UNIVERSITY OF CALIFORNIA,
IRVINE

Cache Optimization for the Modern Web

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Jenny Lam

Dissertation Committee:
Professor Sandy Irani, Chair
Professor Michael Dillencourt
Chancellor's Professor Michael T. Goodrich

2015

DEDICATION

To Mom, Annie, Michael and Linda.

TABLE OF CONTENTS

*Portions of this chapter is included with permission from ACM[30].

*This chapter is included with permission from SIAM[37].

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I thank the National Science Foundation for its support under grant CCF-0916181, 1011840 and 1228639; and the Defense Advanced Research Projects Agency (DARPA) under agreement no. AFRL FA8750-15-2-0092. I also thank UCI's Computer Science and the School of ICS for ensuring my funding during my time at UCI. I thank ACM and SIAM for giving me permission to include their copyrighted material in this work. I thank my co-authors for their permission to use our unpublished work in this work.

I would like to thank my advisor Sandy Irani for giving me the opportunity to learn to do research in computer science, and my collaborators at the University of Southern California, Shahram Ghandeharizadeh and Jason Yap, for introducing me to research on cache optimization, and for providing the traces for the studies conducted in this work.

I thank Sandy Irani, Michael Carey, Michael Dillencourt, Michael Goodrich, and Phil Sheu for serving on my thesis committee. I would like to thank David Eppstein, Michael Goodrich and Michael Mitzenmacher for allowing me to work with them on the graph watermarking problem.

I would like to thank the faculty of the Computer Science department at UCI for sharing their knowledge and offering their support. In particular, I would like to thank Rina Dechter for being my mentor when I first came to UCI.

I would like to thank Lars Otten and his predecessors for maintaining a LaTeX class for the UCI thesis. You have made the process of putting this dissertation together so much easier.

I would like to thank the ICS staff, and in particular Karina Bocanegra, Kris Bolcer, Holly Byrnes, Mark Cartnal, Jason Cleaver, Cindy Kennedy, and Carolyn Simpson, for their constant helpfulness with regards to all administrative matters.

I am grateful for the friendship and camaraderie of fellow students Dmitri Arkhipov, Zach Becker, Jack Cheng, Thomas Debeauvais, Zach Destefano, Will Devanny, Timothy Johnson, Nil Mamano, Ali Mehrabi, Keren Ouaknine, Pawel Pszona, Joe Simons, and Lowell Trott. I would also like thank Maartin Löffler for introducing the weekly tea time tradition to the theory group and for hosting wonderful game nights.

I am thankful for the beauty of the UCI campus in which I was surrounded everday. I thank Piya Bose and the staff at Palo Verde for making living on campus feel like home.

I had many inspirational teachers on my academic journey: Mme Kassan, Wang Lao Shi, M Beghdali, M Vesse, Ken Sill, Coach MacKay, Herbert Enderton, Daniel Grossman, Geoffrey Mess, Yiannis Moschovakis, James White; teaching colleagues: Art Moore and Fred Feldon; professional colleagues in industry: Todd Tran, Harkeerat Bedi, Amir Khapour. I am thankful for their wisdom, mentorship and friendship.

Finally I thank my family and friends for their love and support.

CURRICULUM VITAE

Jenny Lam

EDUCATION

Doctor of Philosophy in Computer Science **2015**
University of California, Irvine *Irvine, California*

Master of Science in Computer Science **2012**
University of California, Irvine *Irvine, California*

Master of Arts in Mathematics **2007**
University of California, Los Angeles *Los Angeles, California*

Bachelor of Science in Mathematics **2007**
University of California, Los Angeles *Los Angeles, California*

Associate in Science in Physics **2003**
Santa Ana College *Santa Ana, California*

ABSTRACT OF THE DISSERTATION

Cache Optimization for the Modern Web

By

Jenny Lam

Doctor of Philosophy in Computer Science

University of California, Irvine, 2015

Professor Sandy Irani, Chair

Key-value stores are used by companies such as Facebook and Twitter to improve the performance of web applications with a high read-to-write ratio. They operate as caches for frequently requested content or data that is costly to obtain, such as the result of a computationally expensive database query. We study two design problems associated with key-value stores.

The first problem we consider is the design of eviction policies that are particularly suited to the constraints of a key-value store. Current implementations use Least Recently Used (LRU), a popular and simple eviction policy. However, LRU does not take into consideration the time to obtain an item from its source (referred to as fetch time) which can can vary widely. If the fetch times for items stored in a cache vary significantly, a more sophisticated eviction algorithm such as GreedyDual-Size provides better performance in terms of total fetch time. But GreedyDual-Size can be costly to implement. We propose an eviction policy called Cost Adaptive Multi-queue eviction Policy (CAMP) that closely approximates GreedyDual-Size's caching performance while being as fast as LRU to implement. We show that CAMP's competitive ratio is a factor of $(1 + \varepsilon)$ more than GreedyDual-Size's competitive ratio, where ε is a parameter that depends on the number of bits of precision used to compute the eviction priority of cached items.

In addition to eviction decisions, key-value stores also typically manage the placement of data objects. The current state of the art uses a technique called slab allocation in which items are assigned to one of several LRU queues according to their size. To handle changing workloads, the queues must be dynamically resized. Current schemes have been handling this problem in an ad hoc manner. We propose a variant of CAMP that manages its own memory layout and show that if it is given a modest amount of additional memory to account for fragmentation, it is competitive against an offline optimal algorithm that does not specify layout.

The second problem we investigate is the design of memory hierarchies using multiple types of memory technology for caching. Advances in storage technology have introduced many new types of storage media which present a system designer with a wide array of options in designing caching middleware. We provide a systematic way to use knowledge about the frequencies of read and write requests to individual data items in order to determine the optimal cache configuration. The ability to replicate a data item in more than one memory bank can benefit the overall performance of the system with a faster recovery time in the event of a memory failure. The key design question we are concerned with is how to best assign data items to memory banks, given that we have the option of replicating objects in order to maximize performance. Our performance model takes into account retrieval, update and recovery time. We study two variants of this problem. In the first variant which we call the *cache configuration problem*, we have a fixed budget and must decide which types of the storage media to purchase, how much of each to buy and how to place data objects in this system once the capacity of each storage medium is determined. In the second variant which we call the *subset assignment problem*, the storage hardware has already been purchased and we are solely concerned with data placement.

Both problems are NP-hard since they are generalizations of the knapsack problem. We make the reasonable practical assumption that there are many more data items than there

will be storage media, and that each storage medium is orders of magnitude larger than any single data item. These assumptions allow us to efficiently find nearly optimal solutions. Thus, for the cache configuration problem, we show that the problem is equivalent to the multiple-choice knapsack problem. We provide results from an empirical study that evaluates our algorithm in the context of a memory hierarchy for a key-value store as well as a host-side cache to store disk pages. The results show that selective replication is appropriate with certain failure rates, but that it is not advantageous to replicate data items with slim failure rates. For the subset assignment problem, we devise an algorithm loosely based on the cycle canceling algorithm for the minimum cost flow problem and give theoretical bounds for its running time. Our algorithm solves the linear programming relaxation in time $O(3^{d(d+1)}\,\mathrm{poly}(d) \cdot n \log(n) \log(nC) \log(Z))$, where d is the number of storage media, n the number of distinct data items that can be requested, Z the maximum size of any object, and C the maximum cost for storing an item.

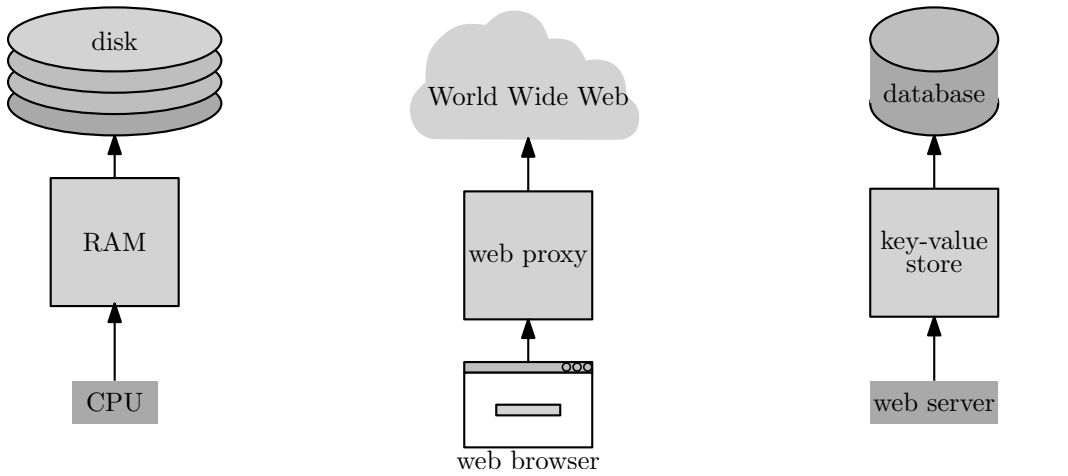
Chapter 1

Introduction

In the computing world, caching is one of the most important techniques available for the enhancement of system performance. The idea is simple: if a piece of data is used over and over again, and fetching that data takes time, we can save time by keeping a copy in a local bank of memory which we can quickly access. This local data store is the so-called cache. This idea is so fundamental that it has been used time and again in many different contexts.

Caching was first used within a single computer, under the computing paradigm known as the Von Neumann architecture. A computer traditionally consists of three parts: a central processing unit or CPU, which processes data, the main memory or RAM, and a hard disk. In the simplified caching point of view depicted in Figure 1.1a, disk is the source of all data, whereas main memory, which can be accessed orders of magnitude more quickly, plays the role of the cache. Data in disk is divided up into blocks of memory called pages, which are the smallest units of data that can be transferred to main memory.

Because main memory is more expensive than disk, it is also significantly more limited in capacity. A natural question is to decide what data to keep in that limited space. If the cache is completely full at the time we decide to cache a piece of data, we must decide which



(a) Paging or disk caching (b) Web caching (c) Database caching

Figure 1.1: Caching in three different contexts. In each case, the bottom level represents the entity that is requesting data, also known as the cache client. The top level represents the authoritative source of the data. The middle layer represents the cache. Note that these three systems are related to each other: the web browser in 1.1b is an application that could be run on a machine represented in 1.1a and the webserver in 1.1c is one of the many components hidden in the cloud in 1.1b.

other cached data to throw out. A strategy for making this decision systemically is known as an *eviction policy* or a *replacement policy*.

One eviction policy known as Least Recently Used (LRU) chooses to discard the data object that has been requested the least recently. LRU has stood the test of time to become the de-facto standard of eviction policies thanks to its simplicity and remarkable effectiveness in a wide range of situations. The reasons for its effectiveness have been thoroughly studied and documented by computer scientists since the late 1960s, and can be summed up by the fact that LRU is particularly good at predicting what will be popular in the near future. This is because, in most applications observed in practice, when an object is requested, it is likely to be requested again soon after [17, 16]. LRU simply takes advantage of this common property of request sequences called locality of reference. Despite the popularity of LRU and its variants [40, 62, 49, 39, 54, 55], there are situations that call for a different kind of eviction policy.

The mid-1990s marked the beginning of widespread use of the Internet in the West [12]. Common user experience issues such as slow webpage loading [38] gave rise to the need to improve response time. One strategy was through the use of web caches, which took the form of web-browser caches [65, 19] as well as web-proxies located throughout the Internet [10, 73, 20], as illustrated by Figure 1.1b. Web content takes many forms, such as text, image and video. Each web data object is cached in its entirety and the sizes of these objects vary significantly. Moreover, web content comes from many different sources, all of which take varying amounts of time to access, due to factors such as network traffic conditions and physical distance. In contrast, disk pages are uniform in size and take roughly the same amount of time to fetch. These differences were the motivation behind the design of an eviction policy for web caches called GreedyDual-Size [11].

Caching found another use as part of the effort to support an ever growing number of connections to webservers running complex applications. To serve a web query, a webserver may have to issue queries of its own to a database or other application-specific server. In order to avoid the recomputation cost associated with reissuing the same queries, dedicated servers called key-value stores are used as general-purpose cache [3, 48] to store the result of the most expensive and frequent database queries or other lengthy computations, as depicted in Figure 1.1c. However accessing content on these servers was still slow because of the low performance of input/output operations on disk. In 2003, Brad Fitzpatrick released memcached [22, 23], a distributed caching software that stores data in main memory. By moving cache from disk to main memory and deploying caching servers loaded with RAM, webservers were better able to handle applications with a high read-to-write ratio. At the time of this writing, memcached is the most popular key-value store manager, and is used by companies such as Facebook [60], Twitter [64] and Wikipedia [72].

This thesis focuses on caching in key-value stores and consists of two parts corresponding to two types of design problems related to caching middleware. The first problem we investigate is the

design of eviction policies for an in-memory key-value store such as memcached. Specifically, Chapter 2 presents a cache eviction policy named Cost Adaptive Multi-queue eviction Policy or CAMP, which achieves a better caching performance than current implementations of memcached, which use LRU, without the implementation overhead of GreedyDual-Size. Chapter 3 introduces an eviction policy based on CAMP which also decides the placement of data items in memory.

The second problem we investigate is the design of a multi-level caching middleware that goes beyond using just RAM and disk to take advantage of new, faster and cheaper, memory technologies. Specifically, we propose to study the static assignment of cacheable data to subsets of the memory banks which make up the multi-level cache. We assume that we have at our disposal several types of storage media, each with different read and write latencies, bandwidths, failure rates and prices. We also assume an offline input model of requests, that is, we have a probability distribution over the set of data that can be requested. In Chapter 4, we show how to model the expected time to service a request in terms of these input parameters. We propose an algorithm that determines based on a fixed budget how much, if any, of each type of memory to use when building our cache. The algorithm also gives an optimal placement of data items to the memory banks of the cache. The placement may entail replicating an item and assigning it to multiple memory banks, which can help mitigate the impact of a high hardware failure rate. Finally, in Chapter 5, the hardware makeup of the cache is already determined in that there is a set of memory banks whose capacities are given as part of the problem input. The goal is to place each item on a subset of the memory banks so that the capacities of each memory bank is not exceeded and the total cost is minimized.

Chapter 2

CAMP: a Cost Adaptive Multi-queue eviction Policy*

2.1 Introduction

To increase the responsiveness of its web services, a large website such as Facebook may include a general purpose caching layer as part of its distributed architecture. The caching layer consists of servers dedicated to caching content from various applications, which must share this resource despite differences in their access patterns, the size of the objects being cached (which are called key-value pairs), and the time to obtain the key-value pairs from the source [70]. An eviction policy that only takes into account one of these factors may cause different application workloads to impact one another negatively, thus decreasing the overall effectiveness of the caching layer. As an example, consider two applications that are part of a social networking site: one shows the profile of members of the social network whereas the other determines the advertisements to be displayed. Suppose that there are

*Portions of this chapter is included with permission from ACM[30].

millions of key-value pairs corresponding to these member profiles, and that each is computed through a simple database lookup that executes in a few milliseconds. Suppose also that the second application consists of thousands of key-value pairs computed by a machine learning algorithm that processes Terabytes of data and requires hours of execution. With limited memory capacity and a high frequency of access for member profile key-value pairs, a simple algorithm that manages memory using recency of references such as Least Recently Used (LRU) will evict most of the key-value pairs of the second application.

One possible way to remedy this problem is by manually partitioning the available memory into disjoint pools and let each pool be managed using LRU. Key-value pairs with similar costs would be grouped together and each group assigned to a different pool [60]. In our example, we would have two pools: one for the key-value pairs corresponding to members profiles and one for those corresponding to advertisements. An important drawback of this approach is its necessity for a expert familiar with the different classes of applications to come in and identify the pools, construct grouping of key-value pairs, and assign each group to a pool. Every time the service provider introduces a new application or discontinues an existing one, the expert must be involved again in these assignment and rebalancing tasks.

In this chapter, we introduce a cache eviction policy called Cost Adaptive Multi-queue eviction Policy (CAMP). Unlike LRU, CAMP takes into account both the size and recomputation cost of key-value pairs, as it is an approximation of GreedyDual-Size [11]. As a result, it achieves better caching performance. Unlike a standard heap-based implementation of GreedyDual-Size, CAMP uses LRU queues, which results in lower overhead associated with the decision making. These LRU queues are dynamically constructed based on the size and cost of key-value pairs. The number of such queues depends on the distribution of costs and sizes of the key-value pairs. Because CAMP manages these LRU queues without partitioning memory, there is no need for manual intervention from an expert. Furthermore, CAMP is robust enough to prevent an aged expensive key-value pair from occupying memory

indefinitely. Such a key-value pair is evicted by CAMP as competing applications issue more requests.

CAMP is parameterized by a variable which controls the amount of precision in the computation of eviction priorities, and which affects implementation time as well. We show that CAMP is $(1+\varepsilon)k$-competitive, where k is the competitive ratio of GreedyDual-Size, ε which is a function of the precision parameter. At the highest level of precision, CAMP's eviction decisions are essentially equivalent to those made by GreedyDual-Size. Our empirical results show that CAMP does not suffer any degradation in the quality of its eviction decisions at lower precisions. Moreover, it is able to make those decisions much more efficiently than GreedyDual-Size. GreedyDual-Size requires an internal priority queue to determine a key-value pair to evict from the cache. The time to maintain its data structures consistent in a thread-safe manner is expensive because it requires synchronization primitives [41] with multiple threads performing caching decisions. Moreover, CAMP performs significantly fewer updates of its internal data structures than GreedyDual-Size, reducing the number of times it executes the thread-safe software dramatically.

This chapter is organized as follows. Section 2.2 starts with a description of GreedyDual-Size and CAMP. Section 2.3 presents a simulation study comparing CAMP to LRU and the pooled approach that partitions resources, demonstrating its superiority. Section 2.4 describes an implementation of CAMP using a variant of Twemcache and compares this implementation with the original that uses LRU. Our results demonstrate that CAMP is as fast as LRU and provides superior performance as it considers, in addition to recency of requests, the size and the cost of the key-value pairs. Related work is described in Section 2.5 and a conclusion is given in Section 2.6.

2.2 Algorithms GreedyDual-Size and CAMP

The algorithm GreedyDual-Size (Algorithm 1) was developed in the context of replacement policies for web proxy caches. It captures many of the benefits of LRU and considers the fact that data objects on the web have varying sizes and incur varying time delays to retrieve depending on their location and network traffic [11]. The same principles apply in the context of maintaining the identity of key-value pairs occupying the memory of a key-value store, although the cost of an object in the key-value store setting may denote computation time (or some other quantity) instead of retrieval time. Even though the algorithm is applicable to a wide variety of settings in which caches are deployed, we adopt the terminology used for cache augmented database management systems [32].

GreedyDual-Size is based on an algorithm called GreedyDual, developed by Neal Young [75], that handles objects of varying cost but uniform size. GreedyDual-Size assigns a value $H(p)$ to each key-value pair p in the key-value store. $H(p)$ is computed from a global parameter, L, as well as from size(p), the size of p and cost(p), the cost of p. The value of $H(p)$ approximates the benefit of having that key-value in the key-value store. When there is insufficient memory to accommodate an incoming key-value pair, the algorithm continually evicts the key-value pair with the lowest value until there is room in the key-value store to store the incoming key-value pair.

