RICE UNIVERSITY

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

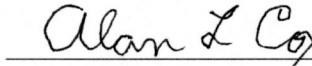
by

Conglong Li

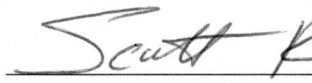
A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

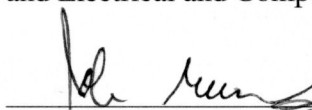
APPROVED, THESIS COMMITTEE:



Alan L. Cox, Chair
Associate Professor of Computer Science
and Electrical and Computer Engineering



Scott Rixner
Associate Professor of Computer Science
and Electrical and Computer Engineering



John Mellor-Crummey
Professor of Computer Science and
Electrical and Computer Engineering

Houston, Texas

May, 2014

ABSTRACT

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

by

Conglong Li

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, such as database queries and dynamically generated web pages. Currently, these key-value stores use either the *Least Recently Used (LRU)* replacement policy or an approximation to it for choosing a key-value pair to be evicted from the store. However, if the cost of recomputing cached values varies significantly, as in the RUBiS and TPC-W benchmarks, then neither of these replacement policies is the best choice. When deciding what key-value pair to replace, it can be advantageous to take the cost of recomputation into consideration. To that end, this thesis proposes a new cost-aware replacement policy, *GD-Wheel*, which is the first amortized constant time implementation of the *GreedyDual* replacement policy. This thesis 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 99^{th} percentile read access latency for the application.

Acknowledgments

First and foremost, I would like to greatly appreciate my advisor, Prof. Alan L. Cox, for his guidance and support on my research. When I started working with Alan as an undergraduate student, I knew nothing about research in computer science. Over these three years Alan guided me, inspired my ideas, encouraged my progress and steered my research. He motivated and helped me with his great knowledge of computer systems and architecture; he taught me skills from using version control to evaluation designs; he taught me how to write technical papers and reviewed all of my writings and presentation slides. For these, and much more, I express my utmost gratitude to him. Without Alan, I would not be where I am today.

I would also want to specially thank Prof. Scott Rixner, who has been guided me for such a long time since my undergraduate study. When I was working with Alan, Scott, and their student Myeongjae on a project about the DRAM memory system, Scott's comments during discussions always inspired my research. I would also like to thank Prof. John Mellor-Crummey for his helpful advice and valuable comments on this thesis.

I would like to thank our group members including Myeongjae Jeon, Mehul Chadha, Thomas Barr, and Brent Stephens for their feedback and advice on my research and practice talk. Also, I am grateful to my friends in Computer Science department including Yunming Zhang, Linge Dai, and Zhaolei Liu for helping me prepare a successful defense. Dr. Jan Hewitt, throughout the ENGI 600 class, helped shape this thesis. Finally, I would like to thank my parents for their unwavering support throughout the years of my study.

Contents

Illustrations

Tables

CHAPTER 1

Introduction

1.1 Introduction

Low-latency access to large scale data storage has become critical for Internet services. Considering the latency it takes for databases to execute the queries or the latency it takes for web applications to render web pages, it's necessary to cache information inside memory for faster reuse. As a distributed caching system, a memory-based key-value store allows distributed systems to make better use of memory by combining memory from different machines into a single virtual pool. To achieve fast data retrieval, memory-based key-value stores are used by many large-scale web applications. For instance, `Memcached` [1] is used at Facebook, Twitter and Zynga; and `Redis` [2] is used at GitHub, Flickr and Stack Overflow.

Key-value stores usually support two basic operations: the GET operation for retrieving the value of the given key; and the SET operation for inserting a new key-value pair. Since the operations on data structures in current key-value stores have a constant time complexity, doing a GET or SET takes constant time regardless of the number of cached items. As a result, these operations have very low latencies. Since memory-based key-value stores are used as caching systems, they have a finite storage size. When the key-value store is full, either the SET request will fail or a replacement will be made so that an old key-value

pair is evicted before inserting the new one. Currently, key-value stores use the *Least Recently Used (LRU)* replacement policy or an approximation to that policy for choosing a key-value pair to be evicted from the store. For example, Fan *et al*. recently proposed a *CLOCK*-based approximation to LRU for use in `Memcached` to increase space efficiency and concurrency [3]. These replacement policies make decisions based on the recency of access on key-value pairs.

Because of the fast response and finite size of memory-based key-value stores, they are usually used as database query caches, web page caches, or caches for any kind of computation results. For the cached key-value pairs, keys can be any arbitrary string and values are usually the results of computations, such as database queries, web pages, or API calls. By caching the results of computations in key-value stores, applications avoid recomputing the results and thereby reduce their read access latencies.

Since the computations have different purposes and even come from different sources, the results of computations cached in key-value stores have different costs. In other words, it takes different times to recompute the results of computations cached in key-value stores. In fact, real-world key-value store deployments [4, 5] and several representative web application benchmarks [6] provide evidence that significant cost variations do exist.

Cost could have different definitions determined by the client. Here we focus on the cost of recomputation times as described above. Because of the computation cost variations, it could be preferable to retain key-value pairs with higher recomputation cost. However, to take the cost into consideration for key-value stores, there are two problems. The first is that key-value stores don't have the cost information. Key-value stores themselves don't have the information since this cost can only be measured outside the cache. The client application must measure the cost and inform the key-value store. However, the current SET operation protocol does not provide an option for clients to include the cost information for

each key-value pair. We need to provide a protocol that allows clients to include the cost information.

The second problem is that neither the LRU replacement policy nor a LRU approximation takes cost into consideration. We need a replacement policy that takes cost into consideration and retains the key-value pairs with higher cost. As a possible solution, the *GreedyDual* algorithm generalizes the LRU algorithm to the weighted caching problem, which is the problem of making replacement decisions for items with non-uniform costs [7]. To solve this problem, the GreedyDual algorithm integrates recency of access and cost of cached objects when making replacement decisions.

However, there is still a problem for implementing the GreedyDual algorithm in key-value stores. Cao *et al*. introduced the best known implementation of the GreedyDual algorithm [8]. This implementation stores all objects in a priority queue. The priority of each cached object is based on both the recency of access and the cost of bringing the object into the cache. Since both inserting a new object and updating the priority of an object require logarithmic time in the number of objects, this implementation is slower than the LRU algorithm. As key-value stores currently use constant time algorithms to perform GET and SET requests, a logarithmic complexity of the replacement policy is undesirable. We need a more efficient implementation of the GreedyDual algorithm.

1.2 Thesis Statement

The performance of web applications can be significantly improved by taking recomputation cost variations into replacement considerations for memory-based key-value stores.

1.3 Contributions

This thesis argues that a key-value store's replacement policy should take recomputation cost variations into consideration. This thesis also argues that key-value stores should let client applications specify a cost for each key-value pair via the SET operation. As a demonstration, this thesis presents a new cost-aware replacement policy, *GD-Wheel*, which takes the recomputation cost into account when making replacement decisions. GD-Wheel is an efficient implementation of the GreedyDual algorithm for a limited range of costs. The GD-Wheel replacement policy uses a data structure called *Hierarchical Cost Wheels*. Inspired by Varghese and Lauck's *Hierarchical Timing Wheels* [9], the Hierarchical Cost Wheels is an efficient priority queue for a limited priority range. Using the Hierarchical Cost Wheels data structure, the GD-Wheel replacement policy yields an implementation of the GreedyDual algorithm with amortized constant time complexity per operation.

This thesis also describes the implementation of GD-Wheel in `Memcached`. Changes to the SET request protocol enable clients to optionally provide cost information with each key-value pair. Based on the slab memory allocation inside `Memcached`, key-value pairs with different sizes are stored separately in different *slab classes* with separate replacement structures. To balance the memory allocated for each slab class, `Memcached` has a re-balancing policy that periodically moves slabs between slab classes based on the eviction rates among different slab classes. In fact, this rebalancing policy is very conservative and ineffective. We replaced each slab class's LRU replacement policy with GD-Wheel. As an alternative to the original rebalancing policy, we have implemented a new cost-aware rebalancing policy for `Memcached` that rebalances the memory based on the cost information among different slab classes.

We evaluated the performance of `Memcached` with GD-Wheel and the cost-aware

rebalancing policy using the *Yahoo! Cloud Serving Benchmark (YCSB)* [10]. The whole evaluation consists of two parts. The first part is a direct comparison between GD-Wheel and LRU with our single size workloads that use a single size for all key-value pairs. By doing so, we avoid the side effect of the rebalancing policy. 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. Over all of the workloads, our approach reduces the total recomputation cost by an average of 73% and as much as 90%; our approach reduces the average read access latencies by an average of 33% and as much as 53%; our approach reduces the 99th percentile read access latencies by an average of 70% and as much as 85%.

The second part is an evaluation of the rebalancing policy in `Memcached` with our multiple size workloads that use different sizes for key-value pairs with different costs. This evaluation shows that GD-Wheel combined with our cost-aware rebalancing policy, when compared to LRU with the original rebalancing policy, greatly reduces the total recomputation cost as well as the average and 99th percentile read access latency for the data. Over all of the workloads, our approach reduces the total recomputation cost by an average of 68% and as much as 79%; our approach reduces the average read access latencies by an average of 37% and as much as 56%; our approach reduces the 99th percentile read access latencies by an average of 73% and as much as 83%.

1.4 Organization

This thesis is organized as follows. Chapter 2 provides deeper motivation for our work. Chapter 3 presents the GD-Wheel replacement policy. Chapter 4 describes the implementation of GD-Wheel in the `Memcached` key-value store. Chapter 5 describes the original rebalancing policy in `Memcached` and presents our alternative cost-aware rebalancing

policy. Chapter 6 describes our experimental methodology, and presents our experimental results. Chapter 7 discusses related work. Finally, we summarize this thesis and present future work in Chapter 8.

CHAPTER 2

Motivation

This chapter provides deeper motivation for our work. First, we explain how memory-based key-value stores are used by web applications. Then we discuss different sources of cost variations in recomputation of key-value pairs. Finally we summarize the motivation.

2.1 How Key-Value Stores are Used

For a web application such as an online bookstore, it may receive HTTP requests asking for book detail, new books, and best sellers, etc. Then, the application generates necessary database queries and request the database to execute the queries. After receiving the query results, the application generates the HTTP web page and send the response. As described previously, there is a spectrum of what could be cached in key-value stores: it could be as low level as database query results; it could also be as high level as HTTP web pages; or it could also be both low level and high level computations cached in the same store.

Figure 2.1 illustrates how a key-value store is used to 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 first 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 hash table 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

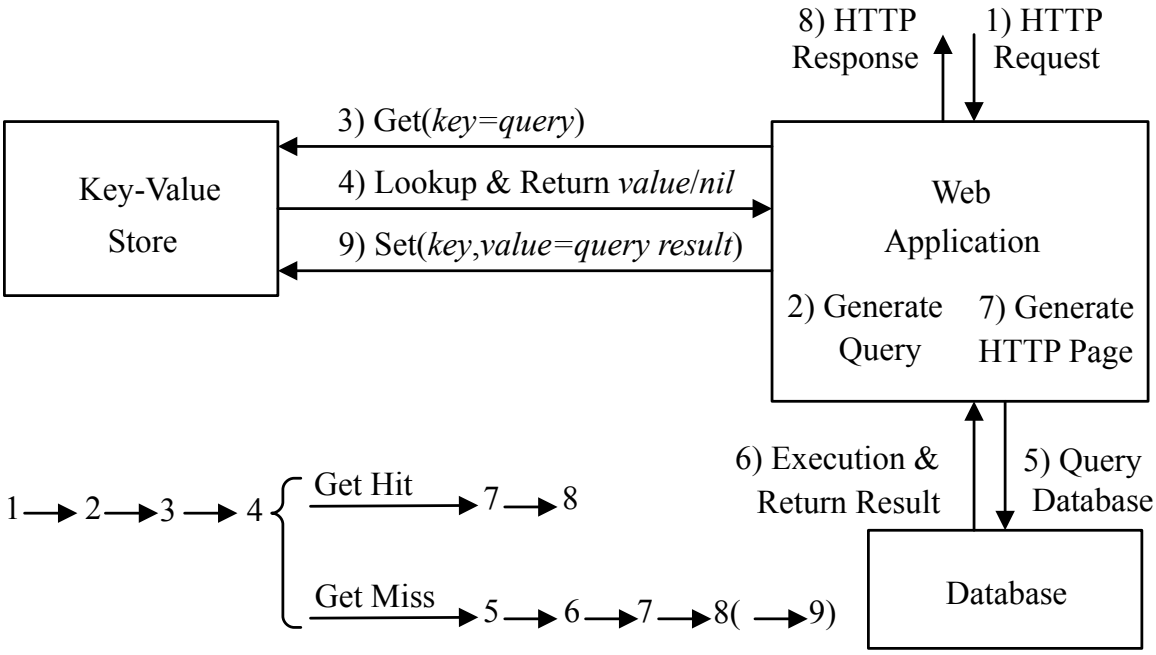


Figure 2.1 : Using a Key-Value Store as a Database Query Cache.

