	74%
Single Max	53%	85%	90%
Multiple Avg.	37%	73%	68%
Multiple Max	56%	83%	79%

Table 4: Results Summary for Single and Multiple Size Workloads.

Tail Read Access Latency Figure 15 shows the 99th percentile application read access latency. The results show that GD-Wheel with the cost-aware rebalancing policy greatly reduces the 99th percentile application read access latency, by an average of 73% and as much as 83%. In workload 1, GD-Wheel alone achieves the same improvement on tail access latency as GD-Wheel with the cost-aware rebalancing policy. This is because most of the key-value pairs in workload 1 (80%) reside in the slab class with lowest average cost, thus the tail latency could be improved without rebalancing. In the other two workloads, the cost-aware rebalancing policy provides additional improvement on tail access latencies.

6.5 Results Summary

Table 4 summarizes the average and maximum reductions on average read latency, tail read latency, and total recomputation cost for both single and multiple size workloads. Results for single size workloads show that the GD-Wheel replacement policy could provide better performance than LRU under real-world cost variations. GD-Wheel combined with the cost-aware rebalancing policy achieves similar improvements in the multiple size workloads as GD-Wheel alone in the single size workloads. This shows that the cost-aware rebalancing policy exploits the cost information among different slab classes and provides cost-aware rebalance decisions.

7. Related Work

The GD-Wheel replacement policy presented in this paper is built upon the GreedyDual algorithm [31]. The GreedyDual algorithm has been adopted for cost-aware replacement policies in different storage and caching systems. However, to the best of our knowledge, this paper is the first to propose and evaluate a cost-aware replacement policy for memory-based key-value stores with amortized constant complexity. This section presents related work in three dimensions: cache replacement algorithms, different applications of the GreedyDual algorithm, and improvements on individual key-value storage nodes.

Cache Replacement Algorithms Cache replacement algorithms attempt to minimize various cost metrics, such as miss ratio, total cost, and average latency. Popular cache replacement algorithms [9, 17, 22, 25] aim to maximize the hit ratios. However, these replacement algorithms, unlike GreedyDual, do not consider the cost of each cached item. Although these algorithms are good choices for solving the paging problem where the cost is the same, these algorithms are not necessarily as good as GreedyDual for solving the weighted caching problem.

Applications of The GreedyDual Algorithm Cao *et al.* introduced the best known implementation of the GreedyDual algorithm. In addition, they introduced GreedyDual-Size which takes size into consideration [11]. Cao *et al.* applied GreedyDual-Size to a proxy cache and results show that GreedyDual-Size achieves improvement on hit ratio, byte hit ratio, and network costs. Pai *et al.* introduced a simple, practical strategy for locality-aware request distribution (LARD) [27]. They examined both LRU and GreedyDual-Size algorithms in the back-end's memory caches and GreedyDual-Size achieves higher throughput than LRU. PAST is a large-scale peer-to-peer persistent storage utility [28]. For the cache management, PAST's cache replacement policy is based on the GreedyDual-Size algorithm. Results show that GreedyDual-Size performs better than LRU in terms of global cache hit ratio and average number of routing hops.

The above related work applied the GreedyDual-Size algorithm to various different caching systems. In our GD-Wheel replacement policy, we didn't take the object size into consideration because Memcached separates objects of different sizes into different allocation classes and primarily performs replacement within an allocation class.

In an earlier workshop version of this paper, we presented some results for a preliminary of implementation of GD-Wheel. To the best of our knowledge, that workshop paper was the first paper to argue for the use of cost-aware replacement policies in key-value stores [18]. Compared to that workshop paper, this paper describes an implementation of GD-Wheel with smaller metadata, provides a comparison to the best-known implementation of the GreedyDual algorithm (GD-PQ), and presents a new cost-aware rebalancing policy.

Recently, Ghandeharizadeh *et al.* introduced *CAMP*, which implements an efficient approximation to the GreedyDual-Size algorithm for key-value stores [16]. CAMP stores key-value pairs in different LRU-ordered queues according to their cost-to-size ratios. To find the key-value pair with the least priority across all of the LRU queues, CAMP maintains a priority queue that tracks the priority of the key-value pair at the head of every LRU queue. To limit the number of LRU queues and the size of the priority queue, CAMP uses a rounding technique so that key-value pairs of approximately the same cost-to-size ratio are stored in the same LRU queue. In contrast to CAMP, GD-Wheel implements the exact GreedyDual algorithm. GD-Wheel uses the exact cost information provided by clients. The replacement decisions made by GD-Wheel are exactly the same as the the best-known implementation of the GreedyDual algorithm (GD-PQ).

Improving Individual Key-Value Stores FAWN-DS is a high-performance key-value storage system built on the FAWN cluster architecture [7]. The FAWN architecture—a Fast Array of Wimpy Nodes—is designed to couple low-power, efficient embedded CPUs with flash storage to provide fast, efficient, and cost-effective access to large, random-access data. FAWN-DS consists of an in-memory hash table index and an on-flash log-structured datastore. By using relatively slow CPUs with a limited memory capacity, FAWN-DS provides over an order of magnitude more queries per Joule than conventional disk-based systems.

SILT (Small Index Large Table) is a memory-efficient, high-performance flash-based key-value store that combines the features of previous work described above [19]. It requires only 0.7 bytes of DRAM per entry and retrieves key-value pairs using on average 1.01 flash reads each. SILT uses a series of three basic key-value stores, each with a different emphasis on memory-efficiency and write-friendliness, and an analytical model for tuning the system to meet different workload needs.