Algorithm 1 GreedyDual-Size

```
function INITIALIZATION
    L ← 0
function SERVE(item i)
    if i is not in memory M then
        while the amount of free space in the cache is less than size(i) do
            evict the item j with the smallest H(j)
            L ← min_{j∈M} H(j)
        bring i into memory
    H(i) ← L + cost(i)/size(i)
```

The following proposition is useful in understanding how GreedyDual-Size works:

Proposition 2.1.

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

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

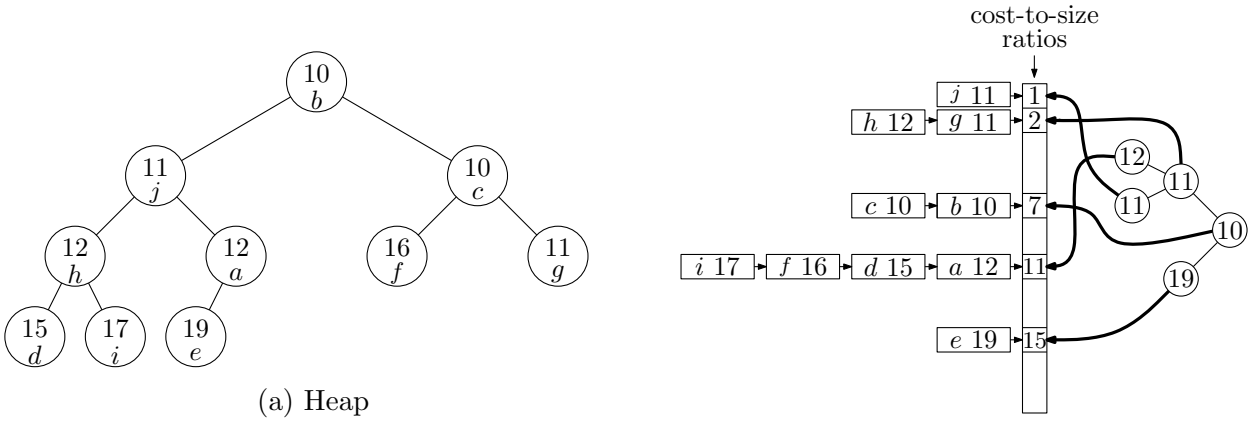
For claim 2, L will not exceed $H(p)$ for any p in the cache because its new value (assigned in line 2 or 6) is the smallest $H(p)$ among all the key-value pairs in the cache. Any eviction performed in line 5 only makes it easier to satisfy claim 2. Finally, when a key-value pair is brough into the key-value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

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

An implementation of GreedyDual-Size must maintain a data structure to identify and delete the key-value pair with the minimum priority efficiently. Typically, key-value pairs are maintained in a data structure that can retrieve and delete the key-value pair with the minimum priority. Normally this is accomplished by an implementation of a priority queue like a binary heap, which is a tree-based implementation of a priority queue which maintains the property that the priority of any node in the tree is at most the priority of its children, or the theoretically efficient Fibonacci heap [24]. The worst-case performance of any priority queue is a $\log n$ cost per operation, where n is the number of key-value pairs in the priority queue. For extremely large key-value stores, such as those in use by Facebook and Twitter, overhead that scales even logarithmically as a function of the number of key-value pairs in the key-value store results in a significant cost that could potentially be avoided.

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

With GreedyDual-Size, the *priority* or H value (the two terms will be used interchangeably) of every key-value pair in the key-value store is the sum of two values: the global non-decreasing variable L and the cost-to-size ratio of the key-value pair. CAMP rounds the priority for every key-value pair by rounding the cost-to-size ratio before adding it to L. The rounding scheme results in a smaller set of possible cost-to-size values for key-value pairs stored in the key-value store. CAMP takes advantage of the rounding by grouping the key-value pairs



(a) Heap

(b) CAMP's priority queue

Figure 2.1: A heap used in a straightforward manner (2.1a) contains many more nodes than the heap in CAMP's LRU-heap hybrid (2.1b) when there are only a few distinct cost-to-size ratios.

in its data structure according to the cost-to-size ratio instead of by priority value. The eviction decisions between CAMP and GreedyDual-Size differ slightly for two reasons. First, CAMP rounds the cost-to-size ratio in determining the H value of a key-value pair. Second, in evicting the key-value pair with smallest priority, CAMP breaks ties according to LRU, whereas GreedyDual-Size breaks ties arbitrarily.

Figure 2.1 shows the storage schemes for GreedyDual-Size and CAMP, with each circle denoting the H value of a key-value pair. Figure 2.1a shows a typical priority-queue-based implementation of GreedyDual-Size in which a set of key-value pairs are stored in a heap based on their priority value. Figure 2.1b shows CAMP's data structure in which key-value pairs are grouped into queues according to their cost-to-size ratio. Key-value pairs in a queue are ordered according to their priority which is their H value. CAMP maintains a heap containing the priority of the data item at the head of every queue. Thus, to identify a candidate key-value pair to evict from the key-value store, CAMP locates the key-value pair with the smallest priority among the heads of each of the queues.

CAMP's implementation is efficient based on the following key observation. If the key-

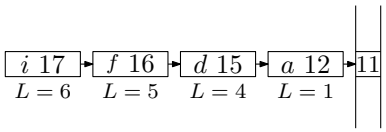


Figure 2.2: An LRU queue within CAMP.

value pairs within each queue are stored according to LRU, then the key-value pairs are automatically ordered according to their priority. For this reason, the queues maintained by CAMP are termed LRU queues. To understand why this observation holds, recall that all the items within an LRU queue have the same cost-to-size ratio. Furthermore, the H value of a key-value pair is the value of L at the time of its last request plus its cost-to-size ratio. Since L increases over time, a key-value pair that is requested earlier on will have a smaller H value and appear towards the front of the queue, whereas a key-value pair that is referenced more recently will have a larger H value and appear towards the end of the queue. In particular, the first key-value pair in each queue has the smallest priority.

As an example, consider the LRU queue in Figure 2.2 which is one of the five queues shown in Figure 2.1b. It points to the array value 11, containing all the key-value pairs that have cost-to-size ratio 11. Each node shows the key-value pair's H value, and the shown L value is the value of L at the time of its last request. Since L increases over time and key-value pairs are inserted at the tail of the queue, key-value pairs are ordered by the value of L at the time of the request. Since all these key-value pairs have similar cost-to-size values, they are also ordered by their H value. Specifically, a is the least recently requested key-value pair in its queue.

With CAMP, the complexity of processing a key-value store hit for a referenced key-value pair is the complexity to update the LRU queue ($O(1)$) plus the complexity to update the heap. The worst case for the latter is logarithmic in the number of non-empty queues instead of the number of key-value pairs in the key-value store since the nodes in the heap tree structure of CAMP identify LRU queues, see Figure 2.1b. With our implementation of

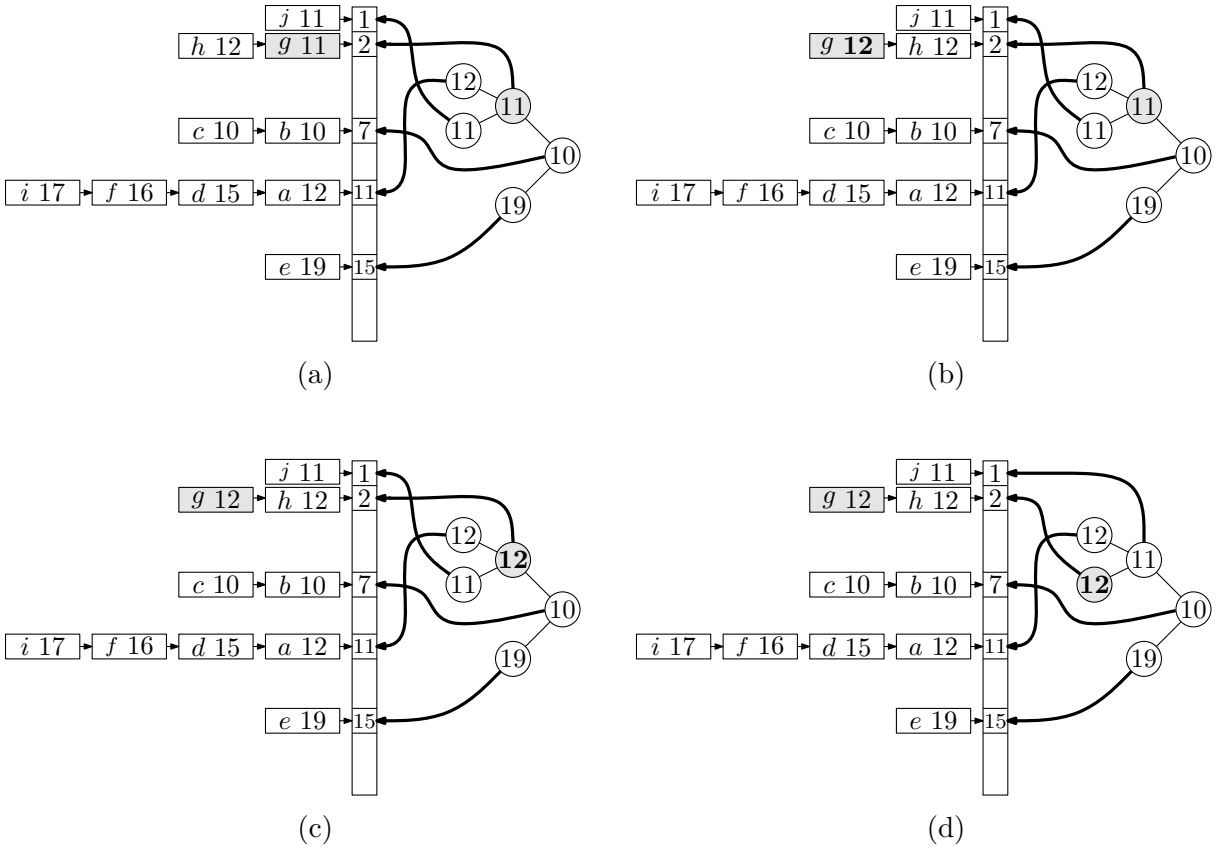


Figure 2.3: CAMP update on a key-value store hit. If g is requested (2.3a), it is moved to the back of its queue (2.3b), and the heap is updated accordingly (2.3c and 2.3d).

CAMP, we chose to use an 8-ary implicit heap as suggested by the recent study [47] on priority queues. Here, 8-ary means with branching factor at most 8. Moreover, a heap is implicit if it uses the usual array implementation rather than with pointers.

To illustrate the processing of a key-value store hit using the same running example of Figure 2.1, consider a new reference for g. The key-value store locates g using a hash table (Figure 2.3a), moves it to the end of its LRU queue (see Figure 2.3b), and updates its value to $10 + 2 = 12$ where 10 is the minimum priority (L value) and 2 is the cost-to-size ratio of g. Now, the new head of the queue has priority 12. CAMP updates the value of the heap node pointing to this queue (Figure 2.3c), causing the heap to be updated as shown in Figure 2.3d.

One way to improve performance is by limiting the number of LRU queues. We can do so by

regular rounding	CAMP's rounding
10110**1011** → 10110**0000**	1011**01011** → 1011**00000**
00101**0011** → 00101**0000**	001010**011** → 001010**000**
00000**1010** → 00000**0000**	00000<u>1010</u> → 000001010
00000**0111** → 00000**0000**	000000<u>111</u> → 000000111

Table 2.1: Rounding with (binary) precision 4

assigning key-value pairs with "similar" cost-to-size values to the same queue. Similarity in this context has a specific meaning, in that values that have different orders of magnitude should remain distinct. Therefore, a rounding scheme that simply truncates a fixed number of bits will not work. Instead, CAMP uses the integer rounding scheme described in [53]. Given a number x, let b be the order of its highest non-zero bit. To round x to precision p, zero out the $b-p$ lower order bits or, in other words, preserve only the p most significant bits starting with b. If $b \leq p$, then x is not rounded. Table 2.1 illustrates the difference between these rounding schemes with examples. With regular rounding, too much information is kept for large values and too little information is kept for small values. Since we don't know the range of values a priori, we don't know how to select p to balance the two extremes. Therefore, we prefer the amount of rounding to be proportional to the size of the value itself (right column).

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

Proposition 2.2. If the original values for the cost-to-size ratio are integers in the range $1, \ldots, U$, then the number of distinct rounded values is at most $(\lceil \log_2(U+1) \rceil - p + 1)2^p$ where p is the selected precision.

Proof. Let x denote the value to be rounded. Since x is in the range 1 through U, the binary representation of x uses at most $\lceil \log_2(U+1) \rceil$ bits. Bit locations are numbered 1 through $\lceil \log_2(U+1) \rceil$ with 1 being the lowest order bit. Let b be maximum of p and the

location of the highest order non-zero bit in the binary representation of x. The scheme zeroes out all bits except those in locations $b, b-1, \ldots, b-p+1$. There are at most $\lceil \log_2(U+1) \rceil - p + 1$ possible values for b. For each value of b, there are 2^p possible rounded values encoded in bits $b, b-1, .., b-p+1$. Thus, the total number of distinct rounded values is $(\lceil \log_2(U+1) \rceil - p + 1)2^p$. □

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

Proposition 2.3. The competitive ratio of CAMP is $(1+\varepsilon)k$, where $\varepsilon = 2^{-p+1}$, where k is the ratio of the cache size to the size of the smallest key-value pair.

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

Now let σ be a sequence of requests and let $\bar{\sigma}$ be the same request sequence but with rounded cost-to-size ratios. Define CAMP(σ) to be the total cost of CAMP on input σ and let OPT(σ) be the total cost of the optimal offline algorithm on input σ. CAMP makes the same eviction decisions on σ as it does on $\bar{\sigma}$ because it rounds the cost-to-size ratios in σ. However, it pays potentially a factor of $(1+\varepsilon)$ more on each cache miss. Therefore CAMP(σ)/$(1+\varepsilon) \leq$ CAMP($\bar{\sigma}$). CAMP makes the same decisions as GreedyDual-Size

on $\bar{\sigma}$ because the values are already rounded, so $\text{CAMP}(\bar{\sigma}) = \text{GreedyDual-Size}(\bar{\sigma})$. Since we know that GreedyDual-Size is k-competitive, $\text{GreedyDual-Size}(\bar{\sigma}) \leq k\text{OPT}(\bar{\sigma})$. Finally, OPT will have at least as large an overall cost on σ as it will on $\bar{\sigma}$ because all the cost-to-size ratios are at least as large which means that $\text{OPT}(\bar{\sigma}) \leq k\text{OPT}(\sigma)$.

Putting it all together, we get

$$\frac{\text{CAMP}(\sigma)}{1+\varepsilon} \leq \text{CAMP}(\bar{\sigma}) = \text{GreedyDual-Size}(\bar{\sigma}) \leq k \cdot \text{OPT}(\bar{\sigma}) \leq k \cdot \text{OPT}(\sigma),$$

and CAMP is $(1+\varepsilon)k$-competitive. □

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

The rounding scheme makes no a priori assumptions as to the values of the cost-to-size ratios other than assuming that the ratio of the smallest to largest cost-to-size ratio is bounded by

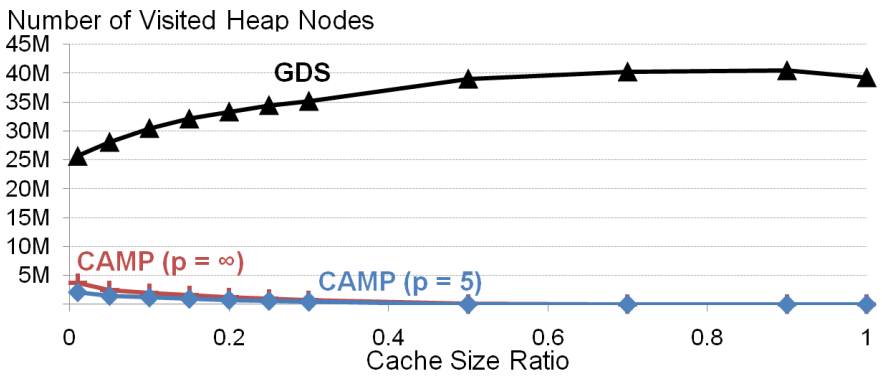


Figure 2.4: Number of visited heap nodes as a function of the cache size ratio. The label GDS stands for GreedyDual-Size.

U. Converting all values to an integer is just a mathematically convenient way of expressing that assumption. The goal is to use the range 1 to U as effectively as possible in expressing the range of cost-to-size ratios. The larger the value of U, the more space that must be set aside for potential LRU queues. The conversion from fractional values to integers is achieved by multiplying all values by a fixed multiplier and rounding to the nearest integer. Selecting a large number for the multiplier would result in large rounded values and would require a large upper bound U. This explains why we do not simply use the cache size for the multiplier.

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

Figure 2.4 compares the number of visited heap nodes in a heap-based implementation of GreedyDual-Size and in CAMP when run using the trace-driven simulation of Section 2.3. This quantity is an indication of the amount of runtime overhead of each implementation: in the case of GreedyDual-Size, this is the number of nodes that are visited when the heap is updated due to an insertion or deletion. In the case of CAMP, insertions and deletions from the queue comprise a constant time update to an LRU queue as well as the occasional

update to the heap when the head of an LRU queue changes.

There are two contributing factors to CAMP's significantly smaller number of node visits. First, the number of nodes in GreedyDual-Size's heap is equal to the number of key-value pairs in cache whereas the number of nodes in CAMP's heap is equal to the number of non-empty LRU queues, which is very small as noted in Figure 2.5b. Since the number of node visits required by a heap update grows logarithmically with the number of nodes in the heap, GreedyDual-Size's heap updates can take longer than CAMP's. The second contributing factor is that GreedyDual-Size makes more heap updates than CAMP does. In particular, GreedyDual-Size updates its heap every time the priority of a key-value pair is updated. On the other hand, CAMP only does so whenever the priority value of the head node of an LRU queue changes, or when an LRU queue is created or deleted.

Figure 2.4 shows the number of visited nodes by GreedyDual-Size increases as a function of the memory size. This trend is reversed with CAMP. The GreedyDual-Size curve increases because the number of nodes in the heap is equal to the number of items in the key-value store and there are many more data items with a larger memory size. In contrast, the CAMP curve decreases because the number of heap nodes, which is equal to the number of non-empty LRU queues, remains constant as a function of cache size. But since more items can be stored when the cache size increases, there are fewer updates to the cache. Hence, the decreasing curve.

2.3 Evaluation

We used a social networking benchmark named BG [5, 6, 26, 27] to generate traces of key-value references from a cache augmented database management system [32]. BG emulates members of a social networking site viewing one another's profile, listing their friends, and

Miss rate	Number of requests that observe a cache miss divided by number of issued requests.
Cost-miss ratio	Sum of the costs of key-value references that observe a cache miss divided by the sum of the costs for all key-value references.
Cost	Amount of time required to compute a key-value pair.
Cache size ratio	Memory size divided by the total size of the key-value pairs competing for the memory.