step 7 to generate the HTTP web page and sends the response (Step 8). We refer to this 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 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 example, Figure 2.2 illustrates how a key-value store is used to as a web page cache. After receiving a HTTP request (Step 1), the application asks the key-value store first for the HTTP response page by sending a GET request with the HTTP request as the key (Step 2). After receiving the GET request, the key-value store performs a hash table lookup for the requested key and returns either the cached web page or a *not found* error (Step 3). If the query result is returned, the application skips to step 8 and directly return the HTTP response. We refer to this as a GET hit. If, however, a *not found* error is returned, then the application needs to generate the necessary database query (Step 4),

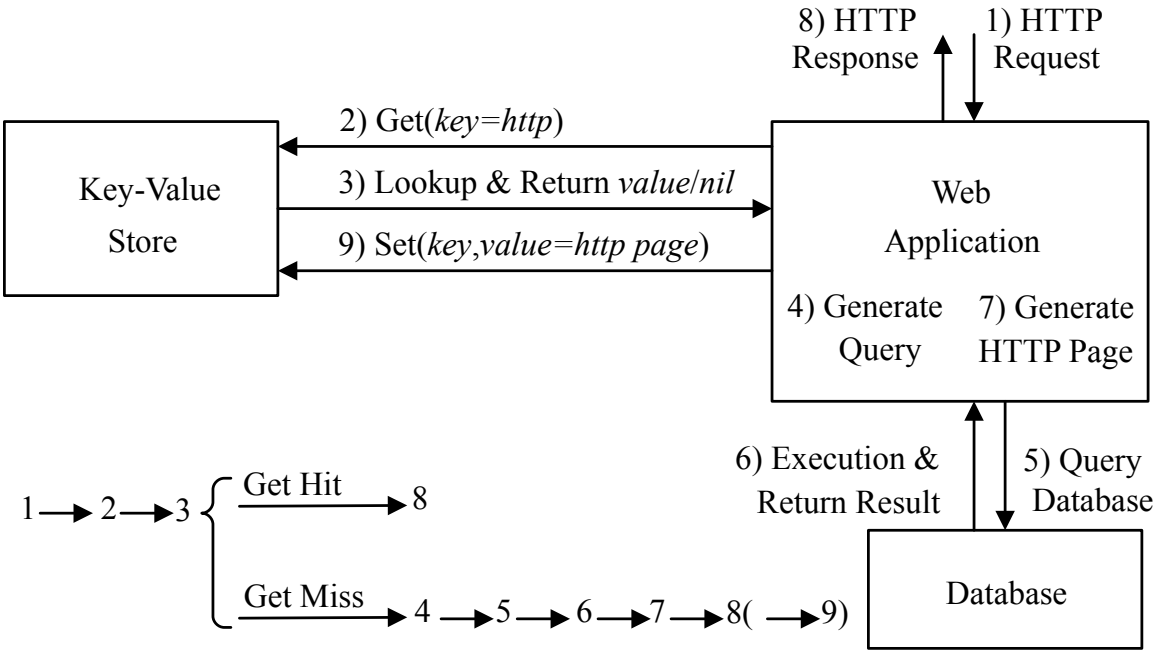


Figure 2.2 : Using a Key-Value Store as a Web Page Cache.

request the database to execute the query (Steps 5&6), and then generate the HTTP web page (Step 7). We refer to this as a GET miss. After the execution in the database, the application usually sends a SET request to retain the HTTP web page for later use (Step 9).

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 values. For the query results, the recomputation cost is the sum of steps 5 and 6 in Figure 2.1. For the web pages, the recomputation cost is the sum of steps 4 to 7 in Figure 2.2. However, current key-value stores don't provide the option for clients to include such recomputation cost information in SET requests.

There are several ways to invalidate any stale data inside key-value stores. The first way is including an optional expiration time parameter on SET requests so that key-value pairs will expire and be deleted after the specified time has elapsed. Alternatively, the web application could send a request to delete or update certain stale data in key-value store,

since the application is aware of all changes to the data in database.

2.2 Cost Variations of Recomputations

There exist two common sources of cost variations for recomputations: cost variations from heterogeneous cache usage and cost variations among objects of the same type.

2.2.1 Cost Variations among Heterogeneous Cache Usage

The first source of cost variations is different kinds of simultaneous cache usage, in other words, 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: as an optimization for disk where cache is used as the in-memory serving layer to shed load from databases; and as an optimization for CPU where cache is used as a buffer to store items that are expensive to recompute [4]. Twemcache stores recently and frequently accessed Tweets to reduce the frequency of disk accesses. As an optimization for CPU, Twemcache also stores recently rendered Tweets which includes computed metadata such as the number of retweets, favorites, etc.

As another example of heterogeneous cache usage, Facebook uses `Memcached` for various purposes [5]. Facebook uses `Memcached` as a query cache to lighten the read load on databases. In particular, Facebook use `Memcached` as a demand-filled look-aside query cache. Facebook also leverages `Memcached` as a more general key-value store for caching computation results such as sophisticated machine learning algorithm results which could be used by a variety of applications.

To accommodate these differences on cache miss cost, 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 separating key-value pairs with different recomputation costs would work, it requires prior analysis to determine the exact size of each pool. If the workload characteristics change over time, such a static partitioning of `Memcached` servers may result in inefficient usage of memory. It could be be more efficient to maintain a single pool and make replacement decisions based on the recomputation cost variations.

2.2.2 Cost Variations among Objects at the Same Level

The second source of cost variations in recomputations among objects is 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*. implemented a web page cache above two web application benchmarks: *RUBiS* and *TPC-W* [6].

RUBiS implements the core functionality of an auction site modelled after eBay [11]. It defines interactions including registering new users, browsing items by category or region, bidding, buying or selling items, and leaving comments. Table 2.1 summarizes the extra response time that Bouchenak *et al*. measured in the case of a cache miss for different interactions. For some simple interactions such as browsing items (Browse Cat and Browse Rgn), the extra response time on cache misses is as low as 10*ms*. On the other hand, the extra response time on cache misses is as high as 240*ms* for complicated interactions such as showing user information including buying and selling history (About Me).

TPC-W is a web server and database performance benchmark that simulates an online bookstore [12]. It defines interactions including accessing a user home page, listing new

Request	Percent of Requests	Extra Response Time on Cache Misses
Browse Cat	14%	10 *ms*
Browse Rgn	3%	10 *ms*
Search Cat	25%	60 *ms*
View Item	22%	65 *ms*
Put Cmt	2%	70 *ms*
View User	6%	70 *ms*
Buy Now	2%	75 *ms*
View Bids	3%	75 *ms*
Put Bid	10%	85 *ms*
Search Rgn	9%	95 *ms*
About Me	4%	240 *ms*

Table 2.1 : RUBiS: Extra Response Time on Cache Misses for Different Interactions.

Request	Percent of Requests	Extra Response Time on Cache Misses
admin request	2%	10 *ms*
order display	1%	10 *ms*
order inquiry	1%	10 *ms*
product detail	20%	25 *ms*
search request	24%	25 *ms*
home interaction	19%	45 *ms*
new products	6%	150 *ms*
best sellers	7%	210 *ms*
execute search	20%	300 *ms*

Table 2.2 : TPC-W: Extra Response Time on Cache Misses for Different Interactions.

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

Table 2.3 : Summary of Extra Response Time on Cache Misses for Different Interactions.

products and best sellers, registering a new user, updating the shopping cart, ordering, etc. Table 2.2 summarizes the extra response time that Bouchenak *et al*. measured in the case of a cache miss for different interactions. For some simple interactions such as displaying orders and showing product detail, the extra response time on cache misses is as low as 10*ms* to 25*ms*. On the other hand, the extra response time on cache misses is as high as 210*ms* to 300*ms* for complicated interactions such as showing best sellers and executing searches.

To summarize, Table 2.3 categorizes the extra response time measured in the case of a cache miss for RUBiS and TPC-W. We categorize the extra response time in three groups, where the ratio of response times between different groups is roughly 1:7.5:20. These numbers demonstrate that there exist large variations in the execution times for different interactions.

2.3 Summary

Based on the motivation, we show that there exist significant cost variations for results of computations. For example, the recomputation times for RUBiS and TPC-W range from 10 *ms* to 300 *ms*, for a ratio of 1:30. As a cache for web applications, it could be beneficial for key-value stores to provide the option for clients to include such cost information,

and it could be beneficial to take such cost variations into eviction decisions. In the later evaluation of the GD-Wheel cost-aware replacement policy, our workloads follow the cost variations adopted from RUBiS and TPC-W. Results show that GD-Wheel could provide improvement on average and tail read access latency under such cost variations.

CHAPTER 3

GD-Wheel Replacement Policy

This chapter presents GD-Wheel, a new cost-aware replacement policy. First we introduce the GreedyDual Algorithm and its best known implementation using a single priority queue. Then we present our efficient implementation of the GreedyDual algorithm inside GD-Wheel: we first present our Hierarchical Cost Wheels data structure, which is an efficient priority queue for a limited cost range; then we show that using the Hierarchical Cost Wheels, GD-Wheel yields an efficient implementation of the GreedyDual algorithm; finally we provide a time complexity analysis to show that GD-Wheel has amortized constant time complexity per operation.

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 [7]. The GreedyDual algorithm is of practical interest because it generalizes the LRU algorithm and integrates recency of access and cost of cached objects when making replacement decisions.

Described in Algorithm 1, 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

Algorithm 1 The GreedyDual Algorithm

```
Initialize Lowest ← ∅
For each requested object p
if p is already in memory
    H(p) ← 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
        Lowest ← {a | a ∈ memory and H(a) = min{H(b) | b ∈ memory}}
        Evict the least recently used object q in Lowest
        Reduce H value of all cached object by H(q)
    H(p) ← c(p)
    Insert p into priority queue
end
```

the lowest H value $H(q)$ is evicted. If multiple objects have the lowest H value, the least recently used one will be evicted. Then all cached objects reduce their H values by $H(q)$. Since objects with higher cost and recently inserted or reused objects will have higher H values, the GreedyDual algorithm seamlessly integrates recency of access and cost of cached objects in making eviction decisions. Since an eviction requires subtractions on all cached objects, this implementation requires $O(n)$ time complexity for each eviction, where n is the total number of cached objects.

To reduce the time complexity, Cao *et al*. introduced the best known implementation of the GreedyDual algorithm [8]. Described in Algorithm 2, 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. This inflation value is used as follows. 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

Algorithm 2 The GreedyDual Algorithm under Cao *et al.*'s implementation

```
Initialize Lowest ← ∅
Initialize L ← 0
For each requested object p
if p is already in memory
    H(p) ← 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
        Lowest ← {a | a ∈ memory and H(a) = min{H(b) | b ∈ memory}}
        Evict the least recently used object q in Lowest
        Let L ← H(q)
    H(p) ← L + c(p)
    Insert p into priority queue
end
```

recently used one will be evicted. Then global inflation value L is updated to the evicted lowest H. As the priority value H combines the global inflation value and the cost of cached objects, objects with higher cost and recently inserted or reused objects will have higher priority values.

Cao *et al*. implemented the GreedyDual algorithm by maintaining a priority queue based on the H values. By introducing the global inflation value, this implementation avoids subtractions on all cached objects on every eviction. Consequently, handling an insertion, a reuse, or an eviction requires $O(\log n)$ time. However, there is a risk of overflow for the global inflation value L. To avoid the overflow when the inflation value is too large, a scan of the priority queue is required to reduce the inflation value. Although each scan takes $O(n)$ time, it happens rarely only when the inflation value reaches the size limitation.

Although Cao *et al.*'s implementation reduces the complexity to logarithmic time per operation by introducing the inflation value, it is still undesirable for key-value stores. As key-value stores require constant operation time on GET and SET requests, we need a more

efficient implementation of the GreedyDual algorithm with constant time complexity per operation.

3.2 GD-Wheel Replacement Policy

The GD-Wheel replacement policy is an efficient implementation of the GreedyDual algorithm for a limited range of cost. GD-Wheel replacement policy uses a new data structure, the Hierarchical Cost Wheels, which is an efficient priority queue for a limited priority range. Using the Hierarchical Cost Wheels data structure, GD-Wheel replacement policy yields an implementation of the GreedyDual algorithm with amortized constant time complexity per operation.

3.2.1 Hierarchical Cost Wheels

Inspired by Varghese and Lauck's *Hierarchical Timing Wheels* [9], the Hierarchical Cost Wheels data structure provides an efficient priority queue for a limited cost range. As shown in Figure 3.1, it is made up of a series of *Cost Wheels*. A Cost Wheel is basically an array of queues. Each Cost Wheel has a clock hand pointing to one of its queues. We will first show how a single Cost Wheel handles operations including insertions, reuses, and evictions. Then we show that by using multiple Cost Wheels in a hierarchy, the Hierarchical Cost Wheels data structure provides a large enough cost range for any reasonable cost variation.