RAMCloud is a DRAM-based storage system that provides fast crash recovery, rather than storing replicas in DRAM [26]. RAMCloud keeps all data in DRAM all the time with full performance potential and inexpensive durability. It uses a log-structured storage in DRAM and scatters backup data across disks over the cluster.

Masstree is a persistent in-memory key-value database with particular optimizations for short and simple queries [21]. It uses a variation of B^+ trees to support range queries and applied optimizations for cache locality and optimistic concurrency control. Consistency and durability are provided by logging and checkpointing.

MemC3 is a redesign of the Memcached key-value store to achieve high concurrency and space-efficiency [14]. Its optimistic cuckoo hashing exploits CPU cache locality to minimize the number of memory fetches and overlap those fetches with different levels of parallelism. Its optimistic locking provides high-performance access to shared data structures while ensuring consistency. Its CLOCK-based eviction policy improves space efficiency and concurrency.

MICA is an in-memory key-value store that provides consistently high throughput and low latency for read/write-intensive workloads with a uniform/skewed key popularity [20]. MICA provides fast and scalable parallel data access by using data partitioning and exploiting CPU parallelism. The network stack achieves zero-copy request processing by interfacing with NICs directly. New memory allocation and indexing in MICA exploits workload properties to accelerate performance with simplified memory management.

To improve the throughput and reduce the CPU overhead of key-value stores, several works implement Memcached over RDMA on soft-iWARP [29] or Infiniband [23].

8. Conclusions

This paper has shown that there exist significant cost variations among the computation results cached in key-value stores. Consequently this paper has argued that key-value stores should provide the option for clients to include the cost information via SET requests, and take such cost variations into replacement decisions.

As a demonstration, this paper has introduced a new cost-aware replacement policy, GD-Wheel, which is an implementation of the GreedyDual algorithm with amortized constant time complexity per operation. GD-Wheel is a more efficient implementation of the GreedyDual algorithm than the previous best known implementation. It could be easily configured to support a reasonable cost range at run-time. GD-Wheel is not only for key-value stores. It could also be applied to any caching systems where cost variations exist.

This paper has described the implementation of GD-Wheel in the Memcached key-value store. In addition, this paper has also proposed a new cost-aware slab rebalancing policy for the Memcached slab allocator. In all of our experiments, GD-Wheel and the cost-aware rebalancing policy greatly reduced the total recomputation cost. As a result, our approach greatly improves the performance of web applications in terms of average and tail access latencies.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Fernando Pedone, for their comments on this work. This work was partially supported by CNS award #1018840 from the National Science Foundation.

References

[1] Memcached Internals for End Users. https://code.google.com/p/memcached/wiki/NewUserInternals.

[2] Using Redis as an LRU cache. http://redis.io/topics/lru-cache.

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

[4] TPC-W: a transactional web e-Commerce benchmark. Transation Processing Performance Council. http://www.tpc.org/tpcw/.

[5] Caching with Twemcache. https://blog.twitter.com/2012/caching-twemcache.

[6] C. Amza, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite. Specification and implementation of dynamic Web site benchmarks. In *2002 IEEE International Workshop on Workload Characterization*, WWC-5, pages 3–13, Nov 2002.

[7] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 1–14, 2009.

[8] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, 2012.

[9] L. A. Belady. A Study of Replacement Algorithms for a Virtual-storage Computer. *IBM Syst. J.*, 5(2):78–101, June 1966.

[10] S. Bouchenak, A. L. Cox, S. Dropsho, S. Mittal, and W. Zwaenepoel. Caching Dynamic Web Content: Designing and Analysing an Aspect-oriented Solution. In *Proceedings of the 7th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '06, pages 1–21, 2006.

[11] P. Cao and S. Irani. Cost-aware WWW Proxy Caching Algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, USITS '97, pages 18–18, 1997.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, 2010.

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

[14] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pages 371–384, 2013.

[15] B. Fitzpatrick. Distributed Caching with Memcached. *Linux J.*, 2004(124):5–, Aug. 2004.

[16] S. Ghandeharizadeh, S. Irani, J. Lam, and J. Yap. CAMP: A Cost Adaptive Multi-queue Eviction Policy for Key-value Stores. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 289–300, 2014.

[17] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 439–450, 1994.

[18] C. Li and A. L. Cox. GD-Wheel: A Cost-Aware Replacement Policy for Key-Value Stores. In *7th Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '13, 2013.

[19] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, 2011.

[20] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI '14, 2014.

[21] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, 2012.

[22] N. Megiddo and D. S. Modha. ARC: A Self-tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 9–9, 2003.

[23] C. Mitchell, Y. Geng, and J. Li. Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC '13, pages 103–114, 2013.

[24] 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 *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pages 385–398, 2013.

[25] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 297–306, 1993.

[26] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, 2011.

[27] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 205–216, 1998.

[28] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, a Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, SOSP '01, pages 188–201, 2001.

[29] P. Stuedi, A. Trivedi, and B. Metzler. Wimpy Nodes with 10GbE: Leveraging One-sided Operations in soft-RDMA to Boost Memcached. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC '12, pages 31–31, 2012.

[30] G. Varghese and T. Lauck. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, SOSP '87, pages 25–38, 1987.

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