Table 2.2: Definitions of terms used.

other interactive actions. The benchmark is configured to reference keys using a skewed pattern of access with approximately 70% of requests referencing 20% of keys. A trace file consists of approximately 4 million rows. Each row identifies a referenced key-value pair, its size, and cost. Cost is either the time required to compute the key-value pair by issuing queries to the RDBMS or a synthetic value selected from {1, 100, 10K}, which is a set of values chosen to simulate widely varying costs between key-value pairs. With the latter, each key-value pair is assigned one of the three possible values with equal probability. Once a cost is assigned to a key-value pair, it remains in effect for the entire trace.

We implemented a simulator that consists of a key-value store and a request generator to read a trace file and issue requests to the key-value store. The key-value store manages a fixed-size memory that implements either the LRU or the CAMP algorithm. Every time the request generator references a key and the key-value store reports a miss for its value, the request generator inserts the missing key-value pair in the key-value store. This results in evictions when the size of the incoming key-value pair is larger than the available free space.

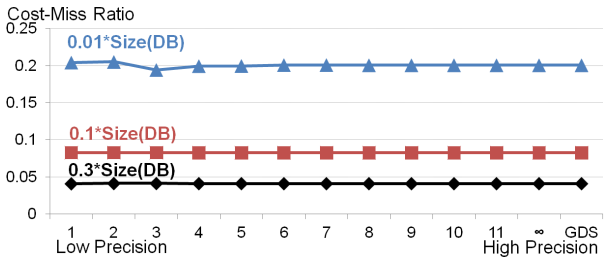
The simulator quantifies two key metrics: miss rate and cost-miss ratio. *Miss rate* is the total number of requests that result in a key-value store miss divided by the total number of requests in the sequence. *Cost-miss ratio* is obtained by summing the costs for each request that results in a key-value store miss divided by the sum of the costs for all the requests. With both, the first request to a particular key-value pair in the trace (called a *cold* request) is not counted because any algorithm will fault on such requests. Since CAMP is tuned to

minimize the total cost of serving the requests in the sequence, the cost-miss ratio is the primary metric used to quantify performance. In the following, we report these metrics as a function of either the precision used by the CAMP algorithm or the *cache size ratio*. The latter is the size of the key-value store memory divided by the total size of the unique objects in the trace file.

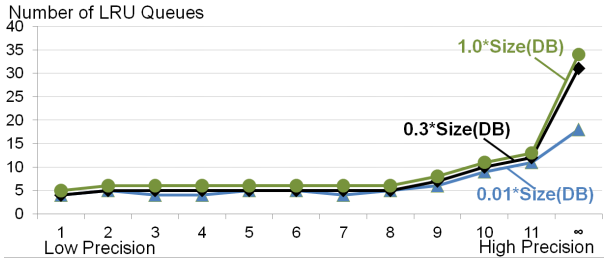
The precision of CAMP is a parameter that can be tuned for performance optimization. Small values of precision result in fewer LRU queues. With larger values of precision, replacement decisions are more finely tuned to differences in the cost-to-size ratio for each key-value pair. The graph in Figure 2.5a shows the cost-miss ratio for CAMP as a function of the precision value. The three curves show the results for three different cache sizes. For the version labeled ∞, no rounding is done after the initial cost-to-size ratio is rounded to an integer. In other words, this version corresponds to the standard GreedyDual-Size algorithm. Figure 2.5a shows that there is almost no variation in cost-miss ratios for different precisions. More importantly, there is almost no difference between the cost-miss ratios of CAMP and standard GreedyDual-Size.

Figure 2.5b shows the number of distinct LRU queues maintained by CAMP as a function of precision. The maximum number of queues possible is the number of distinct possible values for the cost-to-size ratio of the key-value pairs for a particular precision value. However, the key-value store may not hold a key-value pair with a particular cost to size value, so at any given point in time, many of the queues are empty. Figure 2.5b shows the actual number of non-empty queues at the end of the trace. Even for a very low level of precision, CAMP has at least five non-empty queues and outperforms LRU that has only one queue, see Figure 2.5c.

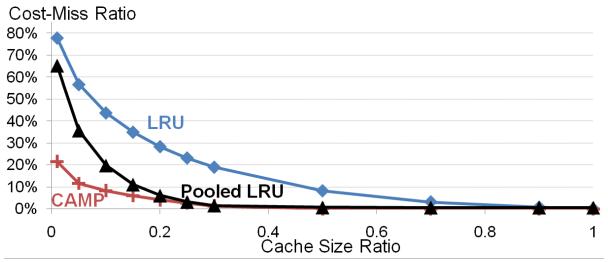
In Figure 2.5c, Pooled LRU is the partitioned-memory scheme described in [60]. This approach partitions the available memory into distinct pools. Each pool employs LRU to manage its memory. Those key-value pairs with similar costs are grouped together according to their cost.



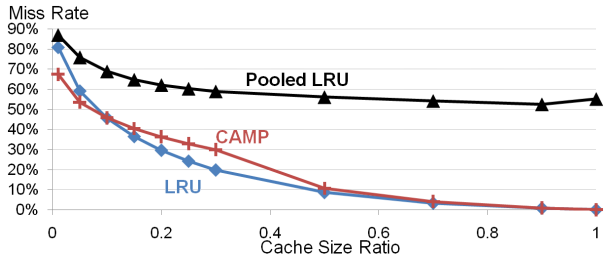
(a) Cost-miss ratio as a function of precision.



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



(c) Cost-miss ratio as a function of the cache size ratio, with the precision of CAMP set to 5.



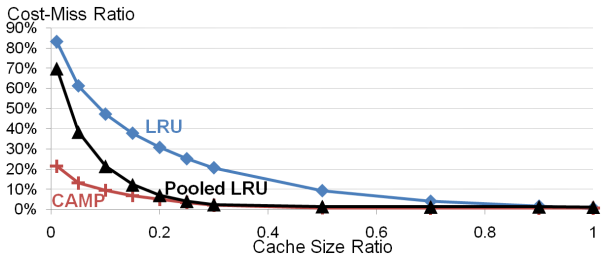
(d) Miss rate as a function of the cache size ratio, with the precision of CAMP set to 5.

Figure 2.5: Simulation results with one trace and cost values selected from 1, 100, 10K.

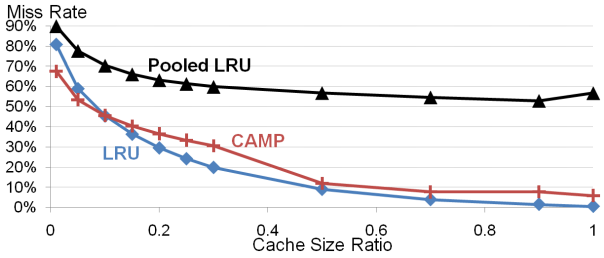
Different groups are assigned to different pools so that cheap and expensive key-value pairs do not compete with one another for the same memory. This is not the same as CAMP, which adjusts the amount of memory used by each queue automatically, as demand fluctuates.

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

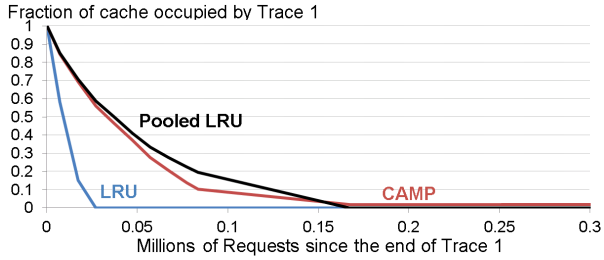
With the BG benchmark-generated trace using synthetic cost values selected from {1, 100, 10K}, Pooled LRU constructs three pools. These pools have approximately the same number of key-value pairs, frequency and size, where the frequency of a pool is the number of references made to key-value pairs in that pool and the size is the amount of memory needed to store all key-value pairs in that pool. With a uniform partitioning of memory, Pooled LRU has both a cost-miss ratio and miss rate similar to LRU. This is a consequence of the key-value pairs assigned to each pool having the same frequency and size. The performance is so close, that we only display the cost-miss ratio for LRU in Figure 2.5c. When memory is partitioned using cost, Pooled LRU improves over LRU's cost-miss ratio. Furthermore, it is able to match CAMP's cost-miss ratio when given a large enough cache size by assigning practically all of it to the most expensive pool. However, this improvement is at the expense of a significantly worse miss rate (see Figure 2.5d), since, even with a large cache size, the cheapest pool has nearly a 100% miss rate and the second pool has a miss rate of 65%.



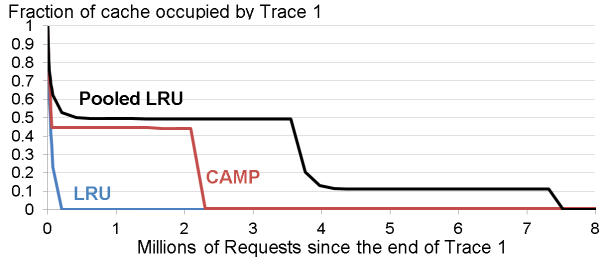
(a) Cost-miss ratio as a function of cache size ratio.



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



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



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

Figure 2.6: Simulation results with changing access patterns.

2.3.1 Evolving access patterns

A key feature of CAMP is its ability to adapt to evolving access patterns by evicting those key-value pairs that were hot in some distant past. This includes expensive key-value pairs. Obtained results show CAMP adapts effectively when compared to LRU and Pooled LRU. Moreover, the overall cost-miss ratio and miss rate trends remain the same as the results of Figure 2.5.

In this experiment, we used ten different traces back to back. Each trace file consists of 4 million key-value references. Moreover, requests from different traces are given distinct identification, so any request from a given trace file will never be requested again after that trace. They are adversarial to CAMP because each trace file is generated using a skewed distribution of access as described at the beginning of this section. This means once the simulator switches from the first trace file to the second one, none of the objects referenced by trace file 1 are referenced again. The same holds true for all other trace files, emulating a sudden shift in access patterns where expensive objects that were referenced frequently are never referenced again. We conducted the above experiment for a variety of cache sizes and observe CAMP adapts across the different trace files to provide a cost-miss ratio and miss rate similar to those observed in the previous section, see Figure 2.6a and 2.6b. Figures 2.6c and 2.6d show the fraction of key-value store memory occupied by the key-values of trace file 1 for two different key-value store memory size ratios, 0.25 and 0.75. These two figures show how well the different techniques adapt to the sudden change in access patterns. The x-axis of these two figures it the number of key-value references issued relative to the start of trace file 2 in millions of requests. The transition to a different trace file is at the 4 million tick mark of the x-axis. The y-axis shows the fraction of key-value store memory size occupied by key-value pairs of trace file 1.

With a small cache size (see Figure 2.6c with a cache size ratio of 0.25), all three algorithms

evict key-value pairs referenced by trace file 1 quickly. LRU is the quickest, evicting all key-value pairs of trace file 1 after 21,000 references of trace file 2. It is followed by Pooled LRU with 131,000 references of trace file 2. CAMP evicts most of trace file 1 key-value pairs quicker than Pooled LRU and slower than LRU. It does not evict all trace file 1 key-value pairs (those with the highest cost-to-size ratio) until 7.7 million references, close to the end of trace file 3. However, these items occupy less than 2% of the total cache size.

With a larger cache size (see Figure 2.6d with a cache size ratio of 0.75), both CAMP and Pooled LRU behave in a step function with CAMP evicting a majority of trace file 1 key-value pairs faster than Pooled LRU. Once again, LRU is quickest as it considers recency of references. Pooled LRU evicts all key-value pairs referenced by trace file 1 after 7.3 million requests are issued, close to the end of trace file 3. CAMP maintains a few of the most expensive key-value pairs of trace file 1 even after 40 million requests are issued. However, these occupy less than 0.6% of the available key-value store memory.

Let us analyze the behavior of each algorithm. LRU evicts the key-value pairs requested in trace file 1 when the total size of newer key-value pairs is greater than the cache size, which occurs before the transition to trace file 3 regardless of cache size. The jump in eviction time at cache size ratio 1 corresponds to the fact that the key-value pair that causes the total size of requested key-value pairs to exceed the cache size is the first key-value pair requested in trace file 3.

The sudden eviction of large portions of trace file 1 key-value pairs by Pooled LRU at large cache sizes (see Figure 2.6c) correspond to the introduction of new key-value pairs at the beginning of trace file 3 and trace file 4, which occur at around the 4 millionth and 8 millionth mark. As in the first experiment, Pooled LRU pools key-value pairs by cost, and for a fixed cache size, each pool is allotted a portion of the cache proportional to the cost value. Since the only occurring cost values are 1, 100 and 10K, 99% of the cache is dedicated to the pool of expensive key-value pairs. On the other hand, the expensive key-value pairs from a single

trace file only occupy a third of the maximum cache size, so that a cache size ratio of 2/3 can store all expensive key-values from two trace files, and a cache size ratio of 1 can store those of 3 separate trace files. Now the point at which all trace file 1 key-value pairs are evicted occurs when the total size of the expensive key-value pairs requested in a subsequent trace file exceeds the cache size. When the cache size ratio is 1/3 or less, this occurs before the end of trace file 2 (4 million requests). At a cache size ratio of 2/3 or higher, the eviction time occurs during trace file 4 (roughly between 8 and 12 million requests).

Finally, CAMP maintains an LRU queue for each cost-to-size ratio, and unlike Pooled LRU, these queues can be resized dynamically. Because key-value pairs requested at a later time can have higher priority of being evicted than those requested earlier, CAMP only guarantees that those requested after trace file 1 that have the highest cost-to-size ratio will have lower priority than any trace file 1 request. This observation yields the loose guarantee that all trace file 1 key-value pairs will be evicted by the time the total size of all newer requested items with the highest cost-to-size ratio reaches the cache size. According to Figure 2.6d, for a cache size ratio of 0.75, there are cache-resident still key-value pairs from trace file 1 at the end of the simulation. This is explained by the fact that the highest cost-to-size key-values contribute less than 1/20th of the maximum cache size per trace file. Together over the whole trace, these key-value pairs will fit in caches with cache size ratios greater than 0.5. Hence, the condition guaranteeing the eviction of all items in trace file 1 is satisfied with the .25 cache size of Figure 2.6c and not satisfied with the .75 cache size of Figure 2.6d.

2.3.2 Other traces

The trends reported in Subsection 2.3.1 also hold with other traces. The most insightful results are obtained with the two possible extremes, namely, key-value pairs with variable size but similar costs and key-value pairs with equal size but variable costs. We describe

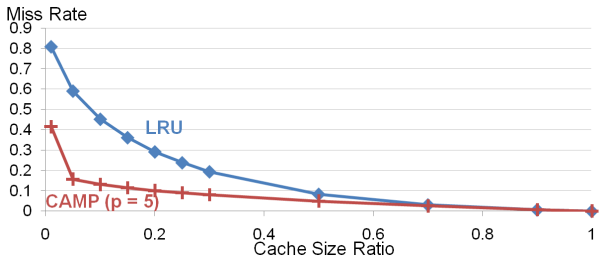
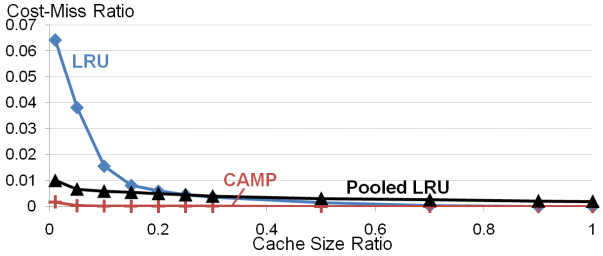


Figure 2.7: Miss rate as a function of cache size with variable size key-value pairs and constant cost.

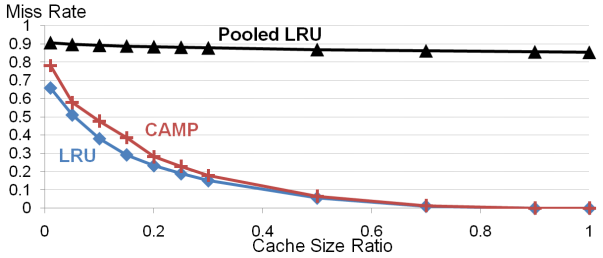
these in turn.

With variable size key-value pairs whose cost is identical, CAMP renders small key-value pairs resident, providing a lower miss rate when compared with LRU, see Figure 2.7. Pooled LRU constructs one pool and behaves the same as LRU. Because the cost of the key-value pairs is set to 1, the cost-miss ratio of a technique is equal to its miss rate. Hence Figure 2.7 can also be interpreted as the cost-miss ratio of LRU and CAMP as a function of cache size ratio.

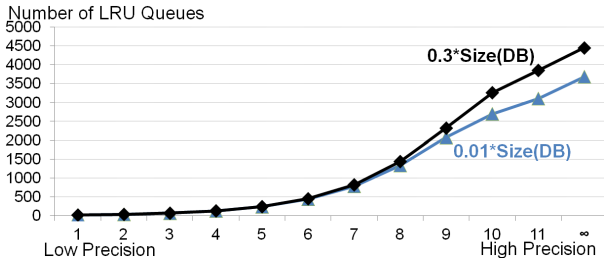
With equal size key-value pairs that incur a different cost, CAMP continues to provide a superior cost-miss ratio to both LRU and Pooled LRU, see Figure 2.8a. With a limited amount of memory, CAMP's miss rate is slightly worse than that of LRU as it favors high cost key-value pairs, see Figure 2.8b. With Pooled LRU, we were challenged in determining the size of different pools as there was no clear way to partition the range of different costs. In the original trace where key-value pairs could only have one of three different cost values, items were pooled by their cost value. Here, we opted to pool items by range of cost values. Specifically, the ranges were 1 to 100, 100 to 10,000 and 10,000 and beyond. The available memory was then divided among the three pools in such a way that each pool received an amount proportional to the lowest cost value in its range. With this assignment, Pooled LRU results in a superior cost-miss ratio with small cache-size ratios. With larger cache size ratios, its partitioning of space makes it inferior to both LRU and CAMP.



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



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



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

Figure 2.8: Simulation results with key-value pairs of equal size and variable costs.

In comparison to the trace with three distinct cost values (Figure 2.5b), the trace with equally sized key-value pairs has key-value pairs with many more distinct cost values. Therefore, for this trace, CAMP creates a larger number of LRU queues when no rounding takes place, see Figure 2.8c. This increase is due to the fact that there is a greater number of cost-to-size ratios as a result of the considerably larger set of cost values. With additional rounding, i.e. at lower precision, the number of LRU queues in the two traces decrease significantly and converge without any sizable performance degradation.