A Single Cost Wheel A single Cost Wheel (Level 1 Cost Wheel in Figure 3.1) has completely the same functionality as a priority queue. The only difference is that the Cost Wheel only supports r different priorities, where r equals the number of queues in the Cost Wheel. Acting as the global inflation value in Cao *et al.*'s implementation, the clock hand

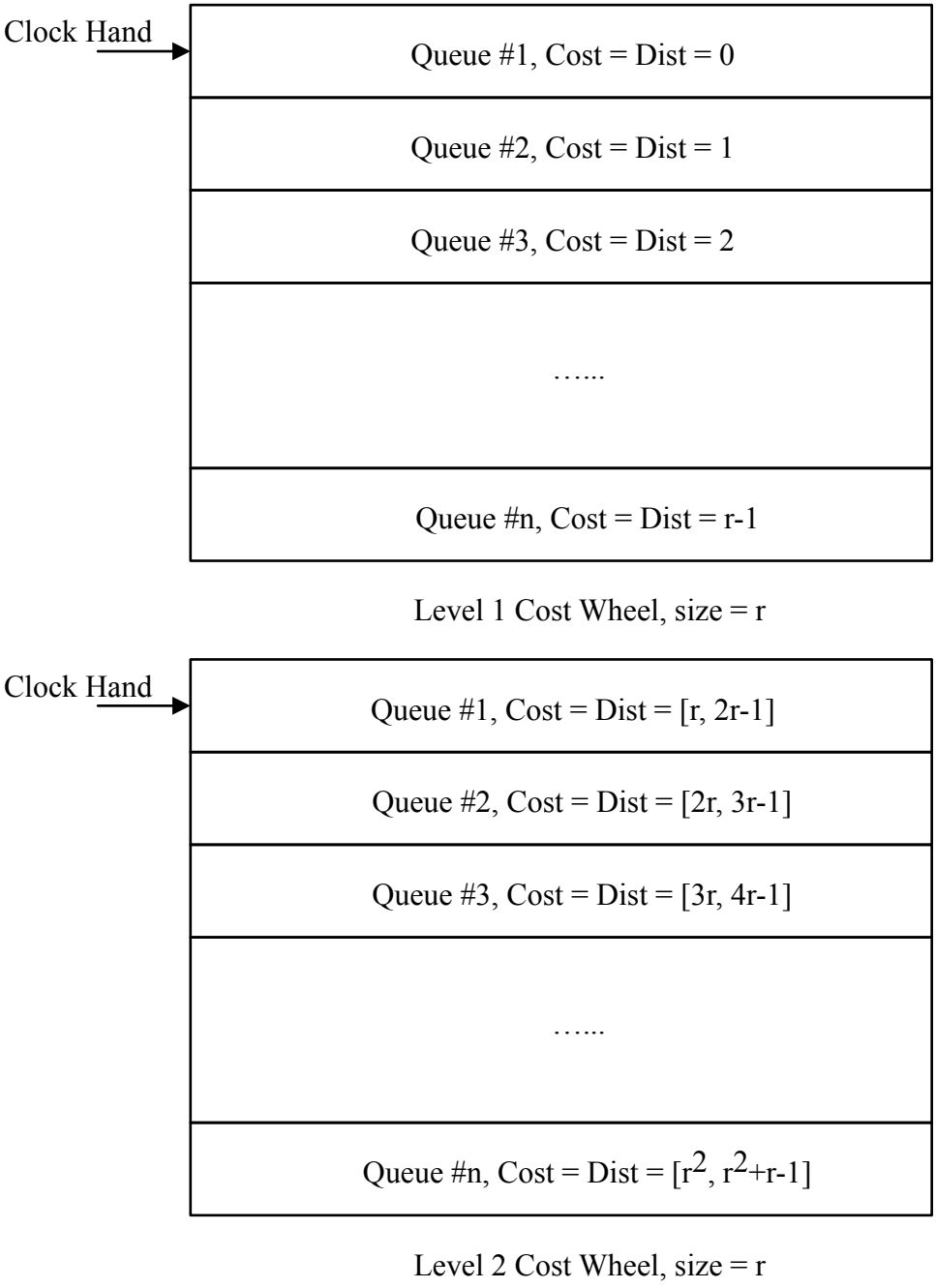


Figure 3.1 : An Example of Two-Level Hierarchical Cost Wheels.

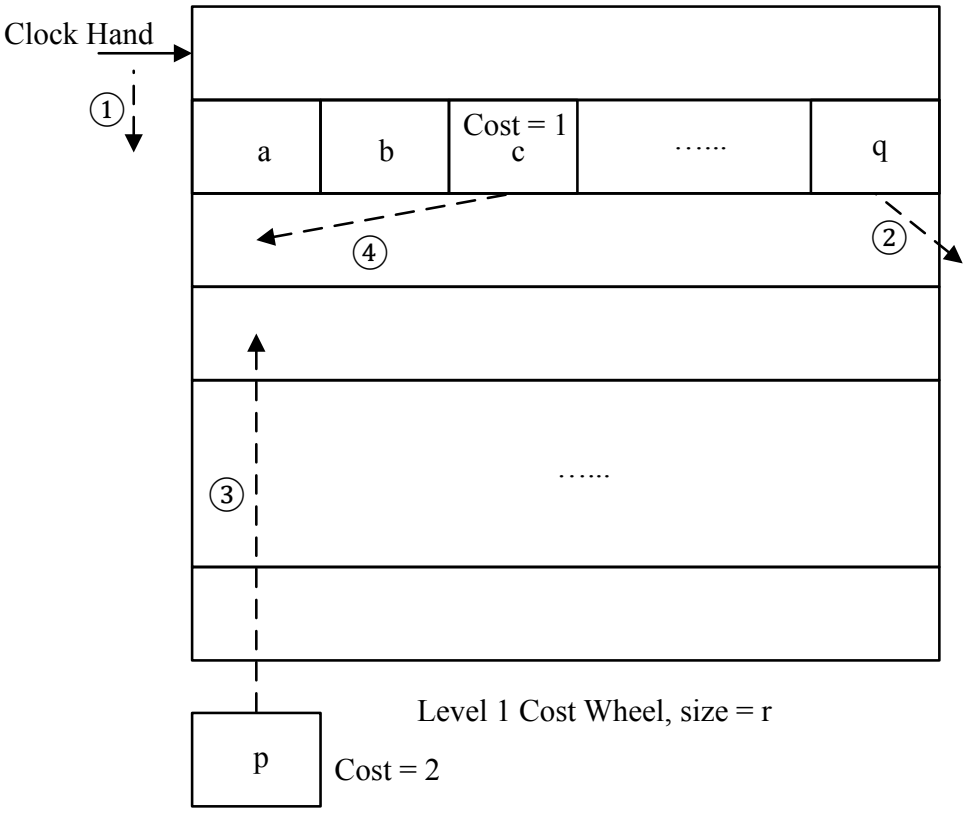


Figure 3.2 : An Example of Handling an Eviction, an Insertion, and a Reuse in a Single Cost Wheel.

keeps advancing 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 $((c+x) \bmod r)^{th}$ queue. As a result, objects with different priorities are stored in different queues.

Figure 3.2 illustrates how to handle evictions, insertions, and reuses in a single Cost Wheel. For the object p to be inserted, if there is not enough room in the memory for p, the clock hand will advance until a non-empty queue is found (Step 1). Then the object q at the tail of the queue pointed to the clock hand will be evicted (Step 2). When there is

enough room in the memory for p, p will be inserted at the head of the selected queue by adding the object's cost to the current position of the clock hand. For example, as shown in Figure 3.2, 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. For example, as shown in Figure 3.2, 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 special 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. As the clock hand advances on evictions, objects with higher cost and recently inserted or reused objects will be relatively further from the clock hand.

Hierarchical Cost Wheels Since a single Cost Wheel's size r is fixed, a single Cost Wheel can only support a limited range of costs. To extend the range of cost in an efficient manner, we used multiple Cost Wheels in a hierarchy such that each higher level Cost Wheel supports a larger range of costs. As shown in Figure 3.1, each queue in the level 1 Cost Wheel only supports a single cost, while each queue in the level 2 Cost Wheel supports r different costs. In general, each queue in a level x Cost Wheel will support r^{x-1} different costs, where r is the number of queues in each Cost Wheel.

The Hierarchical Cost Wheels act the same as a single Cost Wheel on insertions and reuses. Objects are inserted or updated to selected queues by adding the object's cost to the current position of the clock hand. Objects will be evicted from the lowest level Cost Wheel, not from the higher level 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

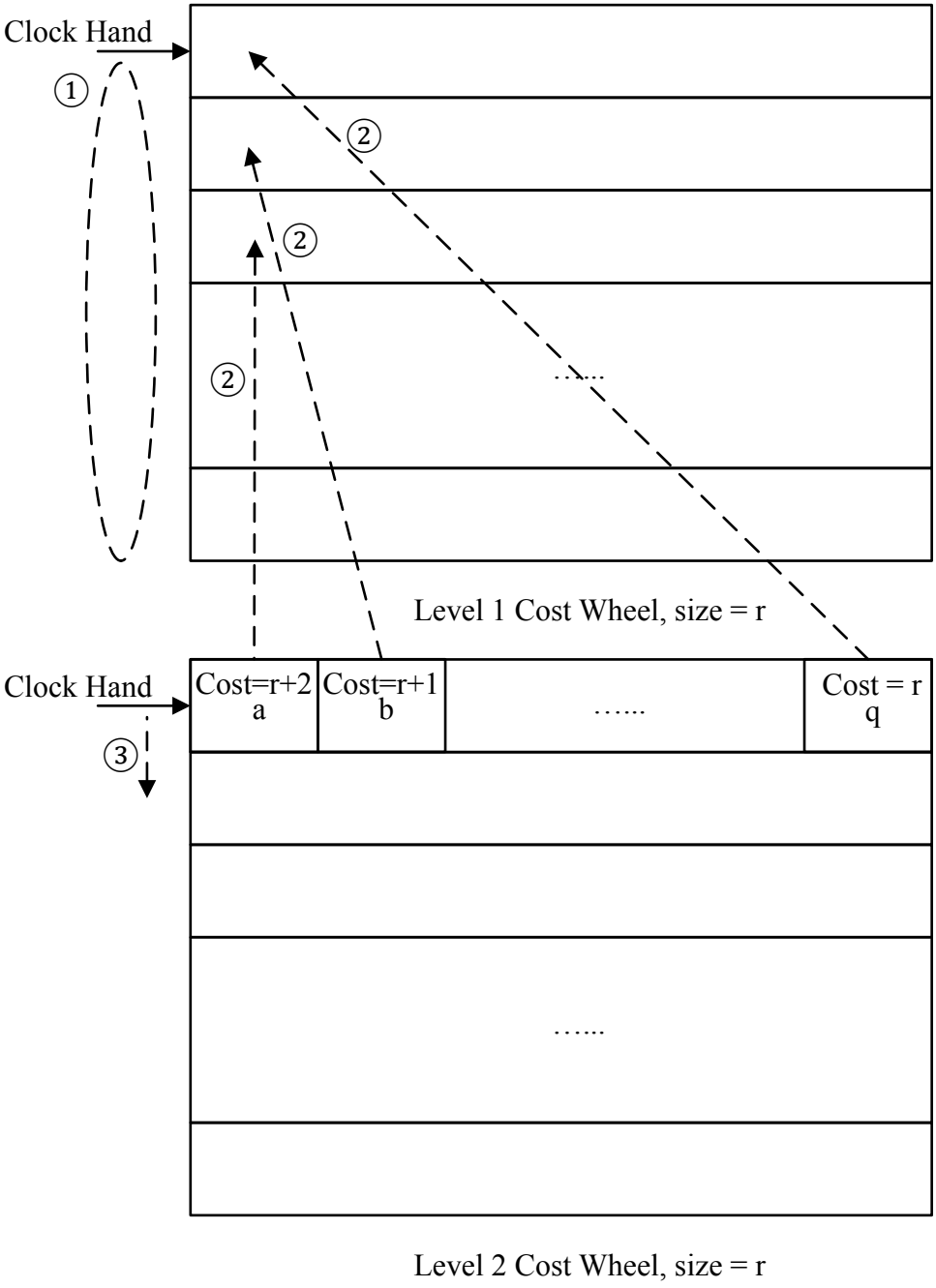


Figure 3.3 : An Example of a Migration in the Higher Level Cost Wheel.

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.

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

Figure 3.3 illustrates how migration is performed. After the clock hand in the level 1 Cost Wheel has completed a whole round (Step 1), 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 2): object a with cost r+2 is moved to the 3^{rd} queue; object b with cost r+1 is moved to the 2^{nd} queue; and object q with cost r is moved to the 1^{st} queue in the level 1 Cost Wheel. After the migration, the clock hand in the level 2 Cost Wheel advances to the next queue (Step 3). If the migrated object is reused, it will "jump" back to a higher level Cost Wheel since it has higher priority again.

3.3 GD-Wheel Time Complexity Analysis

Algorithm 3 above summarizes the GreedyDual algorithm under GD-Wheel's implementation as described in the previous section. It's modified from the original algorithm because of the different data structure. In this section, we will argue that this implementation of the GreedyDual algorithm achieves amortized constant complexity per operation.