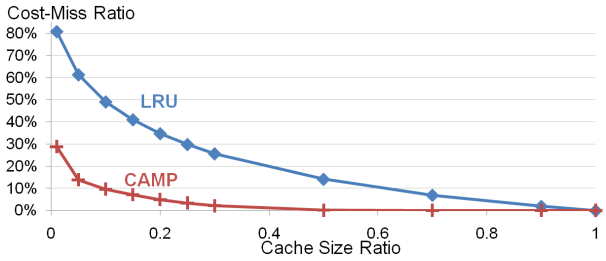
2.4 An implementation

We implemented CAMP in the IQ Twemcache, a modified version of the Twemcache v2.5.3 [64] that implements the IQ framework [33]. This implementation computes the cost of a key-value pair by noting the timestamp of a miss observed by a get (iqget) and the subsequent insertion of the computed value using a set (iqset). The difference between these two timestamps is used as the cost of the key-value pair.

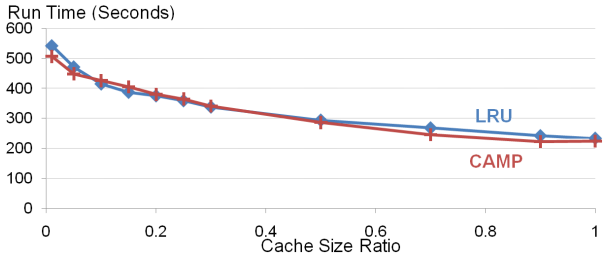
The approach taken to provide recomputation time is to use the service time to compute a key-value pair (and piggybacked as a part of the input to the key-value store). Hints provided by the application are another possibility.

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

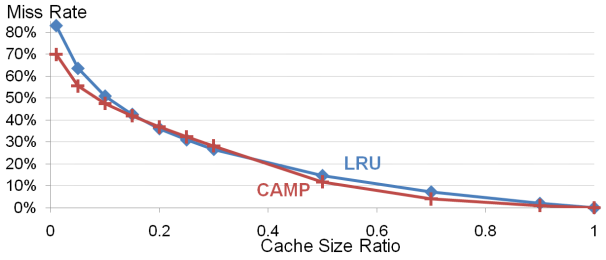
We developed an application that implements the request generator of Section 2.3 by reading a trace file and issuing requests to the key-value store using Whalin client version 2.6.1 [71]. We used the trace file with synthetic costs of {1, 100, 10K} per discussions of Section 4.4. Figure 2.9a shows the observed cost-miss ratio with LRU and CAMP as a function of different



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



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



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

Figure 2.9: Implementation results, where the precision of CAMP is set to 5.

cache size ratios. CAMP incurs a significantly lower cost for the missing keys with smaller cache sizes. This difference becomes smaller with larger cache sizes because the key-value store miss rate drops. These results are consistent with those reported in Section 2.3.

Figure 2.9b shows the amount of time required to run the trace with both LRU and CAMP. It includes the following: (1) the time for either LRU or CAMP to process a cache hit and to make replacement decisions when the memory is exhausted; (2) the time to transmit a key-value pair across the network (with both a cache hit and a cache insert); and (3) the time to copy a key-value pair across the different software layers. The results show that CAMP provides response times comparable to those of LRU. If the cost was included in the reported response times, CAMP would have been significantly faster than LRU, resembling the results reported in Section 2.3. Here, we wanted to show results that demonstrate that an implementation of CAMP is as fast as LRU while ignoring the cost associated with keys.

With both CAMP and LRU, the run time decreases as a function of cache size. The explanation for this is as follows. A get(k) that observes a miss is followed by a set(k,v). With a small cache size that is full, the set operation must evict one or more existing key-value pairs and write the new key-value pair (k, v) in the cache. This write operation requires copying of the key-value pair (k, v) from the network buffer into the memory of the cache manager. A larger cache size reduces the number of such operations because a higher percentage of get(k) operations observe a cache hit, see Figure 2.9c. This explains why the run time improves with both LRU and CAMP using a larger cache size. This also explains why CAMP provides a faster response time than LRU for those cache sizes that provide a lower miss rate, i.e., cache ratio of 0.01.

2.4.1 Discussion

CAMP is suitable for use with multi-core processors as it minimizes the delay incurred with concurrent threads racing with one another to read and write CAMP's data structures. Several features of CAMP enable it scale vertically. First, it only updates its heap data structure (which requires synchronized access) when the head of a LRU queue changes value instead of per eviction. Second, different threads may update different LRU queues simultaneously without waiting for one another. Finally, CAMP may represent each LRU queue as multiple physical queues and hash partition keys across these physical queues to further enhance concurrent access.

CAMP can be extended with use with a hierarchical cache (using SSD, disk, or both) which may persist costly data items. With any finite cache size, should the data set size exceed the cache size, an algorithm must make an eviction decision. CAMP systematically renders such decisions by considering size and cost of key-value pairs, and recency of references with a two level cache.

2.5 Related work

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

LRU-K [61], 2Q [40], and ARC [56] are adaptive replacement techniques that balance between the recency and the frequency features of a workload continuously, improving cache hit rate for

fixed size disk pages with a constant cost. CAMP is different in that it considers both the size of a key-value pair and its cost. CAMP is an efficient implementation of the GreedyDual-Size algorithm [11], visiting significantly fewer heap nodes than GreedyDual-Size, see the discussion of Figure 2.4 in Section 2.2.

Like CAMP, GD-Wheel [50] strives to enhance the efficiency of GreedyDual-Size. There are, however, some significant differences in approach. GD-Wheel rounds the overall priority for each key-value pair instead of the cost-to-size ratio which makes it difficult to evaluate the cost of approximation. The GD-Wheel study does not give a direct comparison between GD-Wheel and GreedyDual-Size. With CAMP we are able to give well-defined guarantees on the competitive ratio of CAMP relative to GreedyDual-Size. Moreover, GD-Wheel does not address how to select their precision parameter N or give an empirical characterization of performance as a function of precision. Finally, GD-Wheel must implement occasional migration procedures wherein all the key-value stores within a GD-Wheel are migrated to the next level. CAMP does not require such a migration step as it uses the cost-to-size ratio as the basis of the rounding scheme (which does not change while a key-value pair is in the cache).

While deciding how space should be assigned (answer to the first question) is a different topic, there are implementations that strive to answer this question in combination with a replacement technique. For example, the memory used by a Twemcache server instance is managed internally using a slab allocation system [7] in combination with LRU. Below, we describe this technique and how it is different than CAMP.

Twemcache divides memory into fixed size slabs (chunks of memory), the default size being 1 Megabyte. Each slab is then assigned a slab class and further sub-divided into smaller chunks based on its slab class. For example, a slab class of 1, the smallest size, would have a chunk size of 120 bytes. This means that a single slab of class 1 can fit 8737 (1 MB / 120 byte) chunks. Every subsequent higher slab class uses chunk sizes that are approximately a factor

of 1.25 larger. So, a slab class of 2 accommodates 6898 chunks each 152 bytes in size. This slab class stores key-value pairs whose size is between 120 and 152 bytes. The largest slab class uses a chunk size that accommodates the entire slab.

When storing a key-value pair, the server determines the amount of memory required to store the key-value pair along with a header for meta-data. The memory allocation attempts the following steps, proceeding in order until it is able to satisfy the allocation request:

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

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

Twemcache attempts to resolve this calcification limitation by randomly evicting a slab from another class if it is unable to allocate an item. This approach may evict potentially hotly accessed items, reducing the cache hit rate. Additionally, the random slab eviction does not deal with the case when a disproportionately small number of slabs are assigned to the

needed slab class. To illustrate, assume only one slab was assigned to the slab class 5 (0.1% of total memory) and the workload changes such that key-value pairs corresponding to slab class 5 are referenced much more frequently. All the requests must compete for chunks in the single slab whereas the remaining cache space is now under-utilized. Since key-value pairs can still be allocated, the random slab eviction does not activate to free up more slabs.

One may address the calcification limitation by separating how memory should be allocated for the key-value pairs from the online algorithm that decides which key-value pairs should occupy the available memory. For example, with a memcached implementation, one may use a buddy algorithm [28] to manage space in combination with CAMP (or LRU). With those caches that run in the address space of an application (e.g., JBoss, EhCache, KOSAR), the memory manager of the operating system may manage space for instances of classes that serve as values for application specified keys. With these, one may use CAMP to decide which key-value pairs occupy the available memory.

2.6 Conclusion

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

Chapter 3

Cache replacement with memory allocation*

3.1 Introduction

Early designers of computer systems noticed that if a computer program generates a request to access a page in memory, that page is more likely than other pages to be requested again in the near future. This phenomenon, known as locality of reference, is the motivation behind organizing data into two tiers of memory. Pages stored in main memory (the cache) are accessed more quickly. However, main memory is costly and limited in capacity. Therefore most of the pages are stored on disk which has a retrieval time that is orders of magnitude slower than main memory. The paging problem is to design an effective page replacement policy: which page in main memory should be evicted in order to make room for a new incoming page? The goal is to minimize the number of cache misses, that is, requests to a page which is not in main memory.

*This chapter is included with permission from SIAM[37].

In the context of computer systems, the cost of a cache miss is the same for all pages. Furthermore, the size of each page is uniform. In the *generalized caching problem*, the items occupying memory are heterogeneous in that each item p has a cost and a size. A cache replacement policy must always maintain the invariant that the sum of the sizes of the items in the cache does not exceed the total size of the cache. The cost of a cache miss is the cost of the item requested. The goal is to minimize the total cost of the cache misses. One of the original motivations for studying generalized caching was to design effective replacement policies for web caches. Data objects on the web are naturally heterogeneous in size, depending largely on their type (e.g., text, video, image, etc.). The cost to retrieve an item not in the cache is the cost to retrieve it from some other location on the internet which can depend on a variety of factors such as the location of the closest copy and network traffic conditions.

Recently, the generalized caching problem has become important in the context of modern database systems. Many queries to large commercial databases with high read-to-write ratio (such as those maintained by Facebook and Twitter) are repeated over time. Query results (called *key-value pairs*) can be stored in memory so that the next time the query is issued, the result can be retrieved from memory instead of recomputed from scratch. The problem of managing a cache of key-value pairs (called a *key-value store*) maps closely onto the generalized caching problem. Key-value pairs vary in size as they contain different types of data. The cost to bring a key-value pair into the key-value store corresponds to the time to compute it from the database and can vary greatly from query to query.

Memcached, currently the most popular key-value store manager, is used by companies such as Facebook, Twitter and Wikipedia. Memcached is given a fixed amount of memory for the key-value store and manages all replacement decisions for that size cache. In addition, memcached manages its own memory layout rather than relying on the underlying operating system's calls to allocate and deallocate memory. Instead of breaking key-value pairs into fixed-size

blocks, key-value pairs are stored contiguously in memory. Thus, the cache replacement policy must be coordinated with a memory allocation policy. This contrasts with the traditional generalized caching problem which only requires that the size of the items not exceed the size of the cache and otherwise disregards how the items are laid out in memory.

Memcached uses a memory management scheme known as the slab allocator [7] in which key-value pairs are assigned to one of a small number of LRU queues based roughly on their size. Each queue is given a number of memory slabs determined by the initial portion of the request sequence. The original version of memcached does not feature dynamic resizing of queues as request patterns change, resulting in a phenomenon known as calcification in which some key-value pairs remain in the key-value store indefinitely. Later variants have added some means of reassigning slabs between LRU queues, including selecting the slab containing the oldest item in the cache [60], or selecting at random slab [64]. So far these approaches to managing memory layout in conjunction with a replacement policy have been ad hoc.

We introduce the *managed memory caching problem* in order to provide a more systematic approach. A problem instance consists of the size of the cache and a sequence of named items together with their size and cost. (Although the problem is motivated by maintaining key-value stores for databases, we use the more generic terms *item* for key-value pair and *cache* for key-value store.) For every item, we must decide where to place it in the cache along with which items to evict in order to make room for the incoming item. Placement is only valid if the entire item can fit into a single contiguous segment of free memory. The goal is to minimize the total cost of requests resulting in a cache miss.

We present an algorithm called CAMP-MALLOC for the managed memory caching problem that is competitive against the optimal algorithm for the generalized caching problem, when it is augmented with enough memory to compensate for fragmentation and give bounds on the fragmentation. CAMP-MALLOC does not require that a key-value pair be moved in memory once it is placed. However, one can imagine a variant of the problem in which

objects can be moved at a cost in order to make longer contiguous vacancies in memory.

We also present simulation results using a social networking benchmark. The fragmentation incurred by our algorithm is typically less than 15% except for very small cache sizes. Caching performance is measured by the *cost-miss ratio* which is the sum of the costs of all requests resulting in cache misses divided by the sum of the costs of all requests in the sequence. While CAMP-MALLOC does suffer from some degradation in performance due to managing memory layout, for most cache sizes, its performance is comparable to competitive page replacement policies that are not concerned with memory layout.

CAMP, described in Chapter 2, provides a good starting point for addressing memory allocation because the items are already organized into a small set of queues. While CAMP uses LRU to prioritize items within a queue, we use FIFO because it requires less movement of objects. Although FIFO has the same competitive ratio as LRU, it generally does not perform as well in practice. The cost of the switch from LRU to FIFO is isolated and measured in the simulations. Finally we devise a way of moving memory between the queues in larger blocks of memory in order to adapt to shifting patterns in the request sequence. There is some fragmentation incurred from laying out objects within a block (which we call *block fragmentation*) and some fragmentation in reassigning blocks from one queue to another (which we call *queue fragmentation*). Both of these phenomena are measured in the simulations.

This chapter is organized as follows. Section 3.2 gives a description of our algorithm for cache replacement with memory allocation. Section 3.3 contains a proof of the competitiveness as well as a theoretical bound on fragmentation of $2\sqrt{2qks_{\max}}$ where q is the number of queues, k the size of the cache and $s_{\max}$ the maximum item size. Finally, in Section 3.4, we report on the results of a trace-driven simulation that compares the performance of our algorithm and several other caching algorithms.

3.2 Algorithm CAMP-malloc

Algorithm 2 CAMP-MALLOC.

```
function INITIALIZATION
    all queues are empty
    all blocks are free
    L ← 0
function SERVE(item i)
    if i is in the cache then
        return
    if queue(i) is empty or i does not fit in queue(i)'s tail block then
        if there is no free block then
            L ← min_{blocks B} H(B)
            B ← argmin_{blocks B} H(B)
            evict all items in B
            remove B from its queue
        append a free block to queue(i)
    B ← block at queue(i)'s tail
    append i to B
    H(B) ← L + cost(i)/size(i)
```

In order to give some intuition for algorithm CAMP-MALLOC (Algorithm 2), we first consider the situation in which all items have the same cost-to-size ratio. In this case, both FIFO and LRU will be k-competitive. However it is much easier to manage memory layout with FIFO by configuring the address space as a circular queue as shown in Figure 3.1.

A request to an item already in the cache does not change the memory layout. There is a gap between the head and tail of the FIFO queue which is always less than the size of the largest object. On a cache miss, objects are evicted from the head of the queue in FIFO order until the gap is large enough to accommodate the incoming item. Then the item is placed at the tail of the queue adjacent to the item that was placed in the cache most recently. There is also a second gap where the queue circles around from the end to the beginning of the memory allocated to the cache.

Note that the physical layout of items in memory always coincides with the FIFO order of

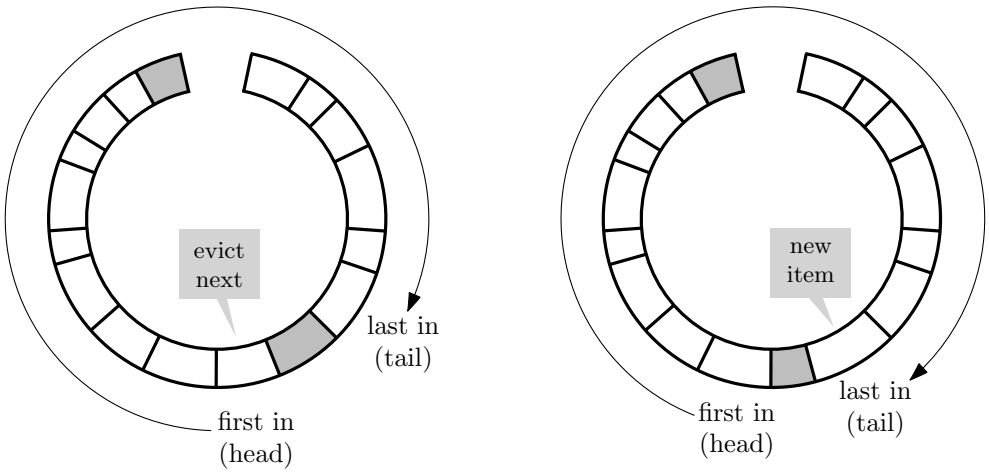


Figure 3.1: A FIFO queue with memory layout. The memory for the cache is organized as a circular array. Each segment is a data object. Gray portions are unused memory. The two diagrams show the state of the queue before and after an item is evicted and a new one is inserted.

the items. By contrast, with LRU, the logical order of the items and their physical layout in memory are different which will result in holes in the layout when items are evicted. Thus, a memory allocation policy used in conjunction with LRU would have to provide a means of filling holes or moving items within the cache.

Since FIFO is more amenable to memory layout than LRU, the first change we make to CAMP is to implement each separate queue as a FIFO queue instead of an LRU queue. In terms of the original GreedyDual-Size algorithm, this amounts to not updating the priority of the requested item on cache hits.

In order to dynamically allocate memory between the queues, we divide memory into same-size blocks. Blocks are large enough to hold many items . We optimize the size of the blocks later. Starting from an empty block, items are placed in the block next to each other in the order in which they entered the cache. Each FIFO queue owns a set of blocks. While the blocks in each queue are logically ordered in FIFO order, they are physically scattered around the memory. Thus, a block can be emptied and reassigned to another queue as needed. If there is no more room in the block at the tail of a queue to insert the item currently being served,

we choose a block to flush and add it to the tail of the queue.

Items are assigned to a queue as a function of their cost-to-size ratio $r = \text{cost(item)}/\,\text{size(item)}$. Let queue(item) and queue(r) both denote this assignment. It is only necessary to track a priority for each block, although the analysis is simplified by keeping the priority for each item as it would have been assigned with CAMP. When an item is inserted into the cache, it is assigned a priority of $H(\text{item}) = L + \text{cost(item)}/\,\text{size(item)}$. Since the priority of a block is updated to $H(\text{item})$ at the time an item is inserted, the priority of a block is always equal to the priority of its most recently inserted (highest priority) item. The priorities of the blocks also coincide with their order in the FIFO queue with the lowest priority block at the head of the queue and the highest priority at the tail of the queue.

3.3 Proof of competitiveness

By induction on the number of requests, it can be observed that the value of L is a lower bound on the priority of any block in the cache and that L is non-decreasing in time. Define an item in cache to be *eligible* if its priority is at least as large as L. Otherwise, we say that the item is *ineligible*. An item is eligible when it is first brought into cache. If an item becomes ineligible at any point, it remains so until its eviction.

Lemma 3.1. The total size of eligible items occupying CAMP-MALLOC's cache is at least

$$k - 2qs_{\text{block}} - ks_{\max}/s_{\text{block}},$$

where k is the cache size, $s_{\max}$ the maximum item size, s_{block} the block size and q is the maximum number of cost-to-size values present in CAMP-MALLOC's cache at any one time.