Algorithm 3 The GreedyDual Algorithm under GD-Wheel's implementation

Let $NWHEELS \leftarrow$ number of Cost Wheels in the data structure
Let $WSIZE \leftarrow$ number of queues in each Cost Wheel
Let *Clock_Hand[NWHEELS]* $\leftarrow$ array of ones //Each clock hand starts from the first queue
For each requested object p
if p is already in memory
    Remove p
    $Wheel\_Idx \leftarrow max\{i \mid 0 < i \leq NWHEELS \text{ and } round(c(p)/WSIZE^{(i-1)}) > 0\}$
    $Queue\_Idx \leftarrow (round(c(p)/WSIZE^{(Wheel\_Idx-1)}) + 1 + Clock\_Hand[Wheel\_Idx]) \bmod WSIZE$
    Insert p to the head of $Queue\_Idx^{\text{th}}$ queue in the $Wheel\_Idx^{\text{th}}$ Cost Wheel
if p is not memory
    while there is not enough room in memory for p
        Clock_Hand[1] $\leftarrow$ the index of next non-empty queue in level 1 Cost Wheel
        Evict q at the tail of the pointed queue
        If *Clock_Hand[1]* has advanced for a whole round back to 1, call migration(2)
    $Wheel\_Idx \leftarrow max\{i \mid 0 < i \leq NWHEELS \text{ and } round(c(p)/WSIZE^{(i-1)}) > 0\}$
    $Queue\_Idx \leftarrow (round(c(p)/WSIZE^{(Wheel\_Idx-1)}) + 1 + Clock\_Hand[Wheel\_Idx]) \bmod WSIZE$
    Insert p to the head of $Queue\_Idx^{\text{th}}$ queue in the $Wheel\_Idx^{\text{th}}$ Cost Wheel
end
Function migration(idx)
    For each object p in the $Clock\_Hand[idx]^{\text{th}}$ queue in the idx^{th} Cost Wheel
        Remove p
        $Cost\_Remainder \leftarrow c(p) \bmod WSIZE^{(idx-1)}$
        $Queue\_Idx \leftarrow (round(Cost\_Remainder/WSIZE^{(idx-2)}) + 1 + Clock\_Hand[idx-1]) \bmod WSIZE$
        Insert p to the head of $Queue\_Idx^{\text{th}}$ queue in the $(idx-1)^{\text{th}}$ Cost Wheel
    $Clock\_Hand[idx] \leftarrow (Clock\_Hand[idx] + 1) \bmod WSIZE$
    If *Clock_Hand[idx]* has advanced for a whole round back to 1, call migration(idx+1)
end

In general, an object is first inserted into the cache, then reused for 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. A reuse of a given object requires one list removal and one list insertion. An insertion of a given object requires one list insertion.

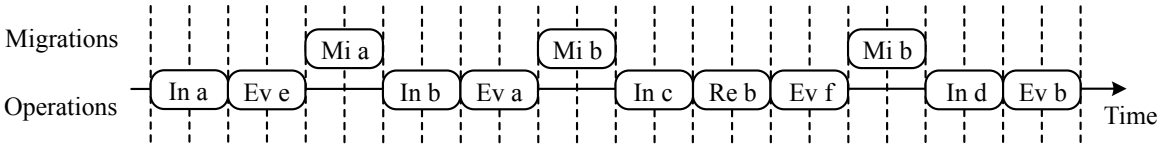


Figure 3.4 : A Sequence of Operations with Migrations. In x = Insert x. Ev x = Evict x. Re x = Reuse x. Mi x = Migrate x.

Thus handling either an object reuse or an object insertion just takes only $O(1)$ time in the worst case.

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 take 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 in 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 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 a single migration of one object takes constant time. Migrations happen when the clock hand in a lower level Cost Wheel completes a whole round. Between any two reuse, insertion or eviction operations on an object, there could only be a constant number of migrations of that object, and therefore constant cost. Considering a sequence of operations for an arbitrary object, the cost per operation, including the cost of migrations between operations, is amortized constant. In the following discussion, we will prove that there could only be a constant number of migrations of an arbitrary object between any two operations on the object.

Figure 3.4 illustrates a timeline example for a sequence of operations with migrations. An object could be reused zero or more times before eviction. For example, object a in Figure 3.4 is inserted and then evicted before any reuse, while object b is inserted, reused, and then evicted. Between two operations on an object, there could be zero or more migrations of the affected object. In Figure 3.4, object a is migrated between an insertion and an eviction of the object a. On the other hand, object b is first migrated between an insertion and a reuse, and then migrated between a reuse and an eviction of the object b. 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 either an insertion or reuse and a reuse; or between either an insertion or reuse and an eviction. In the following discussion of these two cases, we show that there could only be a constant number of migrations of the affected object.

An insertion/reuse followed by a reuse 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, 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 of the object, there could only be constant number of migrations. Thus the migrations between an insertion/reuse and a reuse have constant cost. Any further migrations after the second reuse will be charged to that reuse operation.

An insertion/reuse followed by an eviction As described in the first case, there will be a constant number of migrations of an object until an operation removes the object from the lower level Cost Wheel. Before the eviction of the object, there could only be a constant number of migrations. Thus the migrations between an insertion/reuse and an eviction have

constant cost.

3.4 Summary

To summarize, GD-Wheel provides an efficient implementation of the GreedyDual algorithm with amortized constant time complexity per operation. Across a sequence of operations, handling an insertion, a reuse, or an eviction only takes $O(1)$ time on average. This constant time complexity enables the implementation of the GreedyDual algorithm inside key-value stores without increasing the latencies of GET and SET operations. In the next chapter, we will describe the implementation of GD-Wheel in the `Memcached` key-value store.

CHAPTER 4

Implementation

This chapter describes the implementation of GD-Wheel in the `Memcached` key-value store. First we will introduce the basic components of memory-based key-value stores: an index of the key-value pairs, the memory allocator, and the replacement data structure. This is followed by a more detailed introduction to the `Memcached` key-value store including its slab allocator and LRU replacement policy. Finally we describe the implementation of GD-Wheel in `Memcached`.

4.1 Basic Components of Memory-Based Key-Value Stores

Figure 4.1 illustrates the basic components of memory-based key-value stores: an index of the key-value pairs, the memory allocator, and the replacement data structure. The index consists of a hash table which maps the hash value of a requested key to the location of the key-value pair stored in the memory. The memory allocator processes every allocation and reclaim of each key-value pair. The replacement data structure makes eviction decisions when there is not enough room for the new inserted key-value pair.

Each cached key-value pair has metadata which include 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. Thus the actual memory allocated for each key-value pair is based on the total size of the key, value, and metadata. The hash

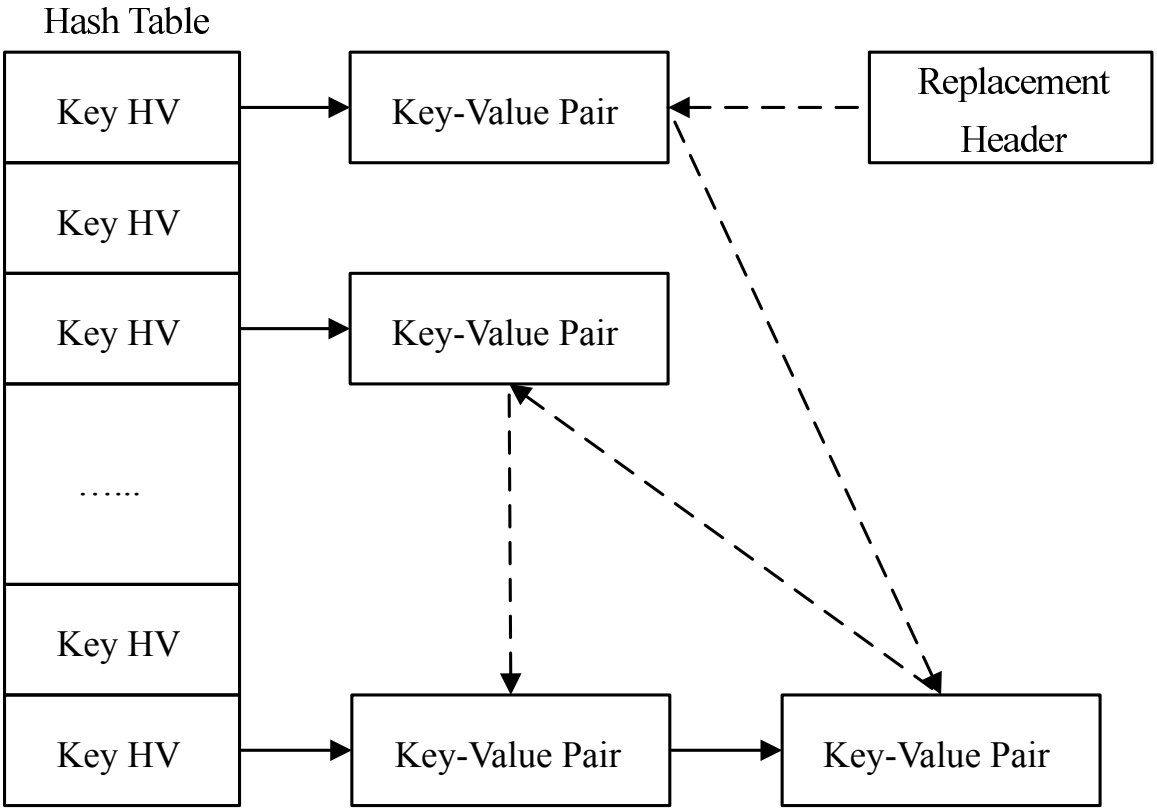


Figure 4.1 : Basic Components of Memory-Based Key-Value Stores.

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 indicate the priority of eviction 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 maps the hash value to the key-value pair in the chain that matches the requested key. After locating the requested key-value pair, key-value store will 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

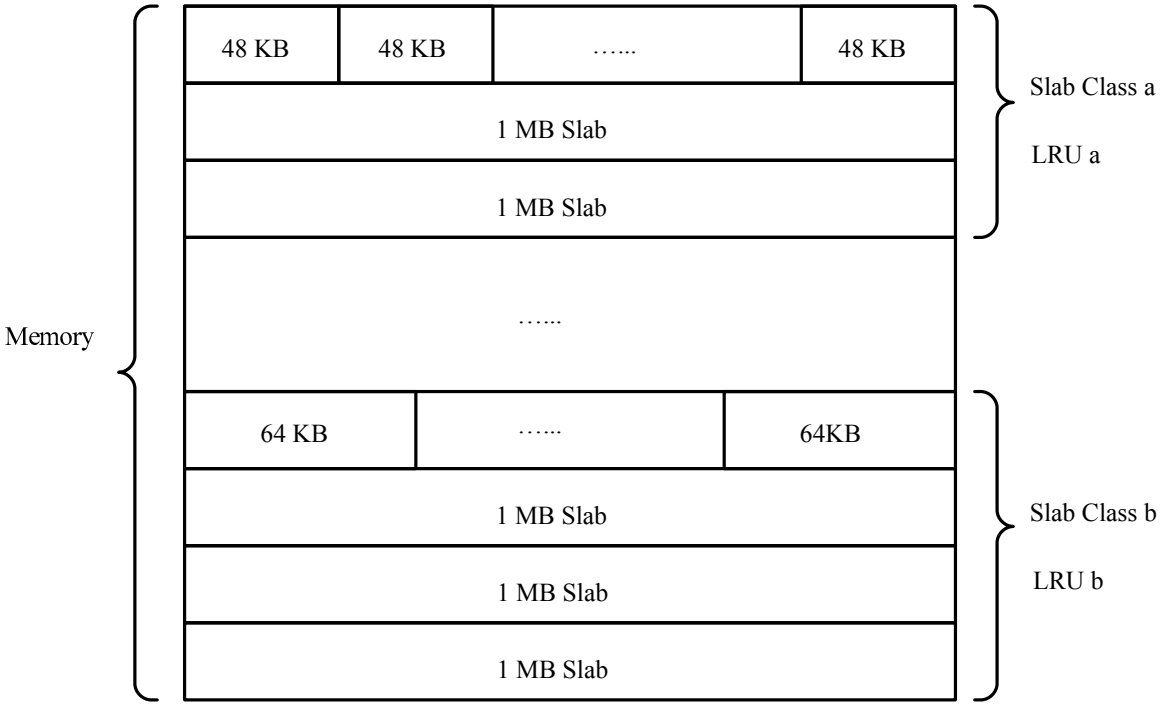


Figure 4.2 : Memcached's Memory Allocation and Replacement

will decide which key-value pair to be evicted and inform the memory allocator to reclaim the memory. After the eviction, the new key-value pair is allocated in the memory and inserted in 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 deployment and ease of development. `Memcached` supports basic operations GET and SET, together with other operations such as DELETE, REPLACE, APPEND, ADD, etc. This section provides a detailed introduction to the `Memcached` key-value store including its slab allocator and LRU replacement policy.