Proof. We call a block an *intermediate* block if it is not the head or tail block in its queue.

Each of the items in an intermediate block was brought into the cache after all the items in the head block for its queue. Therefore, the priority of each item in an intermediate block is at least as large as the priority of the head block in its queue which is in turn at least as large as L. Therefore, all items in intermediate blocks are eligible.

Each intermediate block was the tail block at the time it was being filled. Since intermediate blocks are by definition no longer the tail block, there was a point at which CAMP-MALLOC tried to allocate an item to the block but failed because there was insufficient room. Therefore, the sum of the sizes of the items in each intermediate block is at least $s_{\text{block}} - s_{\max}$. The amount of memory in intermediate blocks is at least $k - 2qs_{\text{block}}$. There are at most k/s_{block} intermediate blocks, each of which has at most $s_{\max}$ wasted space. Therefore, the total size of items in intermediate blocks (a lower bound on the total size of eligible items in the cache) is at least $k - 2qs_{\text{block}} - ks_{\max}/s_{\text{block}}$. □

The $2qs_{\text{block}}$ term in Lemma 3.1 is a bound on queue fragmentation because 2 blocks per queue (at the head and the tail) could be almost empty or contain ineligible items. The $ks_{\max}/s_{\text{block}}$ term is a bound on the block fragmentations because there is at most $s_{\max}$ wasted space in each full block. Therefore, the fragmentation is minimized when

$$s_{\text{block}} = \sqrt{\frac{ks_{\max}}{2q}},$$

which results in a fragmentation amount equal to

$$2\sqrt{2qks_{\max}}.$$

The following theorem shows that, given enough additional memory to accomodate for queue and block fragmentation, CAMP-MALLOC is $k/s_{\min}$ competitive against OPT. Note that $k/s_{\min}$ corresponds to CAMP's competitive ratio.

Theorem 3.2. Suppose that CAMP-MALLOC has cache size k and OPT has cache size h such that

$$h \leq k - 2qs_{\text{block}} - \frac{k}{s_{\text{block}}} s_{\max} \tag{3.1}$$

where q, s_{block} and $s_{\max}$ are defined as in Lemma 3.1. Then for any request sequence

$$\text{cost}(\text{CAMP-MALLOC}) \leq \frac{k}{s_{\min}} \text{cost}(\text{OPT}),$$

where $s_{\min}$ is the minimum item size.

Proof. Let L_f denote the value of L after CAMP-MALLOC has served the entire request sequence. If suffices to prove that

$$\text{cost}(\text{CAMP-MALLOC}) \leq kL_f \tag{3.2}$$

and that

$$L_f \leq \text{cost}(\text{OPT})/s_{\min}. \tag{3.3}$$

To prove (3.2), consider a single unit of memory in CAMP-MALLOC's cache and an item i that occupies it. During the time i is eligible during this occupation, the amount by which L increases is at least

$$\text{cost}(i)/\,\text{size}(i) \leq \Delta L.$$

This is because i becomes ineligible at the moment that L increases passed $H(i)$, which is an amount $\text{cost}(i)/\,\text{size}(i)$ more than the value of L when it entered. Summing this inequality over all items that occupy that unit of cache, and noting the fact that no two items can

occupy the same unit at the same time, we obtain a lower bound on L_f. Now summing over all cache units results in (3.2).

To prove (3.3), think of the execution of each request happening sequentially so that OPT serves the request first and then CAMP-MALLOC serves the request. Define a period of an item to be the duration of time that it is eligible and in CAMP-MALLOC's cache, but not in OPT's cache.

Every increase in L belongs to a period. This is because an L increase corresponds to an item i being a miss for CAMP-MALLOC. Since OPT has already served i, i is in its cache, but it is not in CAMP-MALLOC's cache. But by lemma 3.1 and assumption (3.1), the total size of items in OPT's cache is less than the total size of eligible items in CAMP-MALLOC's cache. Hence, there is an eligible item in CAMP-MALLOC's cache that is not in OPT's. So the L increase belongs to this item's period.

Next, we show that the total amount of increase in L during a period for an item i is

$$\Delta L \leq \text{cost}(i)/s_{\min}. \tag{3.4}$$

Let L_0 be the value of L the last time i enters the cache before this period begins. Let L_1 and L_2 denote the values of L when the period begins and when it ends. Since L is non-decreasing, $L_1 \geq L_0$. The period ends when i becomes ineligible or is evicted, which happens when L increases passed $L_0 + \text{cost}(i)/\text{size}(i)$, or the item is requested again before it is evicted by CAMP-MALLOC, in which case L has not yet grown passed $L_0 + \text{cost}(i)/\text{size}(i)$. In either case, we have

$$L_2 - L_1 \leq \text{cost}(i)/\text{size}(i) \leq \text{cost}(i)/s_{\min}.$$

Finally, we prove (3.3) by summing (3.4) over all items evicted by OPT and removing all

duplicate increases in L from the left hand side of the inequality. □

3.4 Simulations

We used a social networking benchmark named BG [5, 6, 26, 27] to generate traces of key-value references from a cache augmented database management system [32]. BG emulates members of a social networking site viewing one another's profile, listing their friends, and performing other interactive actions. The benchmark is configured to reference keys using an i.i.d. skewed pattern of access with approximately 70% of requests referencing 20% of keys. The trace file consists of approximately 4 million rows. Each row identifies a referenced key-value pair, its size, and cost. Cost is the time required to compute the key-value pair by issuing queries to the RDBMS. Although the cost per key-value pair varies significantly, the distribution of sizes of the objects in the original trace was highly bimodal with sizes clustered around two distinct values (1200 and 83000 bytes) depending on whether the key-value pair contains an image. Such uniformity in sizes gives our algorithm an unfair advantage since it results in very little fragmentation within a block. We therefore randomized the sizes by taking the original size of each object and multiplying by a factor generated by a Gaussian distribution with mean 1 and standard deviation 0.6.

We implemented a simulator that consists of a key-value store and a request generator to read the trace file and issue requests to the key-value store. The key-value store manages a fixed-size memory with a cache replacement algorithm. We implement three distinct key-value store management algorithms: CAMP (the same algorithm studied in Chapter 2), CAMP-FIFO, and CAMP-MALLOC. CAMP-FIFO is the same as CAMP except that the replacement policy for each queue is FIFO instead of LRU. CAMP and CAMP-FIFO only need to maintain that the sum of the sizes of items in the cache does not exceed the size of the cache. CAMP-MALLOC, on the other hand, divides memory into fixed-size blocks

and maintains that the sum of the sizes of the items within each block does not exceed the capacity of the block.

CAMP-FIFO is an intermediate algorithm which we introduce in order to isolate the degradation in performance due to using FIFO instead of LRU and then the degradation in performance due to fragmentation from implementing the memory layout in CAMP-MALLOC. The primary means of evaluating cache performance is the *cost-miss ratio* which is the sum of the costs of the requests resulting in a cache miss divided by the sum of the costs of all the requests. Because the cache starts out empty, the cost-miss ratio only takes into account requests to items that have been requested at least once in the past.

The primary focus of Chapter 2 is the evaluation of CAMP relative to GREEDYDUAL-SIZE, specifically whether caching performance is compromised with lower precision for priorities and the resulting reduction in implementation cost with fewer queues. The data shows that for the benchmarks used in the simulations, the cost-miss ratio of CAMP is comparable to GREEDYDUAL-SIZE even when only a single bit of precision is used for the cost-to-size ratio (effectively rounding the cost-to-size ratio down to the nearest factor of 2). Therefore, we use that same level of rounding in the simulations for this study. For that level of precision and the benchmarks used here, the maximum number of queues possible is 25. In reality only an average of 13 queues are populated at any given point in time in the course of the simulation. The number of queues varies only slightly with the cache size and algorithm.

Figure 3.2 shows the cost-miss ratio of CAMP, CAMP-FIFO, and CAMP-MALLOC for caches of different sizes. The size of each cache is denoted as a percentage of the total sizes of all the key-value pairs requested in the trace. Note that CAMP-MALLOC has an open parameter which is the block size. For this set of data, we post-selected the block size that resulted in the optimal cost-miss ratio in order to get a lower bound on performance loss due to fragmentation. However, the next set of data shows that performance is not particularly sensnitive to choice of block size. The biggest performance degradation occurs

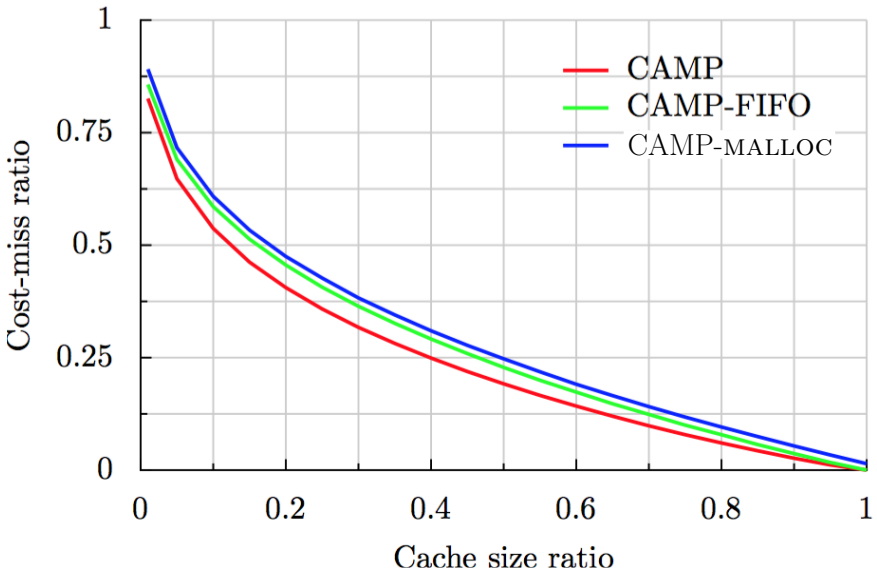
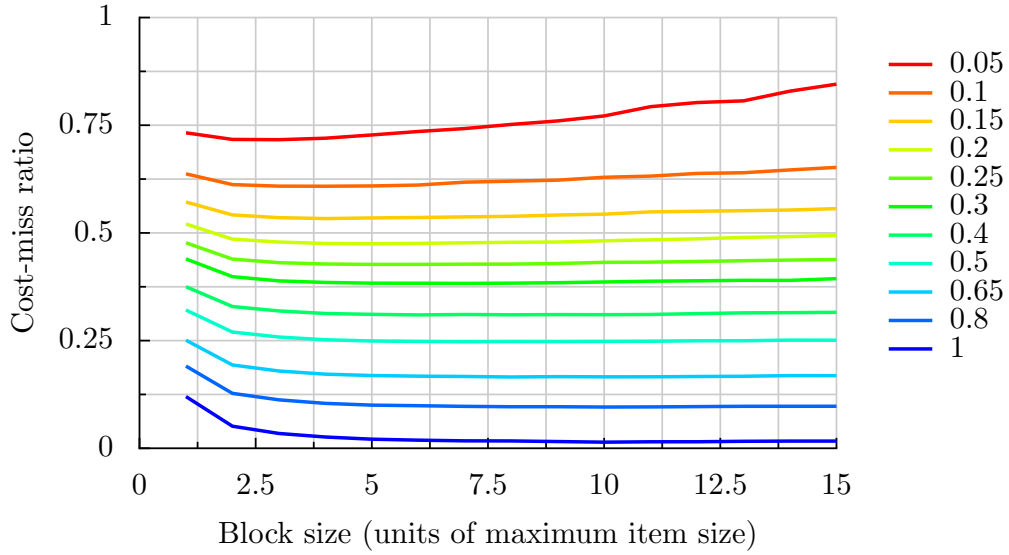


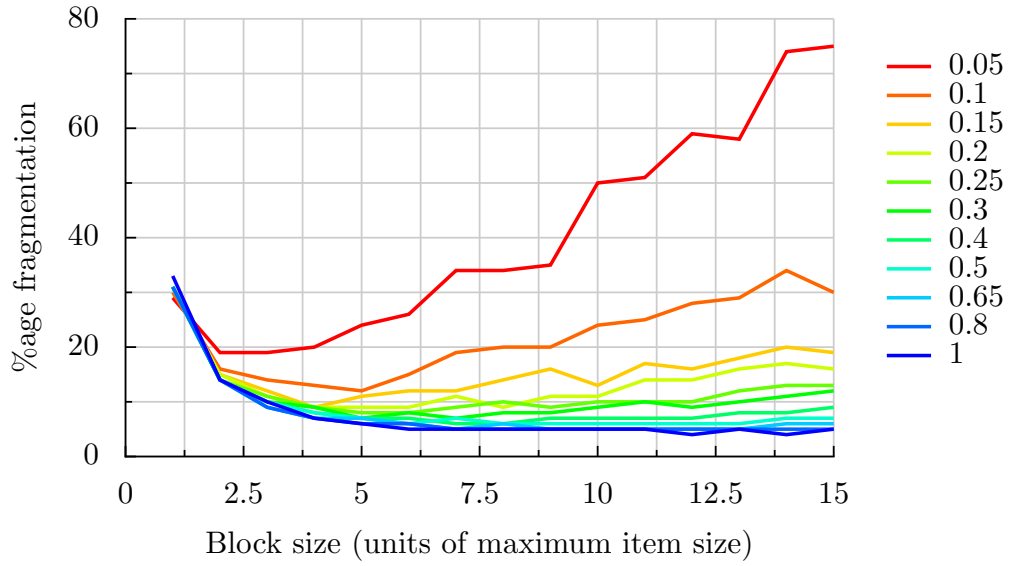
Figure 3.2: Performance of three caching algorithms

between CAMP and CAMP-FIFO. On the other hand, performance degradation due to implementing a memory layout in CAMP-MALLOC is actually minimal.

Figures 3.3a and 3.3b show how the cost-miss ratio and amount of fragmentation of CAMP-MALLOC depend on the block size. The block size is expressed as a multiple of the largest item size as a convenient way to convet the size of the block relative to the items it holds. In practice, the block size would be a fixed number of kilobytes and need not be an integer multiple times the size of the largest item. Fragmentation is measured at the end of the simulation and is defined as the fraction of cache that is unoccupied. There are two competing factors that contribute to fragmentation which need to be balanced in selecting block size. The wasted space within a block (block fragmentation) is roughly the same for each block but is multiplied times the number of blocks. Therefore smaller blocks will result in more block fragmentation. The other contributing factor to fragmentation is the queue fragmentation and comes from the fact that the queue block of each queue is only partially filled. Larger blocks will result in more queue fragmentation. Note that the competitive analysis accounts for fragmentation in the head block as well as the tail block in bounding queue fragmentation



(a) Cost-miss ratio as a function of block size



(b) Fragmentation as a function of block size

Figure 3.3: Performance of CAMP-MALLOC as a function of block size. Each curve corresponds to a different cache size ratio.

because items in the head block can become ineligible and are therefore not the most desirable items to keep in the cache. The empirical results do not count space occupied by ineligible items as wasted space since they could still be resulting in cache hits. Thus, the amount of queue fragmentation accounted for in the analysis is pessimistic in comparison to the simulations.

The data in Figures 3.3a and 3.3b suggest that the best block size is a small constant times the size of the largest item. However, performance does not depend significantly on block size except for very small caches. Fragmentation becomes problematic when the number of blocks per queue becomes too small which happens for smaller cache sizes (i.e., 0.10 and smaller). For the trace we used, $s_{\max} \approx 1.6 \times 10^5$, $q \approx 13$ and the sum of the sizes of all possible items is 8.5×10^8. So for a cache size ratio of 0.1, the average amount of memory per queue is

$$0.10(8.5 \times 10^8)/13$$

which is 40 times $s_{\max}$. For block sizes larger than 10 times $s_{\max}$, there are typically less than four blocks per queue which results in significant tail fragmentation.

Figure 3.4 illustrates the tradeoff in block and queue fragmentation as block size varies with a cache size ratio of 0.1 and shows that the sweet spot in balancing the two is a block size that is roughly 5 times $s_{\max}$. Note that a cache size ratio of 0.1 one of the smallest size caches considered in our simulations for which performance is more sensitive to block size.

3.5 Future work

The fact that using FIFO degrades caching performance more than fragmentation suggests that a hybrid between FIFO and LRU would yield better overall performance. The fact that FIFO performs suboptimally as a replacement algorithm for key-value stores has also

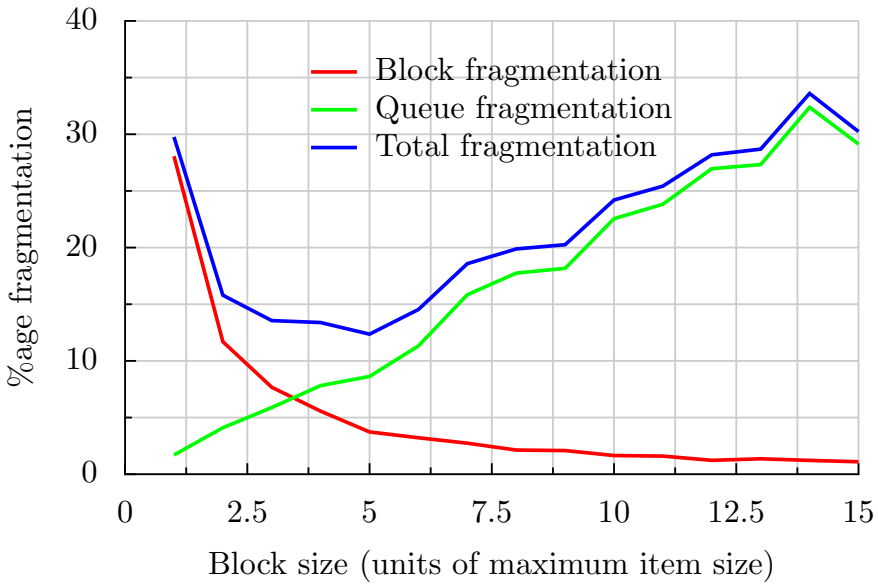


Figure 3.4: Types of fragmentation of CAMP-MALLOC at cache size ratio 0.1

been previously observed [57]. In general, the physical layout of items within the cache will differ from the LRU order. Evicting items in LRU order will result in more fragmentation which will require an algorithm to fill holes created by evictions or movement of data within the cache. An effective cache management policy needs to balance caching performance, fragmentation and the cost of moving data within the cache.

Chapter 4

Memory hierarchy design for caching middleware

4.1 Introduction

Advances in the storage industry have resulted in the recent introduction of non-volatile memory such as PCM, STT-RAM, and NAND Flash. These new forms of storage are anticipated to be much faster than rotating disk as permanent store, and less expensive than DRAM as volatile memory. Unlike DRAM, non-volatile memory retains its content in the presence of power failures. Table 4.1 compares some of the performance characteristics of the main types of storage media available at the time of this writing.