Figure 4.2 illustrates `Memcached`'s memory allocation and replacement designs. `Memcached` uses a slab allocator for memory allocation. This slab allocator uses dif-

ferent slab classes to store key-value pairs within different size ranges. The whole memory is broken up into 1 MB slabs. Slabs are distributed to different slab classes based on the number of key-value pairs stored in each class. Each slab class stores key-value pairs that fall within a certain size range. 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 will be considered for replacement of an inserted object. Key-value pairs are allocated into different slab classes based on their sizes. If there are no free chunks, and no free slabs for that slab class, `Memcached` will look at the end of the slab class's LRU queue for an object to reclaim. It will first search the last few objects for one which has already expired and thus is free for reuse. If `Memcached` cannot find an expired object, it will then evict the least recently used object.

4.3 Implementation in Memcached

Our implementation replaces each slab class's LRU replacement policy with GD-Wheel. We implemented a two level Hierarchical Cost Wheels where each Cost Wheel has 256 queues. 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 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 send the additional cost information with each key-value pair to the `Memcached` server.

CHAPTER 5

Rebalancing Policy in Memcached

This chapter describes the original rebalancing policy in `Memcached` and presents our alternative cost-aware rebalancing policy. We first show how the original rebalancing policy in `Memcached` works. Then we discuss the deficiency of the original rebalancing policy and present our cost-aware rebalancing policy.

5.1 The Original Rebalancing Policy in Memcached

Since the whole memory is distributed to different slab classes and each slab class has its own replacement structure, the eviction rates between different slab classes may vary significantly. To balance the memory allocated for each slab class, `Memcached` has a rebalancing policy that periodically moves slabs between slab classes based on the eviction rates among different slab classes. The rebalancing policy checks the eviction rate of each slab class 3 times per 30 seconds. If a slab class has the highest eviction count 3 times 10 seconds apart, it will take one slab from a slab class which has had zero evictions in the 30 seconds. The policy will select the least recently used slab in the class without evictions to the class with most evictions.

The original rebalancing policy in `Memcached` aims to balance the memory allocated for each slab class based on the eviction rates between slab classes. However, this policy is very conservative and ineffective. First, the policy only takes a slab from a slab class which

has had zero evictions in the last 30 seconds. If there is a slab class with larger chunks and a lower eviction rate, and a slab class with smaller 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.

Based on the recomputation cost variations, a rebalancing policy should also take the cost information into rebalance 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 the average costs in other slab classes, it could be preferable to avoid evictions in the slab class with higher average cost.

5.2 The Cost-Aware Rebalancing Policy

As an alternative of the original rebalancing policy, we implemented a new cost-aware rebalancing policy in `Memcached`. Each slab class will maintain an average cost per byte information. Our rebalancing policy will move slabs from the class with lowest average cost to the classes with higher average cost. We kept a special variable in `Memcached` which remembers the slab class id that has lowest average cost. This variable is updated when the average cost information is changed in any slab class. Such update will take 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 lowest cost class to the higher cost class. The number of slabs to be 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 avoiding potential evictions in high-cost slab classes, our new rebalancing policy completes the cost-awareness inside `Memcached`.

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 of each key-value pair, it cannot collaborate with the LRU replacement policy. In the following evaluation chapter, we will compare the GD-Wheel combined with cost-aware rebalancing policy with the LRU combined with original rebalancing policy.

CHAPTER 6

Evaluation

This chapter describes our experimental methodology and presents our results. First we show the real-world key-value store workload characteristics, which are used to model our workloads. Then we describe the evaluation environment we used and the YCSB Benchmark. This is followed by the evaluation workload characteristics which are modeled based on the real-world workloads. Finally we present our evaluation results.

There are two goals to our evaluation. First, we present a direct comparison of the LRU and GD-Wheel replacement policy that shows how GD-Wheel makes a difference for key-value store. We fulfilled this goal with our study of the single size 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 a single slab class. This avoids side effects of rebalancing policy and provides a direct comparison between LRU and GD-Wheel.

The second goal is to evaluate our cost-aware rebalancing policy in `Memcached`. We fulfilled this goal with our study of the multiple size workloads that use different sizes and therefore different slab classes for key-value pairs with different costs. We show that GD-Wheel together with the cost-aware rebalancing could greatly improve the performance of the application.

6.1 Real-World Workloads of Key-Value Stores

Recently, Atikoglu *et al.* [13] and Nishtala *et al.* [5] 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. In the later evaluation of the GD-Wheel replacement policy and the cost-aware rebalancing policy, we model our workloads based on the characteristics of these real-world workloads.

6.1.1 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 returned values to `Memcached` GET requests have a median size of 135 bytes and a mean size of 954 bytes.

6.1.2 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 `Memcached` 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 dependent on the GET miss rate.

6.1.3 Miss Rate

The miss rate is not negligible. 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. This shows that there still exist a great number of GET misses for the `Memcached` at Facebook.

6.1.4 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.

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.

To show that logarithmic time complexity in replacement structure would affect the GET and SET request latencies, we reproduced Cao *et al.*'s implementation of the Greedy-Dual algorithm in *GD-PQ*, which maintains all key-value pairs in a single priority queue. In the following experiments, we will compare GD-PQ with GD-Wheel and LRU in terms of GET and SET latencies in `Memcached`.

We 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 [10]. When the GD-Wheel or GD-PQ replacement policy is used,

the client additionally provides the cost of the key-value pair with the SET requests to `Memcached`.

6.3 YCSB Benchmark

The YCSB Benchmark is a load-generating tool that generates GET and SET requests based on given workload configurations. It 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 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.

During the measurement phase, each workload sends 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. For GD-PQ and GD-Wheel, we will also send 100 million GET requests for a fair comparison. For each workload executed by the YCSB benchmark, we repeated the experiments several times and the results of hit rates and recomputations costs have a variability no more than 1%. Thus we use two significant digits to report the results of all workloads.

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%)

Table 6.1 : Single Size Workload Configurations.

6.4 Workloads

6.4.1 Single Size Workloads

Table 6.1 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 test with different value sizes derived from the baseline.

6.4.2 Multiple Size Workloads

Table 6.2 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

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 6.2 : Multiple Size Workload Configurations.

value size. We select these three value sizes so that the key-value pairs in different cost groups will fall into different slab classes.

6.5 Results

6.5.1 Single Size Workload Results

Average GET/SET Request Latencies in Memcached Figure 6.1 and Figure 6.2 show the average GET and SET request latencies for the baseline workload with different replacement policies and different `Memcached` cache sizes. The average GET request latencies (Figure 6.1) are all about 220 μs with different replacement policies under different cache sizes. When dealing with GET requests, `Memcached` will send the return value right after the hash table lookup. Changing the 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 (Figure 6.2) 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

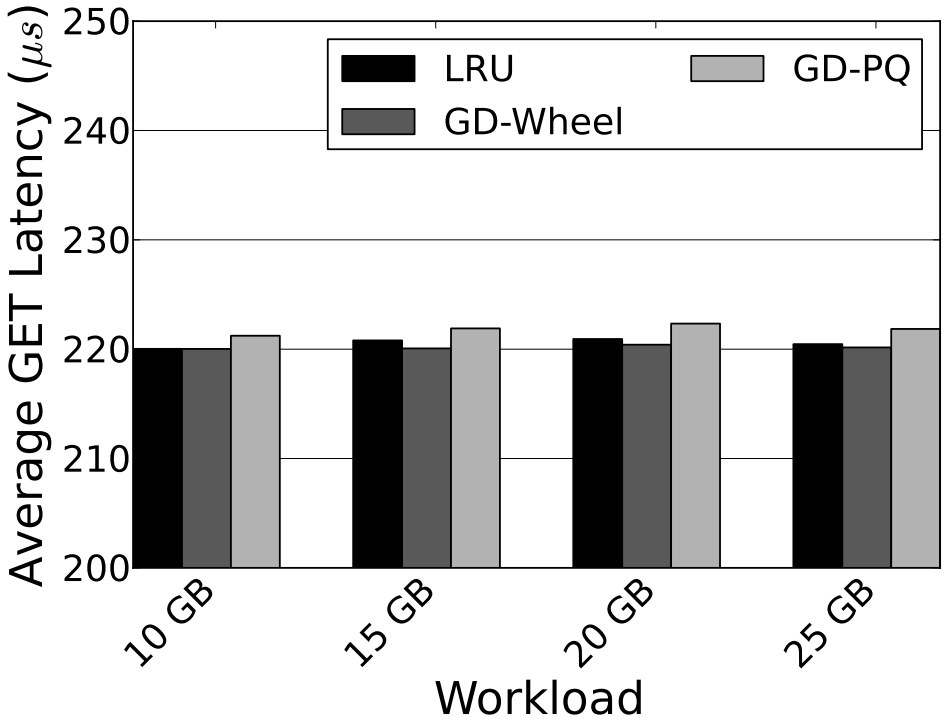


Figure 6.1 : Average GET Request Latencies (μs) for the Baseline Single Size Workload.

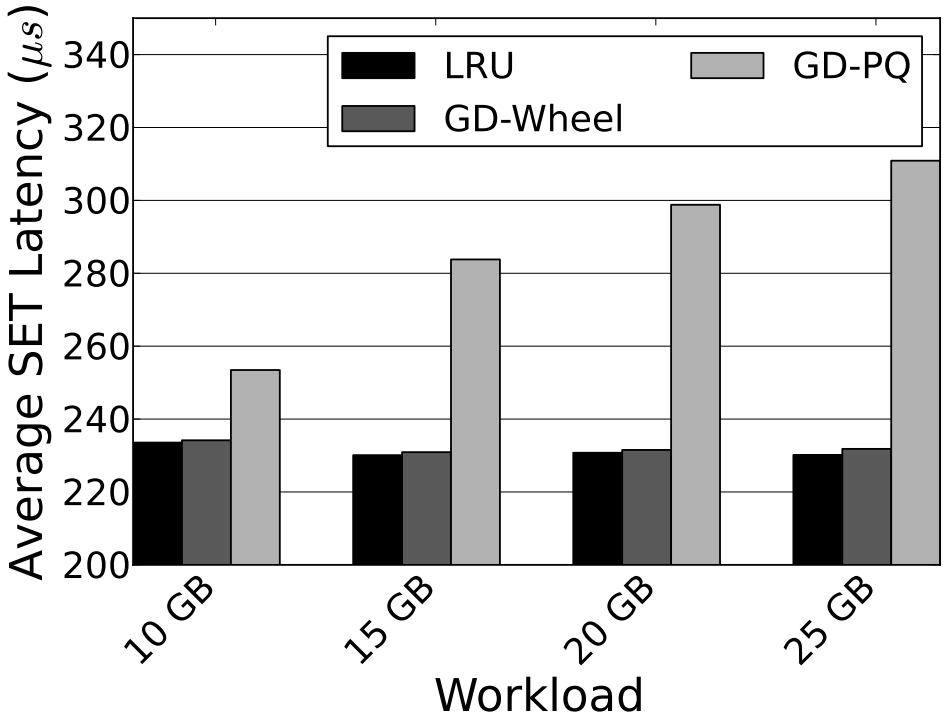


Figure 6.2 : Average SET Request Latencies (μs) for the Baseline Single Size Workload.

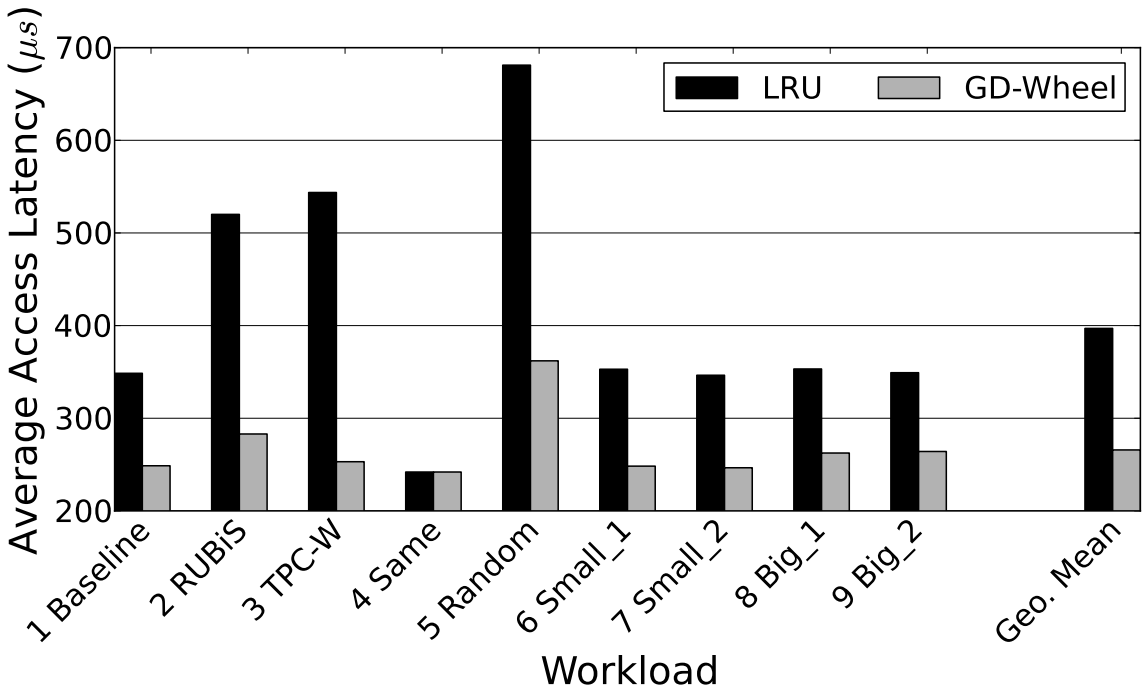


Figure 6.3 : Average Application Read Access Latencies(μs) for the Single Size Workloads.

constant time complexity of LRU and GD-Wheel, GD-PQ's logarithmic time complexity makes the SET request latency increases as cache size increases. As results above show that LRU and GD-Wheel outperform GD-PQ, we will only compare LRU with GD-Wheel for the following evaluations. And we will only use the 25 GB cache size for the following evaluations.