	DRAM	STT-RAM	Memristor	FeRAM	PCM	NAND Flash	disk
Read time (ns)	10–50	10–30	<10	20–40	20–70	25,000	$2–8 \times 10^6$
Write time (ns)	10–50	20–30	10–65	50–500	13–95	200,000	$4–8 \times 10^6$
Retention	<100 ms	weeks	>10 yrs	∼10 yrs	<10 yrs	∼10 yrs	∼10 yrs
Energy/bit $(pJ)^2$	2–4	0.1–1	0.1–3	0.01–1	2-100	$10–10^4$	$10^6–10^7$
3D capability	no	no	yes	yes	yes	yes	n/a

Table 4.1: Alternative data storage technologies [21, 58].

Caching middleware is an immediate beneficiary of non-volatile memory. In the current and next chapters, we explore the possibility of using NVM technology to improve the performance of two types of caching middleware.

The first type of caching middleware is cache augmented data store architecture [32] that extends a database management system with a key-value store that enables an application to lookup the result of queries which are repeated many times. The key insight is that query result look up is faster and more efficient that query processing for those workloads that exhibit a high read-to-write ratio, such as social networking applications [63, 60, 32, 33].

An example is memcached in use by popular internet destinations such as Facebook [60]. It is a distributed in-memory key-value store that augments a data store such as MySQL [4]. Today's memcached uses DRAM for fast storage and retrieval of key-value pairs. By using non-volatile memory as either a replacement or extension of DRAM, one may convert memcached to retain its key-value pairs after a short-lived power failure.

The second type of caching middleware is host-side caches that stage disk pages on a storage medium which is faster than permanent store and brings data closer to the application [66, 15, 68, 57, 69, 9, 35, 51, 44]. With these caches, a data item is either a disk block, a file, or an extent consisting of several blocks. For example, Dell Fluid Cache [15] stages disk pages referenced by a database management system such as MySQL on NAND Flash in order to hide the latency of retrieving these pages from a significantly slower permanent store. In our experiments with a social networking benchmark named BG, Fluid Cache enhanced the performance of MySQL anywhere from a factor of 3 to 20 [31]. It may be possible to boost performance further by extending this transparent cache to consist of both PCM and NAND Flash as PCM is significantly faster than NAND Flash.

These caching scenarios raise the following research questions:

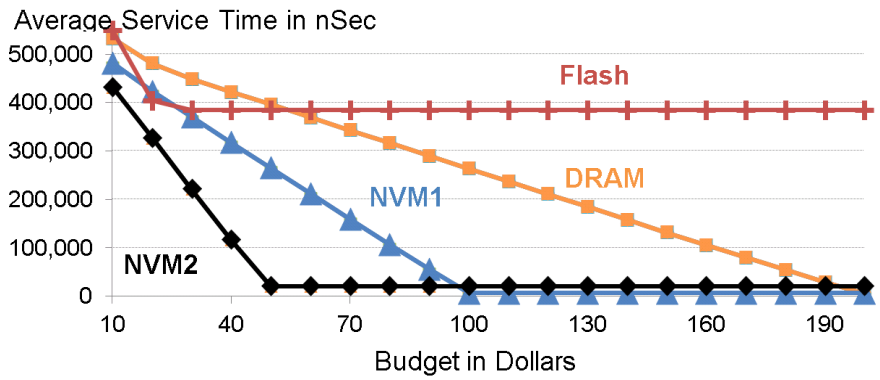


Figure 4.1: Average service time of processing a social networking workload with different choice of storage medium for the cache.

1. Given multiple choices of storage media and a fixed monetary budget, what combination of memory choices optimizes the performance of a workload? Is it appropriate to use only one storage medium or a mix?

2. What is the best policy for assigning key-value pairs across the selected choices of storage media? Should the system replicate key-value pairs across DRAM and NVM or partition them?

In the current and next chapters, we attempt to answer these questions. Unlike in Chapters 2 and 3 in which requests were issued in an online fashion, we use the independent reference model, that is, input sequences are made up of independent requests from a probability distribution over all data. Therefore, our solution to the second question is a static placement of data over storage media. The current chapter provides an answer to the first question, specifying an ideal combination of memory media for a particular budget and workload, whereas Chapter 5 assumes that the capacity and types of memory are already determined.

In this chapter, we use the term *cache* to refer to either one or several storage media used as a temporary staging area for data items. A *data item* might be a key-value pair with the key-value store or a disk page with the host-side cache. Each type of storage medium that contributes space to the cache is termed a *stash*. For example, in a key-value store, the cache

may consist of DRAM and non-volatile memory as two stashes. The stashes in a host-side cache may be PCM and Flash. A copy of a data item occupying the cache is also stored on *permanent* store. With memcached, the permanent store is a data store such as MySQL from which the key-value pair must be recomputed. With a host-side cache such as Dell Fluid Cache, the permanent store may be a Storage Area Network such as Dell Compellent. An application's read request for a data item fetches the data item from either the cache (a cache hit) or the permanent store (a cache miss) into *transient* memory for processing.

The different data items compete for the available space in the stashes that collectively constitute the cache. The goal is to assign data to the stashes with the objective of enhancing the performance of read and write operations referencing those data items. The systems literature has considered two divergent approaches to data placement: *tiering* and *replication*. Tiering maintains only one copy of a data item across the stashes. Replication constructs one or more copies of a key-value pair across stashes. We allow for each item to be placed on any subset of the stashes.

The first contribution of this chapter is an offline optimal algorithm that computes (1) the choice and sizes of the stashes that constitute a cache given a fixed budget and (2) a static placement of data items across the stashes identified in Step 1. The input to the algorithm is the workload of an application (frequency of access to its referenced data items, sizes of the data items, and the time to fetch a data item on a cache miss) and the characteristics of candidate stashes (read and write latency and transfer bandwidths, failure rate, and price). The proposed algorithm uses the distribution of access frequencies to guide overall design choices in determining how much if any of each type of storage media to use for the cache. The algorithm can also be used to guide high-level caching policy questions such as whether to maintain backup copies (replication) of data items in slower, more reliable storage media or whether to only keep a single copy of each item across stashes (tiering).

While the algorithm presented here does compute a sample placement of data across stashes,

the placement is primarily a tool for measuring the average response time for a particular choice of stash sizes. The actual placement of data to caches would naturally need to evolve in real time in order to adapt to fluctuations in the relative popularity of particular data items. This motivates the need for online algorithms for data placement as a future research direction, which we describe in more details in Section 4.6. While a particular choice of stash sizes for a fixed budget may no longer be optimal if workload characteristics change, recent history is the best available source of information for future workload. The method proposed here provides a systematic way to use this information to guide choices that must be made at the time a system is configured.

For example, Figure 4.1 shows the estimated average service time as a function of budget when the cache is limited to a single stash. Each line corresponds to a different storage medium used for the cache, with NVM_1 and NVM_2 corresponding to two representative non-volatile memory technologies (see Table 4.3 for their characteristics). When the budget is tight (small values of x-axis), NAND Flash is a better alternative than DRAM and comparable to NVM_1 because its inexpensive price facilitates a larger cache that enhances service time.

Figure 4.1 illustrates that except in the extreme case where the budget is large enough to store the entire database in DRAM, the two non-volatile memory options are preferable storage media to DRAM. This information would influence a design choice in determining which type of memory to use to cache key-value pairs. The algorithm is flexible and performs the same optimization to realize caches consisting of two or more stashes. In addition, it evaluates choices in which data items are stored in more than one stash.

Our method for cache configuration specifies a size for each stash in bytes. However, one typically purchases memory in certain granularities. We assume that in determining the amount of memory for each stash, the byte figures would be rounded to the nearest value available for each memory type.

The second contribution is our trace driven evaluation of the proposed method. The main lessons of this evaluation are as follows:

1. Some combination of storage media perform considerably better than others over a wide range of budget constraints. For example, the combination of Flash and NVM_2 is a good choice in the context of host-side caches for most budget scenarios (See Figure 4.8). DRAM and NVM_2 does best for cache augmented data stores for most budgets (See Figure 4.12).

2. After a certain threshold point that depends on database size and workload characteristics, spending money on larger and faster caches does not offer significant improvement in performance.

3. Before spending money on an expensive form of non-volatile memory that stores a small fraction of data items, better performance gains might be achievable by purchasing a slightly slower speed storage medium with a higher storage capacity that can stage all data items.

4. Optional replication of data items by our algorithm produces results that are slightly better than tiering and significantly superior to an approach that replicates all data items across stashes.

5. There can be several different placements that approximate the optimal placement in their average service time. These alternatives can be explored by limiting the set of placement options and comparing the average service time under the more restrictive scenario to the average service time in which all possibilities are allowed.

The rest of this chapter is organized as follows. Section 4.2 presents a model of the optimization problem using the language of cache augmented data stores as it is more general. Section 4.3 formalizes this optimization problem as an instance of the multiple-choice knapsack problem

and presents a near optimal algorithm to solve it. Section 4.4 demonstrates the effectiveness of this algorithm in deciding the choice of stashes and placement of data items across them for both cache augmented data stores and host-side caches using a trace-driven simulation study. We describe related work in Section 4.5 and future work in Section 4.6.

4.2 The model

We describe the model using the language of key-value stores because it is the most general. At the end of Section 4.2, we describe the few changes to adapt our methods for designing a host-side cache.

We model query sequences by a stream of independent events in which the occurrence of a particular query (key-value read request) or update (key-value write request) does not change the likelihood that a different query or update occurs in the near future. Independently generated events is the model employed by social networking benchmarks such as BG [5] and Linkbench [4]. If the probabilities of queries and updates to each key-value pair are known a priori, then a static assignment of key-value pairs to each storage medium is optimal. Before each request, the optimal placement minimizes the expected time to satisfy the next request. If the distribution does not change over time, then the same placement will be optimal for every request. In our simulation studies, we estimate the probability that a particular key-value pair is requested by analyzing the frequency of requests to that key-value pair in the trace file. We then determine an optimal memory configuration based on the those probabilities.

For each key value pair k, we assume the following four quantities are known:

- size(k) the size of the key-value pair in bytes.

- compute-time(k) the time to compute the key-value pair from the database.
- read-freq(k) the frequency of a read reference for key k.
- write-freq(k) the frequency of a write reference for key k.

Note that $\sum_k(\text{read-freq}(k) + \text{write-freq}(k)) = 1$.

There is a set $\mathcal{S}$ of candidate stashes for the cache, each made of a different memory type. For example, Table 4.3 shows the memory device types[1] and their parameters used in our simulations. Hence, in our experimental studies for the key-value store, $\mathcal{S}$ is defined as:

$$\mathcal{S} = \{\text{disk}, \text{Flash}, \text{NVM}_2, \text{NVM}_1, \text{DRAM}\}.$$

Each stash $s \in \mathcal{S}$, has the following characteristics:

- read-latency(s) the read latency of candidate stash s.
- write-latency(s) the write latency of candidate stash s.
- read-bandwidth(s) the read bandwidth of candidate stash s.
- write-bandwidth(s) the write bandwidth of candidate stash s.
- price-per-byte(s) the monetary cost of purchasing a byte of s.

The time to read k from s is:

$$\text{read-time}(k, s) = \text{read-latency}(s) + \text{size}(k)/\,\text{read-bandwidth}(s)$$

[1]Since parameters for emerging non-volatile memory technology are not completely known, we used two representative types of non-volatile memory, which we call NVM_1 and NVM_2.

The time to write k to s is:

$$\text{write-time}(k, s) = \text{write-latency}(s) + \text{size}(k)/\,\text{write-bandwidth}(s)$$

4.2.1 Placement options

A copy of a key-value pair k can be placed in one or more of the stashes. We define a *placement option* P for a key-value pair to be a subset of the set of stashes. For example, the placement option $P = \{\text{Flash, DRAM}\}$ represents having a copy of a key value pair on both Flash and DRAM. The placement option $\varnothing$ represents the scenario where a key-value pair is not stored in the cache at all. In each experiment, we define a set of possible placement options for a key-value pairs that allows us to study the trade-offs between different options. For example, in tiering, there is at most one copy of a key value pair in the entire cache, so the collection of possible placements would be: $\varnothing$, {disk}, {Flash}, {NVM_2}, {NVM_1}, {DRAM}. In examining the trade-off between tiering and replication for a system with Flash and DRAM, the set of possible placements would be: $\varnothing$, {Flash}, {disk}, {Flash, disk}.

We define an optimization problem that minimizes the expected time per request given a set of possible placement options and budget to purchase the memory for each stash. For each key-value pair k and each possible placement option P, there is an indicator variable $x(k, P) \in \{0, 1\}$, indicating whether k is placed according to P. If $P = \{\text{Flash}, \text{DRAM}\}$ and $x(k, P) = 1$, then we are using replication with a copy of k on both the Flash stash and the DRAM stash. The following constraint says that k has exactly one placement:

$$\sum_P x(k, P) = 1,$$

where the sum is taken over all possible placement options, including $\varnothing$.

If $x(k, P) = 1$, then we must purchase size(k) bytes of memory for each stash in P. Again, if $P = \{\text{Flash}, \text{DRAM}\}$, we need size($k$) bytes of Flash and size(k) bytes of DRAM for the copies of key-value pair k. Thus, the monetary price of having key-value pair k in placement option P is:

$$\text{price}(k, P) = \sum_{s \in P} \text{cost-per-byte}(s)\ \text{size}(k).$$

If the overall budget is M, then the total cost, summed over all key-value pairs must be at most M:

$$\sum_{k,P} \text{price}(k, P)\, x(k, P) \leq M.$$

4.2.2 Expected service time

The objective of the optimization is to expedite the average processing time of a request. This translates to minimizing the average service time. For a key-value pair k and placement option P, we define service-time(k, P) as the average service time of requests referencing k if k is assigned to stashes specified by the placement P. The placement option $P = \varnothing$ is a special case because it does not assign k to the cache, requiring every read reference to compute k using the data store (i.e., the permanent store) and incur its service time compute-time(k). In this case, service-time($k, \varnothing$) = read-freq(k) $\cdot$ compute-time(k).

For $P \neq \varnothing$, there are three components to the service time for a key value pair k if it is placed according to P: (1) the time spent reading k, (2) the time spent writing k, and (3) the average cost of restoring the copies of k to a failed stash after repair. We consider each in turn.

If key-value pair k is assigned according to P, then upon a read request to k, it is read from

the stash with the fastest read time for k:

$$\Delta_R(k, P) = \min_{s \in P} \text{read-time}(k, s).$$

The average service time to read k is the frequency of a read, times the time for the read: read-freq$(k)\Delta_R(k, P)$.

Upon a write to k, all the copies of k across the stashes dictated by P must be updated. One may assume different models for the service time of this write operation such as a concurrent write to all copies with the slowest stash dictating the time to write

$$\Delta_W(k, P) = \max_{s \in P} \text{read-time}(k, s)$$

or a serial write where the different stashes with a copy are written one after another:

$$\Delta_W(k, P) = \sum_{s \in P} \text{write-time}(k, s).$$

The average time writing k will be the frequency of a write request to k times the time for the writes: write-freq$(k)\Delta_W(k, P)$.

A strength of the proposed model is its flexibility to include alternative definitions of the write model. Our investigation of the concurrent and serial write models showed very little difference as the slowest stash dominates the write cost. Hence, we assume a serial write model.

We model failures as a rate λ that defines the inter-arrival between two failures in terms of the number of requests as $1/\lambda$. For example, a failure rate of 0.001 ($\lambda = 0.001$) means that the average number of requests between occurrences of two failures is 1000. We define a failure event F as the set of stashes that fail at the same time. To simplify discussion, we assume F consists of one failure. However, the model is general enough to express the

optimization problem in which any possible subset F of stashes can fail. For each such failure event, we determine the cost of restoring the contents of the impacted stash. We require a failure rate λ_F for every possible failure event F. Section 4.4.1 details how we compute λ_F in our experiments using both trace files and parameter settings of stashes.

In the presence of failures, it may be advantageous to store a key-value pair k on more than one stash. If k is stored on two stashes and one fails, then the time to repopulate the failed stash is reduced by retrieving a copy of k from the other stash. This is the only motivation for replicating a key-value pair across more than one stash. Having an extra copy of a key-value pair increases the cost of updating on writes. This trade-off between the cost of updating an additional copy and the benefit of having the extra copy in case of failure determines how many copies of a key-value pair is optimal. The optimization problem we define here automatically takes these considerations into account. For each triple (k, P, F), we define fail-cost, the cost of restoring k after a failure event F given that k is stored according to placement option P. fail-cost$(k, P, F) = 0$ if the set of stashes that fail has no overlap with P (i.e., $P \cap F = \varnothing$). Otherwise, there are two components to the cost: first, the cost of retrieving a copy of k. If all stashes of P are wiped clean after a failure event F (i.e. $P \subseteq F$), then retrieval-cost(k, P, F) = compute-time(k). Otherwise k is read from the fastest stash still available:

$$\text{retrieval-cost}(k, P, F) = \min_{s \in P - F} \Delta_R(k, s).$$

The second component of the incurred cost after a failure is restoring k to the stashes in P that failed during failure event F:

$$\text{restore-cost}(k, P, F) = \sum_{s \in P \cap F} \Delta_W(k, s).$$

Now, we assemble all components of the service time of a request referencing a key-value pair k assigned according to the placement option P:

$$\begin{aligned}\text{service-time}(k,P) = {} & \text{read-freq}(k)\Delta_R(k,P) \\ & + \text{write-freq}(k)\Delta_W(k,P) \\ & + \textstyle\sum_F \lambda_F \cdot \text{restore-cost}(k,P,F) \\ & + \textstyle\sum_F \lambda_F \cdot \text{retrieval-cost}(k,P,F)\end{aligned}$$

The goal is to select values for the variables $x(k,P) \in \{0,1\}$ that minimizes:

$$\sum_{k,P} \text{service-time}(k,P)\, x(k,P).$$

In the next section, we show how this optimization problem can be solved using known techniques for the multiple-choice knapsack problem. Knapsack problems are typically maximization problems, so we define an equivalent maximization problem to the problem stated above which maximizes a benefit instead of minimizing a cost. For each placement P and key-value pair k, define:

$$\text{benefit}(k,P) = \text{service-time}(k,\varnothing) - \text{service-time}(k,P).$$

Note that the benefit of placement $\varnothing$ is 0 (i.e., $\text{benefit}(k,\varnothing) = 0$). The constraints and definitions for the problem are unchanged, except that the goal is now to select a placement for each k that maximizes:

$$\sum_{k,P} \text{benefit}(k,P)\, x(k,P).$$

An optimal solution to the minimization problem is also an optimal solution for the maxi-

mization problem and vice versa.