Average Read Access Latency Since there is no database layer in our experiments, we calculate the overall 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.

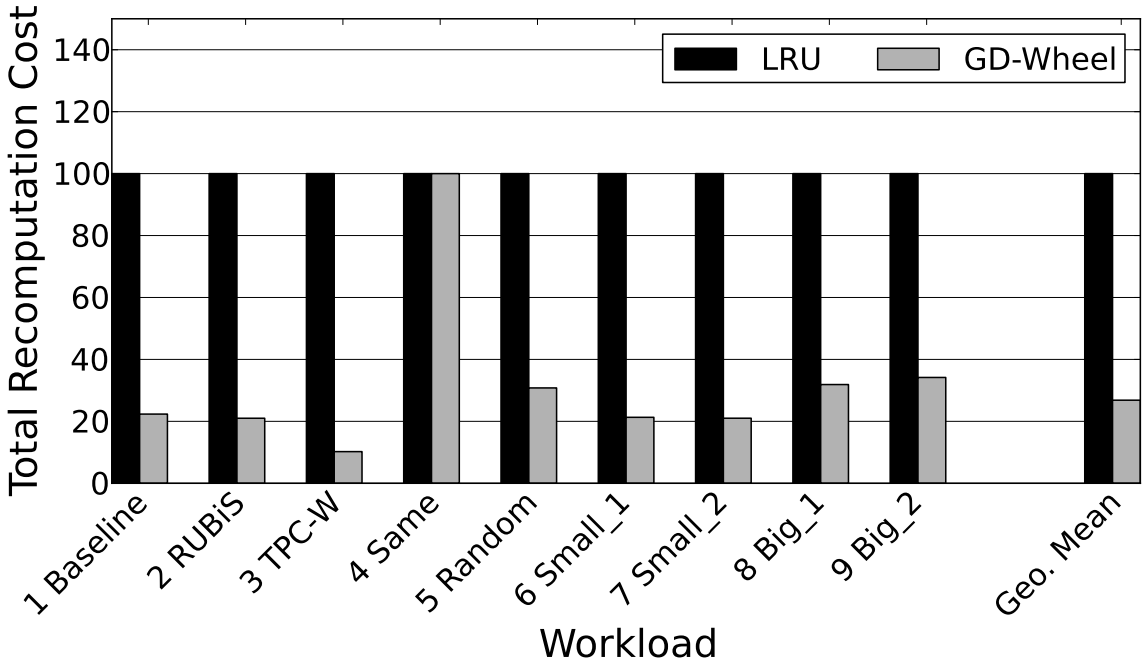


Figure 6.4 : Normalized Total Recomputation Cost for the Single Size Workloads.

Figure 6.3 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.

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 total extra access latency on cache misses. Figure 6.4 shows the normalized total recomputation cost for both LRU and GD-Wheel. All the numbers for LRU are set to 100 and

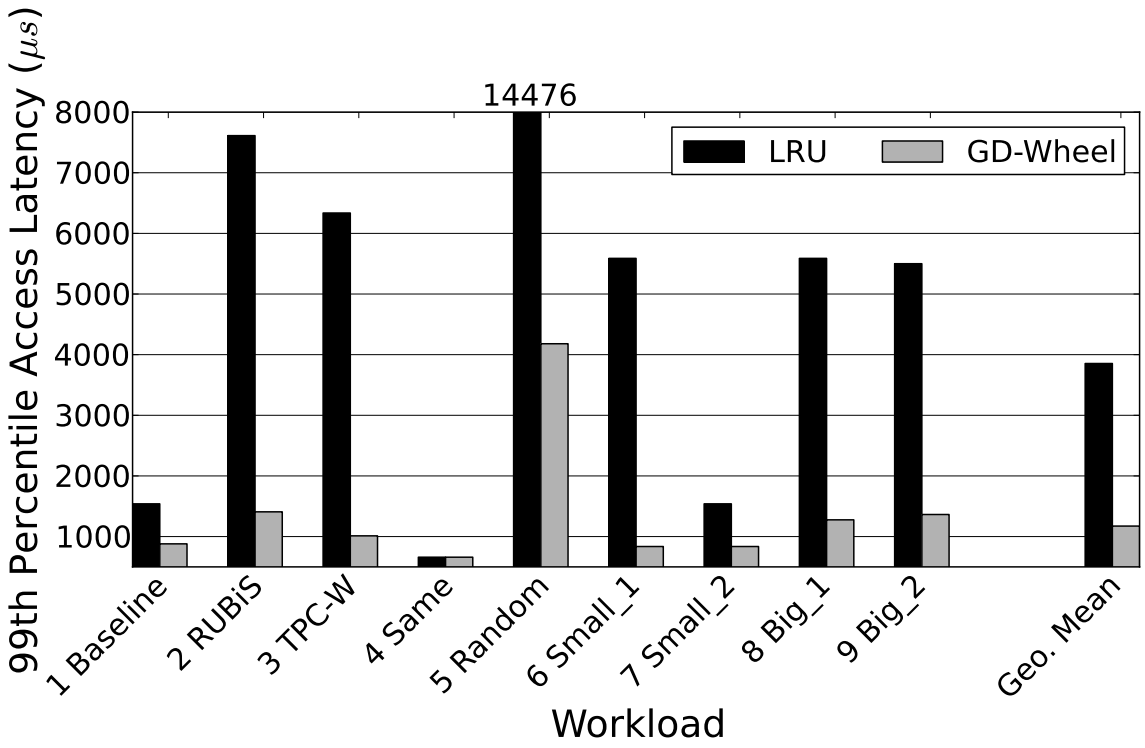


Figure 6.5 : 99th Percentile Application Read Access Latencies(μs) for the Single Size Workloads.

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 73% 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 [14]. Figure 6.5 shows the 99th percentile application read access latency (LRU's Workload 5(14476 μs) are cut off due to the space). The results show that GD-Wheel greatly reduces the 99th percentile application read access latency, by an average of 70% and as much as 85%. For workload 5 where the cost distribution is totally random, GD-

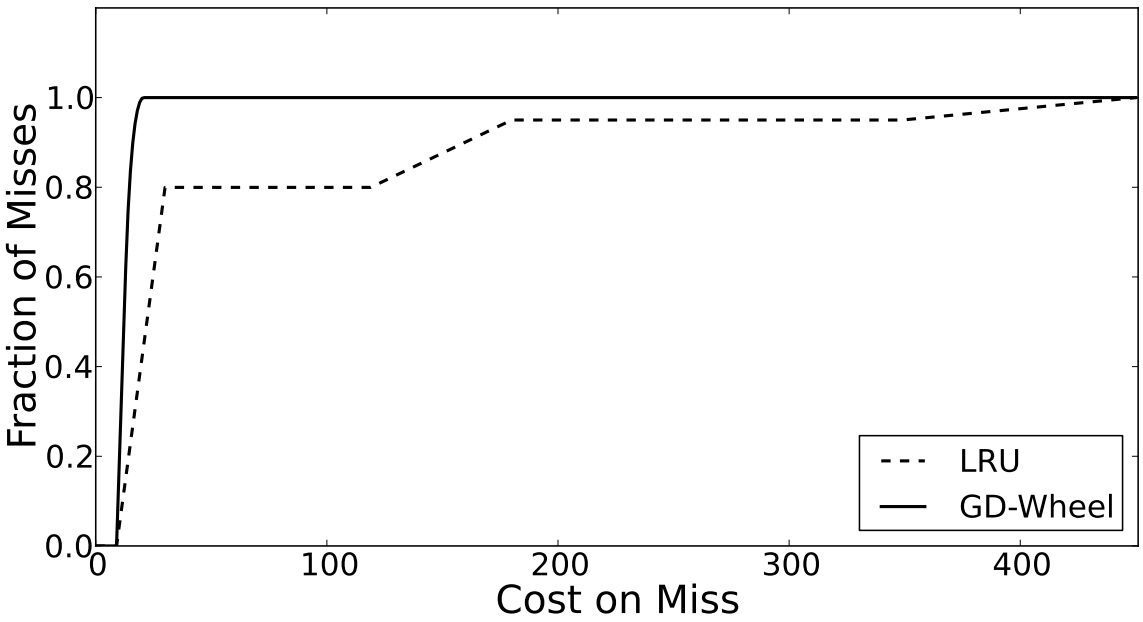


Figure 6.6 : CDF of Recomputation Costs for the Single Size Workload 1 Baseline.

Wheel keeps the 99th percentile miss cost as low as 90, while LRU's 99th percentile miss cost is as high as 324. For all the other workloads, GD-Wheel keeps the tail latencies with low miss cost no larger than 27, while LRU's tail latencies span over all cost groups. Again, the two replacement policies have the same latency in workload 4, where all key-value pairs have the same cost.

CDF of Recomputation Costs The reason for the tail read access latency improvement for GD-Wheel is that GD-Wheel avoids large recomputation costs. Figure 6.6 to Figure 6.9 show the cumulative distribution function (CDF) of recomputation costs for workloads 1, 2, 3, and 5. Since LRU and GD-Wheel perform the same in workload 4, we didn't show the CDF of the workload. Since workloads 6 to 9 perform similar to workload 1, we didn't show the CDFs of the workloads. GD-Wheel avoids high cost on misses for all of the workloads, while the recomputation costs for LRU span the cost distribution.

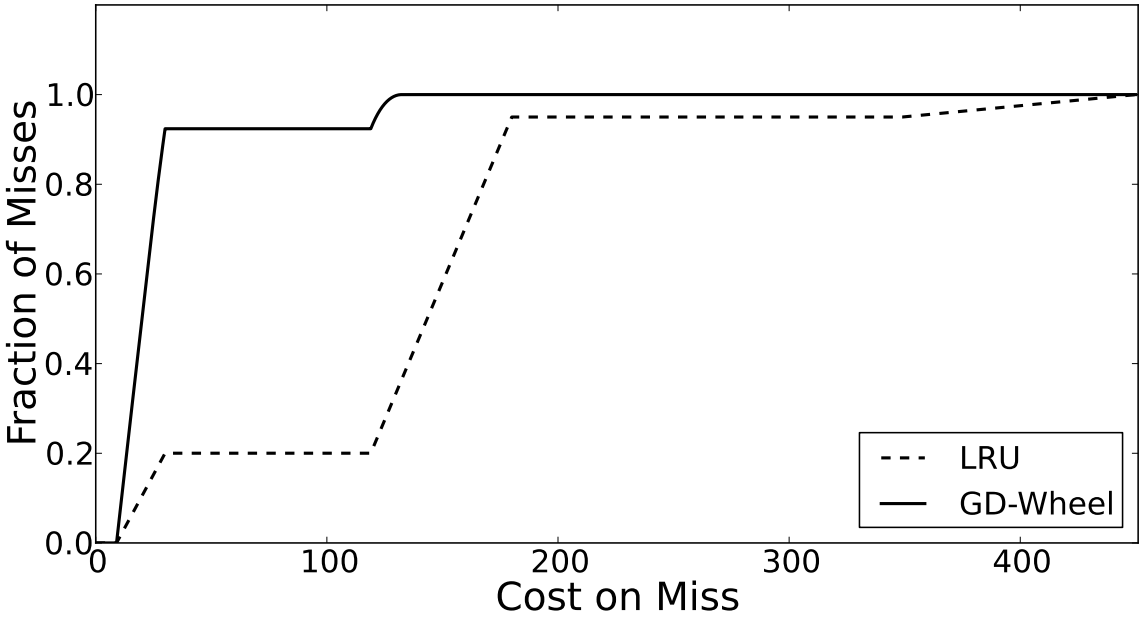


Figure 6.7 : CDF of Recomputation Costs for the Single Size Workload 2 RUBiS.