To summarize, the cache configuration problem is to find a placement for each k (i.e., values for the indicator variables $x(k, P) \in \{0, 1\}$) that maximizes:

$$\sum_{k,P} \text{benefit}(k, P)\, x(k, P),$$

subject to the constraints that for each k:

$$\sum_{P} x(k, P) = 1,$$

and for budget M:

$$\sum_{k,P} \text{price}(k, P)\, x(k, P) \leq M.$$

Every sum over P is taken over the set of possible placement options that we define for a particular experiment (i.e., instance of the optimization problem). The values of the indicator variables dictate the size of the stashes: each stash must be large enough to hold a copy of every key-value pair that has a copy on that stash. For a stash s, we must sum over all placement options that include a copy of a key-value pair on s. The total size of stash s is then:

$$\sum_{k,P \ni s} \text{size}(k)\, x(k, P).$$

4.2.3 Host-side caches

We use the same framework to optimize a configuration for host-side caches. There are two key conceptual differences that are accommodated using our logical formalism. First,

today's host-side caches uses Flash (instead of DRAM of cache augmented data stores, e.g., memcached) and it is natural to extend them with non-volatile memory. Hence, the set of stashes to consider are: $\mathcal{S} = \{\text{Flash}, \text{NVM}_1, \text{NVM}_2\}$. Second, a data item k might be either a disk page or a file. The cost of not assigning k to the cache means it must be serviced using the permanent store (might be a disk controller, a RAID, a Storage Area Network):

$$\text{service-time}(\varnothing, k) = \text{read-freq}(k)\ \text{read-time}(\text{disk}, k) + \text{write-freq}(k)\ \text{write-time}(\text{disk}, k).$$

read-time(disk, k) and write-time(disk, k) are the time to read and write k to disk. Other definitions of Section 4.2 are unchanged.

4.3 The multiple-choice knapsack problem

In the knapsack problem, there is a set of items available to pack into a knapsack. Each item has a benefit and a weight. The knapsack has a weight limit and the goal is to select the set of items to pack into the knapsack that maximizes the total benefit of the items selected while not exceeding the weight limit of the knapsack. In the multiple-choice knapsack problem (MCKP), the items are partitioned into groups with the additional constraint that at most one item from each group can be selected.

In the cache configuration problem, each key-value pair has a set of placement options (including the option of not placing it on any of the memory banks). Therefore each key-value pair defines a class from which we are selecting at most one option. Each selection has an associated benefit (as defined in Section 4.2) and each selection has a price that corresponds to the "weight" of the choice in the knapsack problem. Therefore the cache configuration problem can be cast as an instance of MCKP.

The knapsack problem is a well-studied problem in the theory literature and is known to be

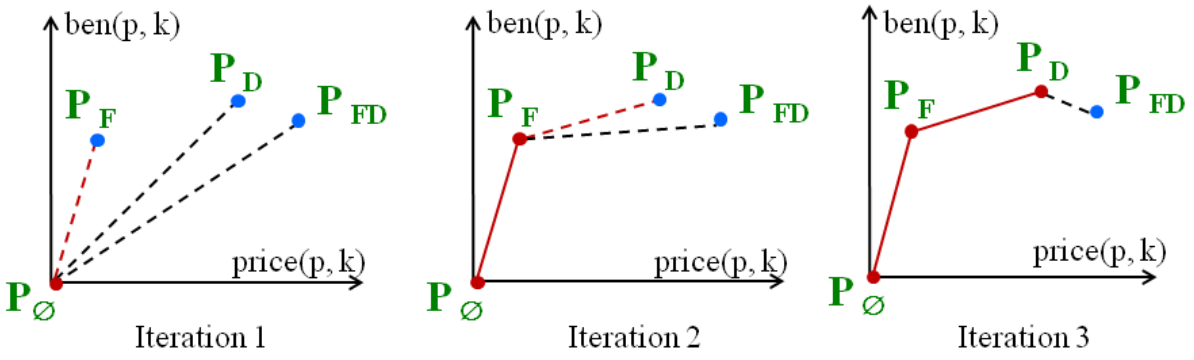


Figure 4.2: Illustration of SetViableOptions. Before the first iteration, the viableList for k is initialized to $[\varnothing]$. The algorithm examines each segment connecting $P_\varnothing$ to the three placements to the right and selects P_F because the segment from $P_\varnothing$ to P_F has the largest slope. The viableList for k is now $[\varnothing, \mathrm{F}]$. In the second iteration, the algorithm looks at the segments connecting P_F to the two placements to the right and selects P_D because the segment connecting P_F to P_D has the largest slope. The viableList for k is now $[\varnothing, \mathrm{F}, \mathrm{D}]$. FD is not added to k's viable list in the third iteration because the slope from P_D to P_{FD} is negative, indicating that placing k on Flash and DRAM costs more money and brings less benefit than placing k on DRAM only.

NP-hard [25]. Since the knapsack problem is a special case of MCKP in which each item is in a category of its own, MCKP is also NP-hard. The books [52] and [43] are dedicated to the knapsack problem and its variants and both include a chapter on MCKP. One can consider a linear programming relaxation in which the indicator variables $x_{p,S}$ can be assigned real values in the range from 0 to 1 instead of $\{0, 1\}$ values. In the case of MCKP, the linear programming relaxation can be optimally solved by the following greedy algorithm: start with the placement $\varnothing$ for all of the key-value pairs. This placement has 0 overall benefit and 0 monetary cost. The algorithm considers a sequence of changes (to be defined later) in which the placement of a key-value pair is upgraded to a more beneficial and more costly placement option. This process continues until the money runs out and the last item can only be partially upgraded. The solution obtained by the greedy algorithm is optimal for the LP-relaxation of the problem [67] and therefore at least as good as the optimal integral solution. Moreover, only one item is fractionally placed. The algorithm used here follows the greedy algorithm but stops short of the last upgrade that results in a fractional placement.

Let $\mathrm{OPT}_{\mathrm{frac}}$ and $\mathrm{OPT}_{\mathrm{int}}$ represent the total benefit obtained by the optimal solutions for the fractional and integer versions of MCKP respectively. Let $\mathrm{GR}_{\mathrm{frac}}$ and $\mathrm{GR}_{\mathrm{int}}$ represent the benefit obtained by the greedy algorithm for the fractional and integer versions of MCKP respectively. We have:

$$\mathrm{GR}_{\mathrm{int}} \leq \mathrm{OPT}_{\mathrm{int}} \leq \mathrm{OPT}_{\mathrm{frac}} = \mathrm{GR}_{\mathrm{frac}}.$$

Moreover, the difference in overall benefit between the greedy integral solution and the greedy fractional solution is at most the benefit obtained by the last fractional upgrade. Since individual key-value pairs are small with respect to the overall database size, the effect of not including the last partial upgrade is not significant. Thus, while MCKP is NP-complete, the method we use provably approximates the optimal cache configuration with additive error that is less than the benefit of a single key-value pair.

The running time of the algorithm depends on n, the number of key-value pairs, and p, the number of different placement options considered. With replication, the largest p would be is 2^m, where m is the number of different stashes since any subsets of the stashes could be a placement option. For tiering, $p = m + 1$ because each placement option is a single stash. The additional 1 comes from the $\varnothing$ option. Our implementation uses a priority queue to select the next upgrade resulting in an overall running time of $O(np \log np)$. There are more complex algorithms to find the greedy solution that run in time $O(np)$ [18, 76]. We chose the $O(np \log np)$ implementation because it was easier to implement and ran sufficiently quickly. Recall that the algorithm is used as an offline step in the design of a cache, not to decide data placement in real time.

We now give a description of the greedy algorithm for MCKP. The p different placement options, $P_0, \ldots, P_{p-1}$, are sorted in increasing order by price, so $\mathrm{price}(k, P_i) \leq \mathrm{price}(k, P_{i+1})$. The option P_0 is the $\varnothing$ option, and $\mathrm{price}(k, P_0) = \mathrm{benefit}(k, P_0) = 0$.

Not every placement option is a reasonable choice for every key value pair. For example, if placement option P for key k has lower benefit and higher price than option P' for k, then P should not even be considered as a viable option for key k. Thus, the first step is a preprocessing step in which a list of viable options is determined for each k. If j is not on k's viable list, then P_j will never be considered as an option for k. The pseudocode for SetViableOptions is given in Algorithm 3.

Algorithm 3 SetViableOptions(k).

```
viableList(k) ← empty list
i ← 0
do
    add i to viableList(k)
    gradient ← −∞
    for j = i + 1, . . . , p − 1 do
        g ← (benefit(k, P_i) − benefit(k, P_j)) / (price(k, P_i) − price(k, P_j))
        if g > gradient then
            gradient ← g
            i ← j
while gradient ≤ 0 and i ≤ #keys
return viableList(k)
```

The procedure SetViableOptions is illustrated graphically in Figure 4.2. For this example, only two memory banks are considered: DRAM and Flash. We consider four placement options for a key-value pair k. The key-value pair can be placed on both DRAM and Flash (FD), Flash only (F), DRAM only (D), or neither ($\varnothing$). Each placement option is represented as a point with the horizontal axis representing the price of placing k on that placement option and the vertical axis representing the benefit. Essentially, a placement option is viable for k if its corresponding point lies on the convex hull of all the points and the slope of the segment connecting it to the previous point is positive.

The fact that the point corresponding to P_D is placed higher than P_{FD} for the particular key-value pair k used in Figure 4.2 means that $\text{benefit}(k, P_D) > \text{benefit}(k, P_{FD})$. This arrangement may not be the same for all key-value pairs k and will in general, depend on a number of different parameters, especially the write frequency of k. The more frequently k is

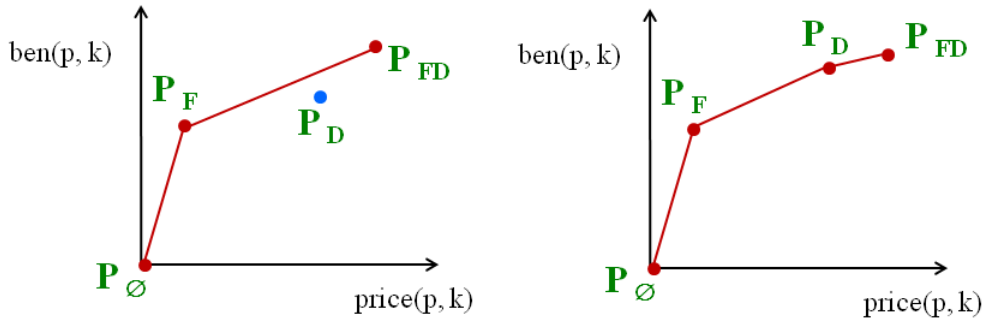


Figure 4.3: The viableList for the key on the left is $[\varnothing, F, FD]$. After F was added, the segment from P_F to P_{FD} had a larger slope than the segment from P_F to P_D, so option D was bypassed and FD was added to k's viable list. The viableList for the key on the right is $[\varnothing, F, D, FD]$.

written, the more costly it is to maintain the extra copy of k on Flash. Figure 4.3 shows two other possible scenarios for the list of viable placements for a key-value pair.

After the viable list for each key-value pair has been determined, the greedy algorithm initializes the placement for each key-value pair to $\varnothing$. In each iteration the greedy algorithm selects the key-value pair k such that the slope of the segment from k's current placement to k's next viable placement is maximized. Then greedy upgrades the placement for k to the next placement option on its list of viable placements. The process continues until the money runs out or until there is no upgrade that improves the overall benefit (i.e., until all the upgrade gradients are negative). The pseudocode for the greedy algorithm is given in Algorithm 4. The greedy algorithm is illustrated with a small example in Figure 4.4.

We illustrate the GreedyPlacement algorithm with 3 key-value pairs using two candidate stashes DRAM and Flash. The graph for each key-value pair is shown in Figure 4.4. Each point in the graph is a placement option with the horizontal axis representing the price and the vertical axis representing benefit. The graph for k_1 and k_3 consists of two segments while the one for k_2 consists of three segments. The shown value next to a segment is its slope, e.g., the two segments of k_1 have a slope of 5 and 0.2, respectively.

Order	Segment Slope	Key-value Pair	Change in Placement	Upgrade description	Price
1	5	k_1	$\varnothing \to F$	Assign k_1 to Flash	1
2	4	k_2	$\varnothing \to F$	Assign k_2 to Flash	1
3	3	k_3	$\varnothing \to F$	Assign k_3 to Flash	1
4	.5	k_3	$F \to FD$	Replicate k_3 to DRAM	4
5	.4	k_2	$F \to D$	Move k_2 from Flash to DRAM	3
6	.2	k_1	$F \to D$	Move k_1 from Flash to DRAM	3
7	.1	k_3	$D \to FD$	Replicate k_3 to Flash	1

Table 4.2: Different iterations of the GreedyPlacement algorithm with the three keys. The order of upgrades is according to the slope of line segments shown in Figure 4.4.

Algorithm 4 GreedyPlacement(*budget*)

moneySpent $\leftarrow 0$
current $\leftarrow$ [0 for all k]
for ever **do**
  Select k with the largest upgrade gradient
  if *current*$[k] + 1 \geq$ length(*viableList*(k)) **then**
    return
  $i \leftarrow$ *viableList*(k)[*current*$[k]$]
  $j \leftarrow$ *viableList*(k)[*current*$[k] + 1$]
  deltaBenefit $\leftarrow$ benefit(k, P_j) $-$ benefit(k, P_i)
  deltaPrice $\leftarrow$ price(k, P_j) $-$ price(k, P_i)
  upgradeGradient $\leftarrow$ *deltaBenefit*/*deltaPrice*
  if *deltaPrice* $+$ *moneySpent* $>$ *budget* or *upgradeGradient* ≤ 0 **then**
    return
  moneySpent $\leftarrow$ *moneySpent* $+$ *deltaPrice*
  current$[k] \leftarrow$ *current*$[k] + 1$

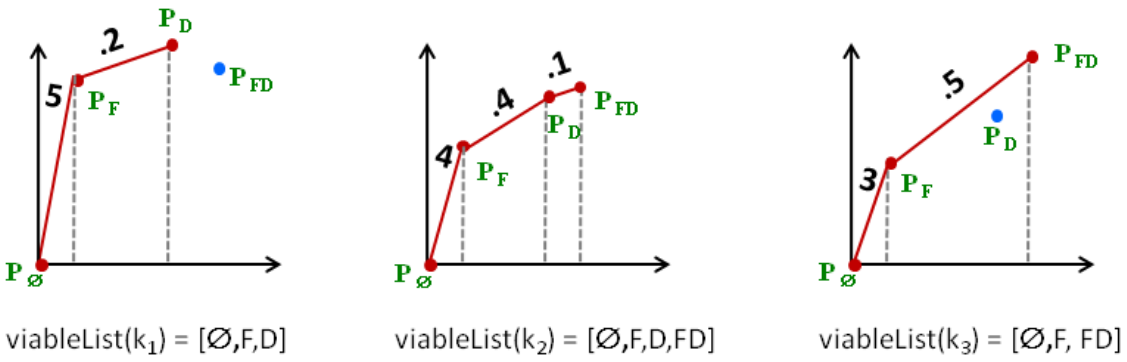


Figure 4.4: Three different key-value pairs and a graph of their placement options.

The horizontal distance between the endpoints of the segments (distance between the dotted vertical lines) is the price of the upgrade. For example, with k_2, the price of an upgrade from Flash to DRAM (distance between the vertical lines P_F and P_D) is higher than the upgrade price from DRAM to both Flash and DRAM (distance between the vertical lines P_D and P_{FD}).

Table 4.2 shows the different iterations of the GreedyPlacement algorithm with a price of 1 for Flash and 4 for DRAM. We assume the size of the three key-value pairs is identical. Table 4.2 shows how GreedyPlacement assigns according to segments with the highest slope first, see column 2 labeled "Segment Slope". The highest slope segment belongs to k_1, transitioning its assignment from $\varnothing$ to the placement option Flash. This is repeated with the second and third highest slope segments, transitioning the assignment of both k_2 and k_3 to Flash. The next highest slope segment is 0.5 (see row 4) and belongs to k_3. It changes the placement of k_3 from DRAM to DRAM and Flash, replicating k_3 onto DRAM with an additional cost of 4, see last column of Table 4.2. The next line segment (slope of 0.4 belonging to k_2) changes the placement of k_2 from Flash to DRAM, resulting in a price of 3, i.e., cost of 4 for DRAM and saving of 1 by removing k_2 from Flash. This process continues until the available budget is exhausted.

A budget of 11 enables the first five iterations of GreedyPlacement. Its final placement will have k_1 on Flash, k_2 on Flash and DRAM, and k_3 on DRAM. A budget of 13 accommodates the sixth iteration to upgrade k_1 to DRAM.

4.4 Evaluation

In this section we used the algorithm presented in Section 4.3 as a tool to evaluate different mixes of stashes under varying budget constraints in the context of key-value store and host-side caches. Table 4.3 shows the parameters for the five types of memory used in this study, including their read and write latency, read and write bandwidth, price in dollars per gigabyte and mean time between failures. The actual parameters of current non-volatile memory technology are still undetermined, so we selected two representative types of non-volatile memory to use in this study. The methods of Section 4.3 can take as input any set of storage devices and corresponding parameters.

A summary of the lessons learned from this evaluation are presented in Section 4.1. The rest of this section is organized as follows. In Subsection 4.4.1, we show how to compute the failure rates using the trace file in combination with the parameter settings of a candidate storage medium (see Table 4.3). Next, in Subsections 4.4.2 and 4.4.3 we present a trace-driven evaluation of host-side and key-value store caches in turn.

4.4.1 Failure rates

Recall from Section 4.2.2 that the inter-arrival between two failures is quantified in terms of the number of requests as $1/\lambda$ where λ is the rate of failures. For example, $\lambda = 0.001$ means that on the average, there are 1000 requests between two failure occurrences. This models two kinds of failures: 1) power failures that cause a volatile stash such as DRAM to lose its

	NVM_1	NVM_2	DRAM	Flash	disk
Read Latency in ns	30	70	10	25000	2×10^6
Write Latency in ns	95	500	10	2×10^5	2×10^6
Read Bandwidth in MB/sec	10×1024	7×1024	10×1024	200	10
Write Bandwidth in MB/sec	5×1024	1×1024	10×1024	100	10
Price in dollars per Gig	4	2	8	1	.1
MTTF/MTBF in hours	21875	43776	8750	87576	87576
MTTR in hours	24	24	10	24	24
MTTF/MTBF + MTTR in years	2.5	5	1	10	10

Table 4.3: Parameter settings of storage medium used in experimental evaluation.

content, and 2) hardware failures that require a stash such as non-volatile memory to be replaced with a new one. The former is characterized by Mean Time Between Failure (MTBF) and the latter is quantified using Mean Time To Failure (MTTF). Both model constant failure rates, meaning, in every time unit a failure has the same chance as any other time instance. MTBF is used for power failure because it pertains to a condition that is repairable. MTTF is used for devices because we assume they are non-repairable and must be replaced with a new one. Both incur a Mean Time To Repair (MTTR). With power failure, MTTR is the time to restore the power and for the system to warm-up the volatile memory with data. With non-volatile memory failure, MTTR is the time for the system operator to shutdown the server, replace the failed non-volatile memory with a new one, restore the system to an operational state, and incur the overhead to populate the new empty non-volatile memory with data.

4.4.2 Host-side cache for mail server

We require a failure rate λ_F for every possible failure event F. In our experiments, we only consider failure events that consist of a single device failure based on the assumption that failure events are sufficiently infrequent that it is unlikely that one memory device fails while another is still down. In these studies, to determine λ_F where F consists of a single stash (i.e., $F = \{s\}$), we multiply the number of requests per hour in the trace times $(MTTF + MTTR)$

for non-volatile memories and $(MTBF + MTTR)$ for DRAM. The number of requests per hour is determined by dividing the total number of requests in the trace by the time (in hours) over which the trace was gathered. A more elaborate way of modeling failure rates would be required for failure events in which more than one device fails.