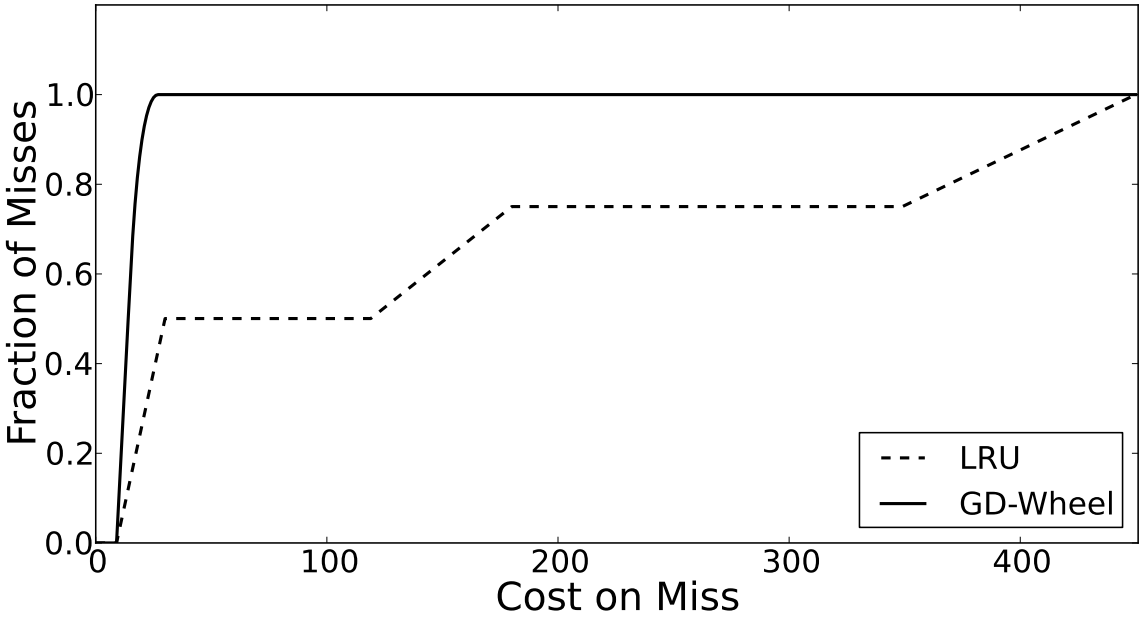


Figure 6.8 : CDF of Recomputation Costs for the Single Size Workload 3 TPC-W.

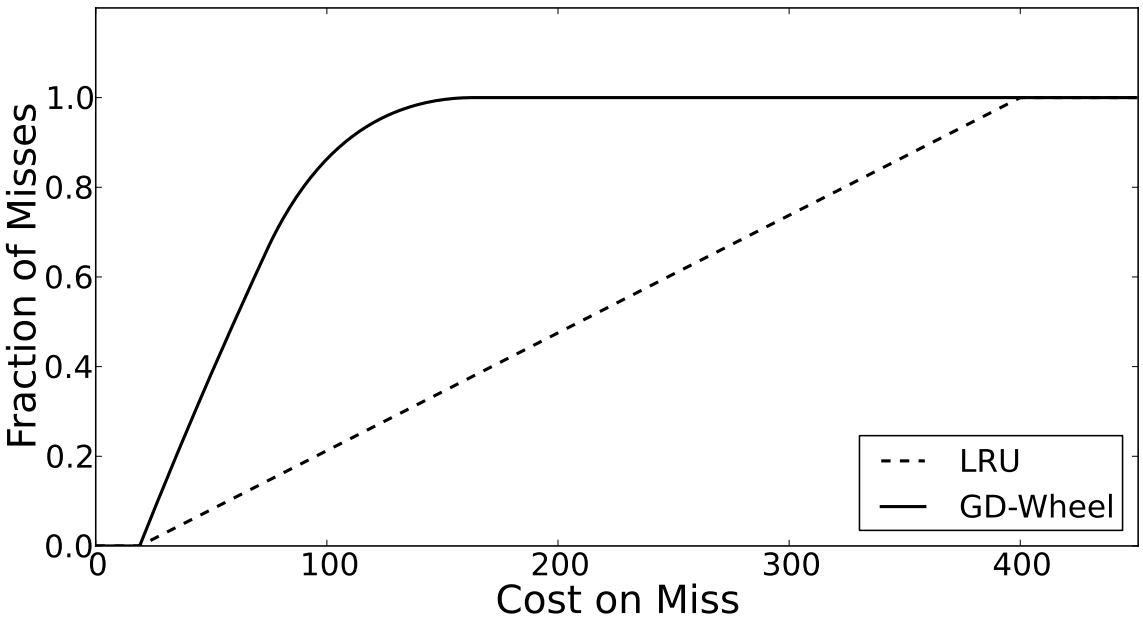


Figure 6.9 : CDF of Recomputation Costs for the Single Size Workload 5 Random.

Workload	LRU	GD-Wheel
1. Baseline	95.01%	95%
2. RUBiS	95%	94.82%
3. TPC-W	95.01%	95.01%
4. Same	95.01%	95.01%
5. Random	95.01%	94.99%
6. Small_1	94.84%	94.84%
7. Small_2	95.09%	95.1%
8. Big_1	94.82%	94.71%
9. Big_2	94.99%	94.82%

Table 6.3 : GET Hit Rates for LRU and GD-Wheel in the Single Size Workloads.

GET Hit Rate Table 6.3 shows 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 similar

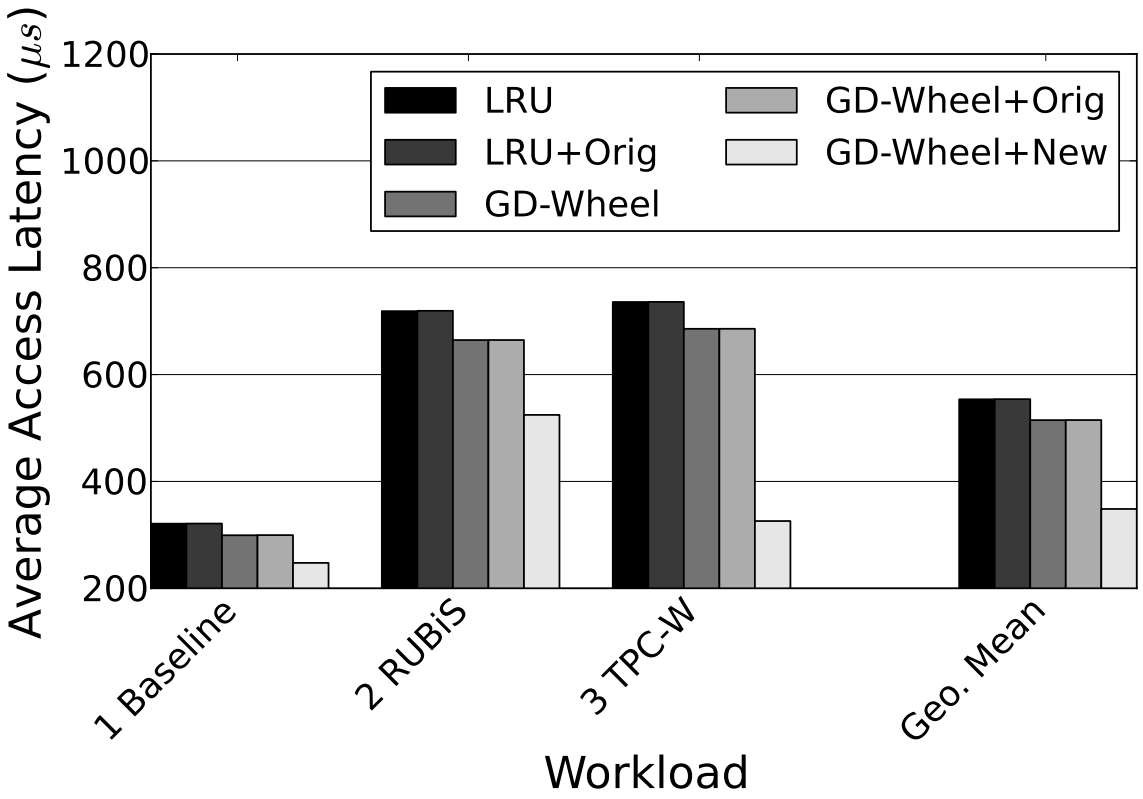


Figure 6.10 : Average Application Read Access Latencies(μs) for the Multiple Size Workloads.

hit rate as LRU.

6.5.2 Multiple Size Workload Results

Average Read Access Latency Figure 6.10 shows the average application read access latency for each workload, 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). Despite the lack of rebalancing policy, there is still some improvement achieved by GD-Wheel alone. This is because that there still exists a small cost variation in each of the three cost groups (10-30, 120-180, and 350-450).

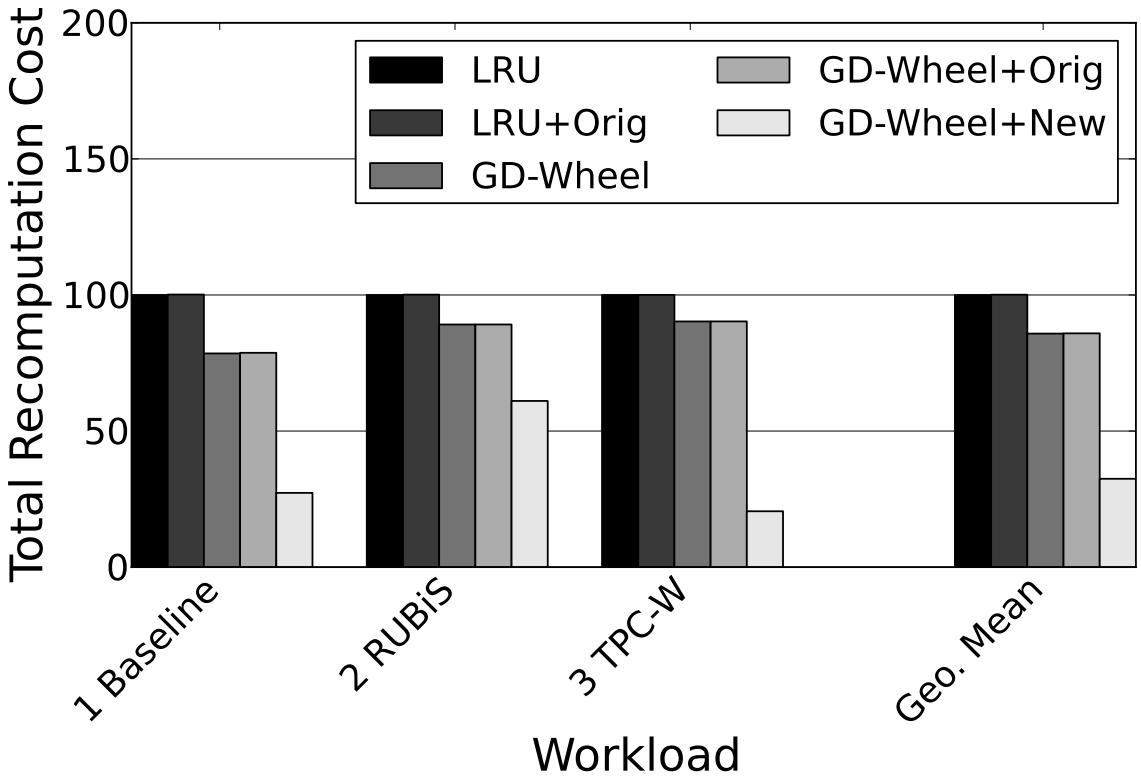


Figure 6.11 : Normalized Total Recomputation Cost for the Multiple Size Workloads.

In addition to that, our cost-aware rebalancing policy combined with GD-Wheel achieves much more 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%. These improvements are similar to what GD-Wheel achieved in our single size workloads. Such results show that our cost-aware rebalancing policy completes the cost-awareness in `Memcached` and takes the cost information among different slab classes into consideration.

Reduction of Total Recomputation Cost Figure 6.11 shows the normalized total recomputation cost for both LRU and GD-Wheel, with the original and cost-aware rebalancing

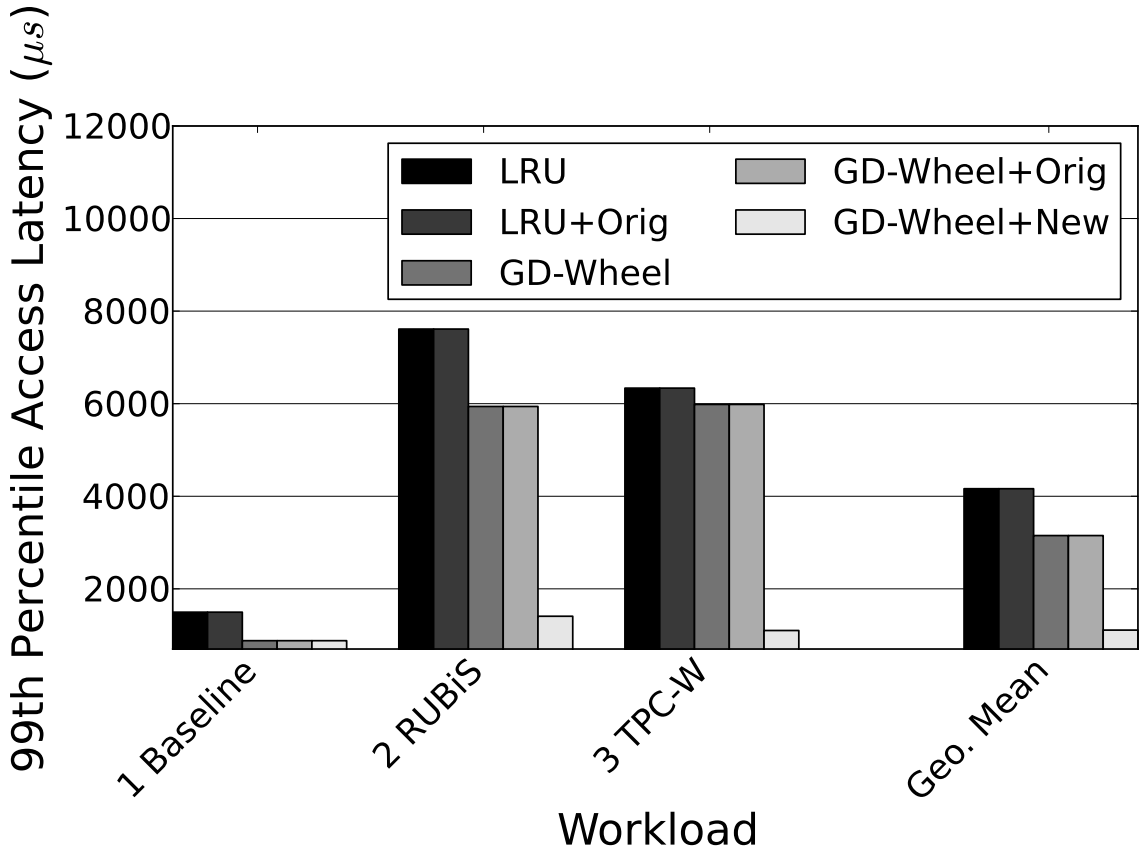


Figure 6.12 : 99th Percentile Application Read Access Latencies(μs) for the Multiple Size Workloads.

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 contrast, GD-Wheel with the cost-aware rebalancing policy greatly reduces the total recomputation cost, compared to LRU, by an average of 68% and as much as 79%.

Reduction	Avg. Read Latency	Tail Read Latency	Total Recomputation Cost
Single Avg.	33%	70%	73%
Single Max	53%	85%	90%
Multiple Avg.	37%	73%	68%
Multiple Max	56%	83%	79%

Table 6.4 : Results Summary for Single and Multiple Size Workloads.

Tail Read Access Latency Figure 6.12 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 same slab class, so that 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.

6.6 Results Summary

Table 6.4 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 improvement 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.

CHAPTER 7

Related Work

The GD-Wheel replacement policy presented in this thesis is built upon the GreedyDual algorithm [7]. 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 thesis is the first to propose and evaluate a cost-aware replacement policy for memory-based key-value stores.

This chapter presents related work in three dimensions. First we present work related to the cache replacement algorithm. Then we present different applications of the GreedyDual algorithm. Finally we present work related to improving individual key-value storage nodes in terms of throughput and cost efficiency.

7.1 Cache Replacement Algorithms

Cache replacement algorithms attempt to minimize various cost metrics, such as miss ratio, total cost, and average latency. Listed below are some of the most popular cache replacement algorithms.

- *Belady's MIN Algorithm* [15]: This algorithm is the theoretically optimal cache replacement algorithm for the paging problem. This algorithm always discards the item that will not be accessed for the longest time in the future. Since it is generally

impossible to predict the next access time of an item, this algorithm is generally not implementable in practice.

- *Random Replacement (RR)*: This algorithm randomly selects a candidate item and discards it to make space when necessary. This algorithm does not require keeping any information such as access history. This policy is the easiest to implement but generally performs poorly.

- *Least Recently Used (LRU)*: This algorithm exploits the temporal locality in the cache and evicts the least recently used item. LRU is one of the most popular replacement policies. Oftentimes, people implement an approximation of LRU such as the reference bit and the second-chance (CLOCK) algorithm.

- *Least Frequently Used (LFU)*: This algorithm maintains an access frequency count for all items and evicts the item that has been used least frequently. However, items with large frequency counts that will not be accessed again tend to remain in the cache, preventing new items from gathering enough references to stay in the cache.

- *LRU/K* [16]: The LRU/K algorithm keeps track of the times of the last K references to items. In particular, LRU/2 (K = 2), the most practical example, memorizes the times of two most recent occurrences for each item and replaces the item with the least second-most-recent occurrence. Since LRU-2 is implemented with a priority queue, it requires a logarithmic complexity in the number of items.

- *2Q* [17]: 2Q implements LRU/2 in constant time. Unlike the priority queue implementation of LRU/2, 2Q uses two queues to maintain items: items with a single reference are maintained in a FIFO queue, and items with multiple references are maintained in an LRU queue. Having constant time complexity, 2Q performs as well

as LRU/2 in terms of hit ratios.

- *Adaptive Replacement Cache (ARC)* [18]: ARC maintains two LRU lists: one list maintains items that have been accessed only once, while the other list maintains items that have been accessed at least twice. By balancing the number of items stored in the two lists, ARC takes both the recency and the frequency of access into consideration.

All the cache replacement algorithms above 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.

7.2 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 [8]. Instead of using the cost of bringing the object into the cache, the GreedyDual-Size algorithm uses the cost divided by size as the final cost of each object. 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) [19]. LARD distributes incoming front-end requests in a manner that achieves high locality in the back-end's memory caches as well as load balancing. They examined both LRU and GreedyDual-Size algorithms in the back-end's memory caches. In all simulations of Web workloads, the GreedyDual-Size algorithm achieves higher throughput than

LRU.

PAST is a large-scale peer-to-peer persistent storage utility [20]. PAST is based on an overlay network of storage nodes that cooperatively route file queries, store multiple replicas, and cache copies of popular files. For the cache management, PAST's cache replacement policy is based on the GreedyDual-Size algorithm. PAST represents the cost as the average number of routing hops required for each file. Results show that GreedyDual-Size algorithm performs better than LRU in terms of global cache hit ratio and average number of routing hops.

The related work above applied GreedyDual-Size algorithm to different caching systems. In our GD-Wheel replacement policy, we didn't include the size into consideration since `Memcached` uses separate replacement data structures for key-value pairs with different allocated sizes.

7.3 Improving Individual Key-Value Stores

There a lot work related to improving individual key-value storage nodes in terms of throughput and cost efficiency. But we are the first to propose and evaluate a cost-aware replacement policy for memory-based key-value stores. In the following of this section, we will present work related to improving both flash-based and memory-based key-value stores.

7.3.1 Flash-Based Key-Value Stores

FAWN-DS is a high-performance key-value storage system built on the FAWN cluster architecture [21]. 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.

BufferHash is an efficient flash-based data-structure that significantly lowers the cost of random hash insertions and updates on flash [22]. Instead of performing individual insertions/deletions one at a time to the hash table on flash, BufferHash performs multiple operations concurrently, resulting in a better amortized cost for each operation. BufferHash accumulates insertions in small in-memory hash tables and later pushes them into a new on-flash hash table in a batch. The on-flash tables are guarded by in-memory Bloom filters [23] to reduce unnecessary flash reads.

FlashStore is a high throughput persistent key-value store that uses flash memory as a non-volatile cache between RAM and hard disk [24]. It uses a single in-memory hash table to index all keys on flash similar to FAWN-DS, with a variant of cuckoo hashing [25] to resolve hash collisions. The hash table stores compact key signatures to reduce RAM usage. Key-value pairs are organized in a log-structure on flash to exploit faster sequential write performance.

SkimpyStash is a flash-based key-value store with much lower memory usage compared to previous work [26]. To achieve the low memory overhead, it moves most of the indexing to flash with linear chaining. Because of this hash table structure, SkimpyStash requires on average 5 flash reads per lookup, which is slower than previous work.

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 [27]. 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.

7.3.2 Memory-Based Key-Value Stores

RAMCloud is a DRAM-based storage system that provides fast crash recovery, rather than storing replicas in DRAM [28]. 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 [29]. 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 [3]. 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 [30]. 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 [31] or Infiniband [32].

CHAPTER 8

Conclusions

This thesis has shown that there exist significant cost variations among the computation results cached in key-value stores. Consequently this thesis has argued that a key-value store's replacement policy should take the cost variations into consideration, and key-value stores should let client applications specify a cost for each key-value pair via the SET operation.

As a demonstration, this thesis 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. GD-Wheel is not only for key-value stores. It could also be applied to any caching systems where cost variations exist.

This thesis has described the implementation of GD-Wheel in the `Memcached` key-value store. In addition, this thesis 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 and improved the average application read access latency as compared to LRU. It also reduced the 99^{th} percentile application read access latency which is critical in modern web applications.

In conclusion, this thesis has proposed a new cost-aware replacement policy, GD-

Wheel, which takes advantage of the cost variations inside memory-based key-value stores. By avoiding evictions on high-cost key-value pairs and therefore reducing the recomputation time outside the cache, GD-Wheel significantly reduces the average and tail application read access latencies by an average of 33% and 70%, respectively. As a result, this cost-aware replacement policy greatly improves the performance of web applications in terms of average and tail access latencies.

8.1 Future work

`Memcached`'s slab allocator stores key-value pairs in different chunk sizes that are determined during compilation. Since key-value pairs in the real world usually have varying sizes, having fixed chunk sizes may generate huge fragmentation. We plan to improve the slab allocator so that it could provide dynamic chunk sizes based on the key-value pairs stored in the cache.

Both the original rebalancing policy and the cost-aware rebalancing policy select the least recently used slab when moving slabs between slab classes. We plan to consider the cost variations during memory allocation to cluster the key-value pairs with similar costs. By allocating key-value pairs with similar costs into the same slab, we could select the slab with least total cost for reclamation.

We modified the SET request protocol so that clients could provide the cost information for each key-value pair. However, currently the GD-Wheel policy could not handle mixed operations with and without cost information. We plan to explore approaches for automatically setting a cost to the key-value pairs without user-provided cost information so that GD-Wheel could handle mixed operations with or without cost information.

Bibliography

[1] B. Fitzpatrick, "Distributed caching with Memcached," *Linux J.*, vol. 2004, pp. 5–, Aug. 2004.

[2] "Redis." http://redis.io/.

[3] 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, (Berkeley, CA, USA), pp. 371–384, USENIX Association, 2013.

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

[5] 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, (Berkeley, CA, USA), pp. 385–398, USENIX Association, 2013.

[6] S. Bouchenak, A. 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, (Berlin, Heidelberg), pp. 1–21, Springer-Verlag, 2006.

[7] N. Young, "The k-server dual and loose competitiveness for paging," *Algorithmica*, vol. 11, no. 6, pp. 525–541, 1994.

[8] 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, (Berkeley, CA, USA), pp. 18–18, USENIX Association, 1997.

[9] 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, (New York, NY, USA), pp. 25–38, ACM, 1987.

[10] 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, (New York, NY, USA), pp. 143–154, ACM, 2010.

[11] 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 *2002 IEEE International Workshop on Workload Characterization*, WWC-5, pp. 3–13, Nov 2002.

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

[13] 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, (New York, NY, USA), pp. 53–64, ACM, 2012.

[14] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, pp. 74–80, Feb. 2013.

[15] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Syst. J.*, vol. 5, pp. 78–101, June 1966.

[16] 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, (New York, NY, USA), pp. 297–306, ACM, 1993.

[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, (San Francisco, CA, USA), pp. 439–450, Morgan Kaufmann Publishers Inc., 1994.

[18] 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, (Berkeley, CA, USA), pp. 9–9, USENIX Association, 2003.

[19] 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, (New York, NY, USA), pp. 205–216, ACM, 1998.

[20] 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 Sympo-*

sium on Operating Systems Principles, SOSP '01, (New York, NY, USA), pp. 188–201, ACM, 2001.

[21] 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, (New York, NY, USA), pp. 1–14, ACM, 2009.

[22] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath, "Cheap and large CAMs for high performance data-intensive networked systems," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI '10, (Berkeley, CA, USA), pp. 29–29, USENIX Association, 2010.

[23] P. J. Desnoyers and P. Shenoy, "Hyperion: High volume stream archival for retrospective querying," in *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, USENIX ATC '07, (Berkeley, CA, USA), pp. 4:1–4:14, USENIX Association, 2007.

[24] B. Debnath, S. Sengupta, and J. Li, "FlashStore: High throughput persistent key-value store," *Proc. VLDB Endow.*, vol. 3, pp. 1414–1425, Sept. 2010.

[25] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, pp. 122–144, May 2004.

[26] B. Debnath, S. Sengupta, and J. Li, "SkimpyStash: RAM space skimpy key-value store on flash-based storage," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, (New York, NY, USA), pp. 25–36, ACM, 2011.

[27] 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, (New York, NY, USA), pp. 1–13, ACM, 2011.

[28] 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, (New York, NY, USA), pp. 29–41, ACM, 2011.

[29] 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, (New York, NY, USA), pp. 183–196, ACM, 2012.

[30] 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, (Berkeley, CA, USA), USENIX Association, 2014.

[31] 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, (Berkeley, CA, USA), pp. 31–31, USENIX Association, 2012.

[32] 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, (Berkeley, CA, USA), pp. 103–114, USENIX Association, 2013.