Today's host-side caches employ one storage medium, namely Flash. This section considers an extension consisting of Flash, NVM_1 and NVM_2 as possible stashes. For our analysis, we used the disk block traces from a production mail server used by a University on a daily basis for one week [46] and several different production servers at Microsoft [42]. The latter includes the back-end server of Live Maps that displays satellite images and photographs of locations for an 18-hour period, the Microsoft Exchange server for an 18-hour weekday period, and a Micorosoft file server trace that covers a 6-hour period (see [42] for details). The former consists of 458 million requests to 14.7 million blocks. The total size of the requested blocks is 56.25 Gigabytes. Obtained results from all traces are similar and highlight the following main lessons:

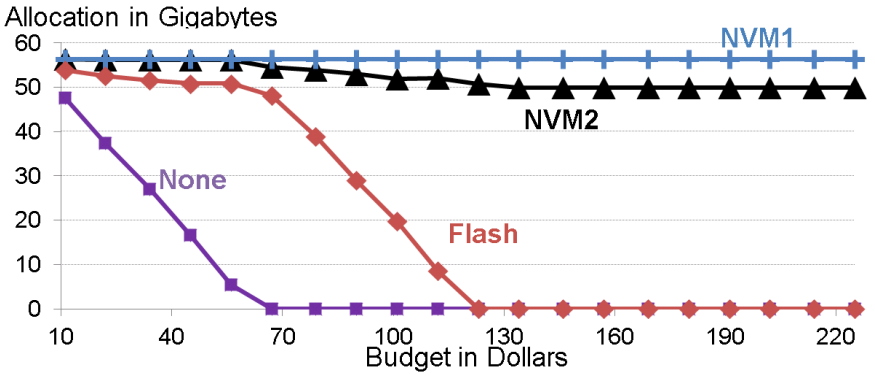
- The ability to replicate some objects (those that are not updated) does result in non-trivial improvement in average service time when failures are taken into account.
- Forcing replication is very costly—because the price is high for data that is updated.
- When a large budget is available, the optimal place for some objects is both stashes and for others is just the single fastest stash. This depends on whether they are updated.

Due to the similarity of the observed trends, this section presents experimental results from the University production mail server only. The first set of experiments examine different cache designs with a tiering policy in which each disk page is stored on at most one stash. In the first of these experiments, all three forms of memory (Flash, NVM_1 and NVM_2) are available as placement options. Figure 4.5a shows the allocation of blocks to stashes as

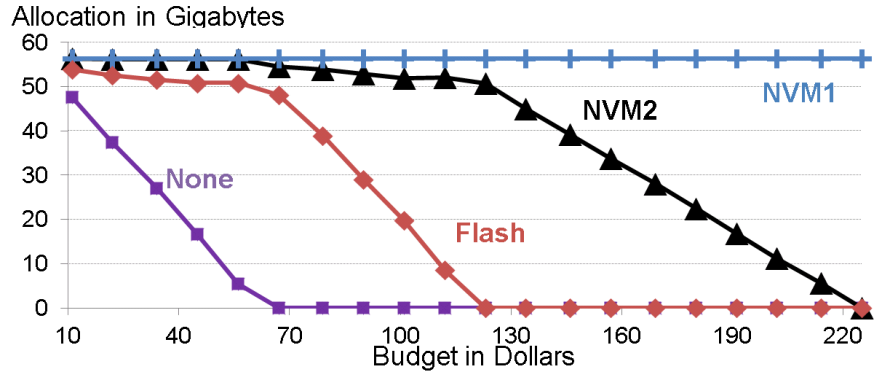
the budget is increased up to $225. For a particular budget, the size of each stash is the vertical distance between the line labeled with that stash and the line below. There is no budget at which the optimal allocation has disk pages in NVM_1 and disk pages not stored in the cache at all. In other words, before spending any money on upgrading disk blocks to NVM_1, it is more cost effective to get all of the disk blocks into the cache. There is sufficient variation in request frequency, however, that for budgets in the range of $100, it is optimal to have disk pages spanning Flash, NVM_2, and NVM_1. Another significant feature of the optimal allocation is that the allocation (and therefore the performance) does not change for budgets larger than $124. Even though NVM_1 has both faster read and write times than NVM_2, it is more favorable to keep 89% of the disk pages in NVM_2, independent of cost. The reason is that failure costs play a significant role, despite the fact that the likelihood of a failure is very small. Disk blocks that are placed in NVM_2 under the optimal placement have relatively low request frequency (averaging 3.4 requests over the course of the week-long trace), and therefore, there is less advantage to having them in NVM_1. Even though failures are very unlikely, the difference in failure rate between NVM_1 and NVM_2 becomes significant in expectation when multiplied times the cost of recovering the page from disk in the event of a failure. By contrast, the disk pages allocated to NVM_1 have a much higher request rate (averaging 233 over the course of the trace) and the benefit of the faster response time of NVM_1 more than offsets the increased probability of having to retrieve them from disk in the case of a failure.

In a single server system, it may not make sense to average over the effect of events like failures that happen once every few years even if they are significant in expectation. On the other hand, with a multi-node cache configuration consisting of a large number of servers, the likelihood of failures in a shorter period of time is much higher and it is sensible to average in their impact.

Figure 4.5b shows the same experiment run except that the cost of failures is not included in



(a) The cost of failures is included.



(b) The cost of failures is not included.

Figure 4.5: The optimal partition of the disk pages among the stashes as the budget varies. The amount allocated to each stash is the vertical distance between the line labeled with that memory type and the one below. In the first graph, the cost of failures is included. For budgets beyond \$124, most of the disk pages are in NVM_2. In the second graph, the cost of failures is not included. At the highest budget (\$225), all the disk pages are in NVM_1.

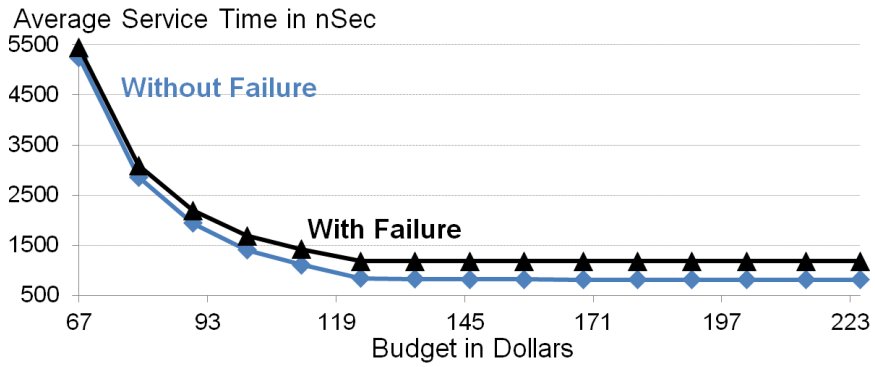


Figure 4.6: The average cost to service requests under the optimal cache configuration for different budgets.

the service time. The service time for a disk page is just the expected time servicing read and write requests to the page. With the cost of failures removed, the optimal placement (with unlimited budget) does have all of the disk pages in NVM_1. Although the allocation changes as the budget is increased, it's not clear whether the additional expenditure has a significant impact on the expected service time. Figure 4.6 shows the expected service time under the optimal configuration and placement for each budget. One line corresponds to the first set of allocations in which the cost of restoring after failures is taken into account. The second line corresponds to the scenario in which failures are not included. The difference between the two lines is the expected cost of failures. The difference is more pronounced (higher than 40% difference) with higher budgets as more disk pages are placed in either NVM_1 or NVM_2. The graph also indicates that performance does not improve significantly in either scenario past a budget of $124. Thus, even though it is optimal with no failures to store everything in NVM_1 if the budget allows, the improvement past $124 is not significant.

It may be favorable to have a cache consisting of fewer stashes. If so, our approach can be used to select a good set of memory types to include. In the next set of experiments we evaluate the impact on service time when the set of memory types is restricted. In Figure 4.7, each curve represents an experiment in which there is exactly one stash. A disk page can either be placed on that stash or excluded from the cache. NVM_2 generally does better than NVM_1 for anything but the full $225 budget because more disk pages can fit in the cache. However, when the budget reaches the maximum $225, the average response time with NVM_1 is 807 ns as compared to 3900 with NVM_2. In Figure 4.8, each curve represents an experiment in which there are exactly two stashes. A policy of tiering is employed, so a disk page can either be included in one of the two stashes or excluded from the cache. A combination of Flash and NVM_1 does better for a broader range of budgets, but NVM_2 and NVM_1 does better at the higher end of the scale.

Finally, we evaluate whether it is beneficial to allow some replication. Disk pages with low

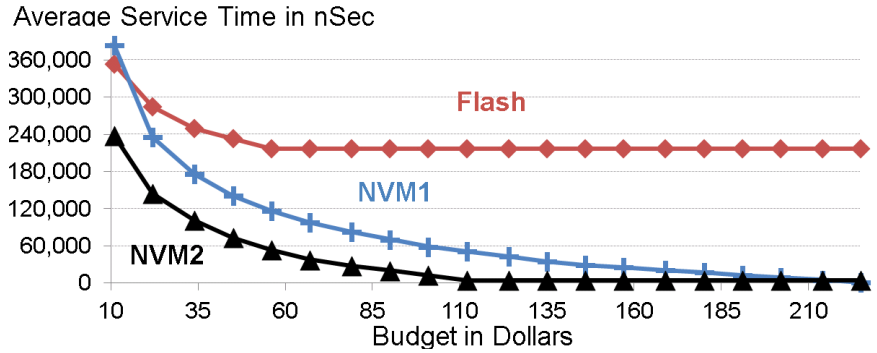


Figure 4.7: The average cost to service requests when there is only one stash. When the budget reaches the maximum \$225, the average response time with NVM_1 is 807 ns compared to 3900 with NVM_1

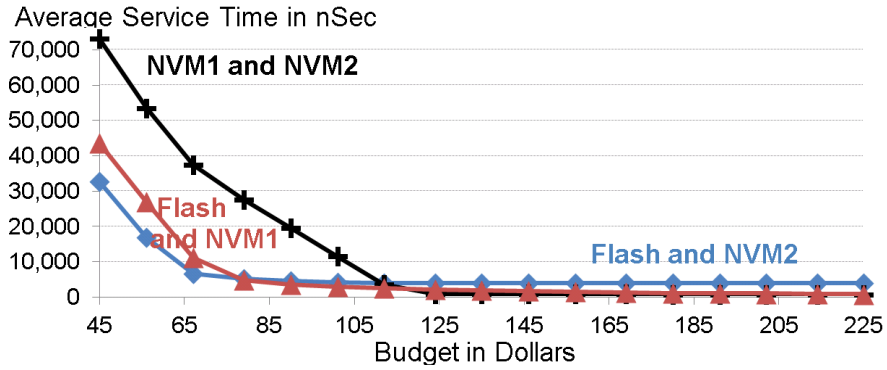


Figure 4.8: The average cost to service requests when there are two stashes. A tiering policy is employed so that a disk page an reside in at most on stash.

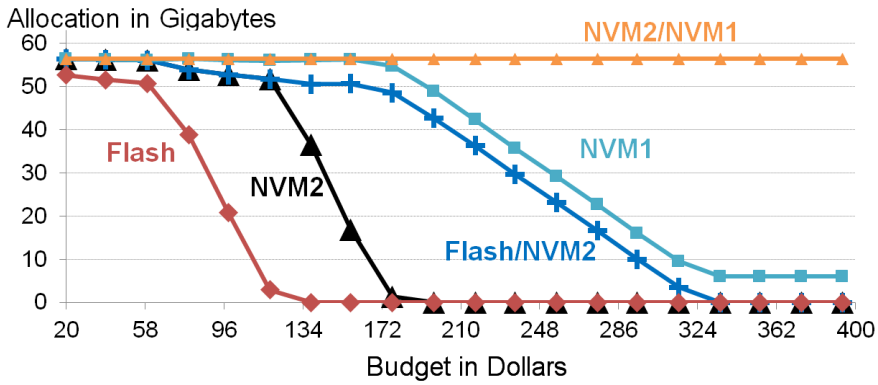


Figure 4.9: The optimal partition of disk pages among stashes when all 8 placements are allowed.

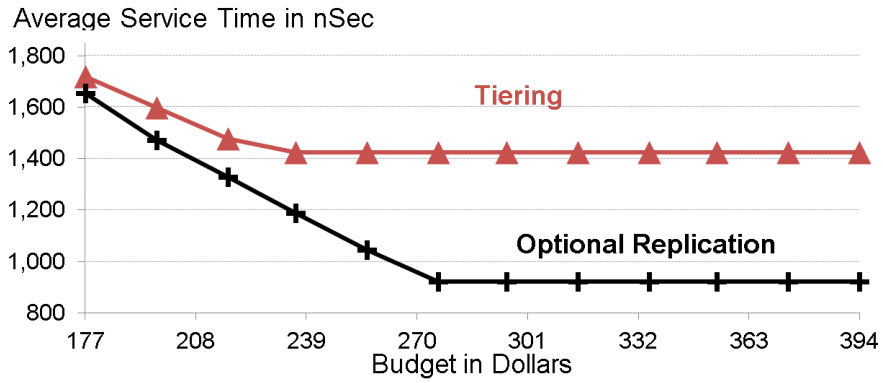


Figure 4.10: Tiering and optional replication with two stashes: Flash and NVM_1. The average service time for forced replication is not shown because even at the very highest budget range, the average response time was $212,459$ ns.

write rates will incur less overhead in having the additional copy on a less expensive but more reliable stash. Figure 4.9 is the allocation of disk pages to stashes when all 8 possible placements on the three stashes is allowed. No disk pages were ever placed on all three caches. The placement option $\{\text{Flash}, \text{NVM}_1\}$ had a very small allocation but was removed from the graph because it was too difficult to see.

Finally, we compare tiering and replication with a cache that includes Flash and NVM_1. We consider two variants of replication. Under *optional replication*, a key-value pair in NVM_1 may or may not also reside in Flash. Under *forced replication*, every key-value pair in NVM_1 must also have a copy in Flash. Naturally, the most flexible variant (optional replication) will be at least as good as the other two policies (forced replication and tiering).

The graph in Figure 4.10 shows that the optimal placements under tiering and optional replication do differ as there are some disk pages that are written so infrequently that the cost of maintaining the additional copy is offset by the expected cost of restoring a copy to NVM_1 in case of a failure. The forced replication option is not even shown because it was very costly in comparison to the other two policies. The trace contained a significant number of disk pages that were updated frequently and therefore incur a high cost for replication. For the high budget range, the average response time for tiering and optional replication are approximately 1.4μs and .9μs, respectively. The corresponding value for forced replication is 212μs.

4.4.3 Cache-augmented data stores

Today's caches such as memcached use DRAM to store key-value pairs. An instance loses its content in the presence of a power failure. In this section, we consider a memcached instance that might be configured with five possible memory types for the cache: disk, Flash, NVM_2, NVM_1, and DRAM.

Our evaluation employs traces from a cache augmented SQL system that processes social networking actions issued by the BG benchmark [5]. The mix of actions is 99% read and 1% write which is typical of social networking sites such as Facebook [8]. The trace corresponds to approximately 40 minutes of requests in which there are 1.1 million requests to 564 thousand key-value pairs. The total size of the key-value pairs requested is slightly less than 25 gigabytes. The cost of storing the entire database on the most expensive stash, DRAM, is just under $200.

When a key-value pair is absent from the cache, it must be recomputed by issuing one or more queries to the SQL system after every read which references it. The time for this computation is provided in the trace file. An update (write request) to a key-value pair is an

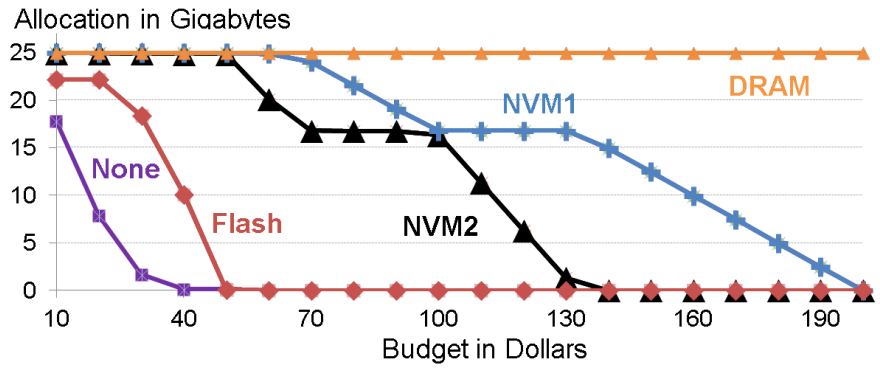


Figure 4.11: The optimal partition of the key-value pairs across the stashes as the budget varies. The cost of stash failures is not included in the evaluation of optimality.

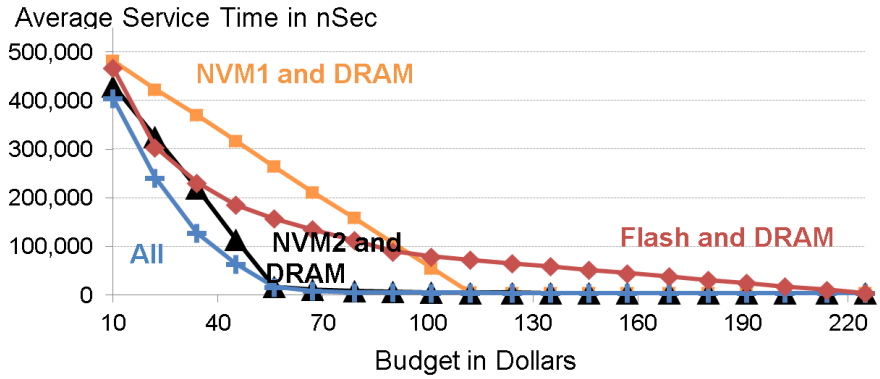


Figure 4.12: Configurations with two stashes using tiering.

update to the relational data used to compute that key-value pair. If the key-value pair is not stored on a stash, it does not need to be refilled (written to the cache).

With all budget scenarios considered, the optimal placement never assigned a key-value pair to disk. For some key-value pairs, the cost of reading the key-value pair from disk was more expensive than computing it directly from the database. Moreover, even for those key-value pairs which were more expensive to compute than to retrieve from disk, disk was not a viable option because the corresponding point was not on the convex hull of placement options. (See the right graph of Figure 4.3 that illustrates a similar scenario in which DRAM is not an option for a key-value pair.)

Figure 4.11 shows the optimal size of each stash under a tiering policy with all five memory types available as placement options. At each budget point, the vast majority of the key-value

pairs were stored in three consecutive stashes which means that it was generally more cost-effective to clear out key-value pairs from very slow stashes before investing in much faster space for the high-frequency items. Although not visible in the graph, under the optimal allocation, even for large budgets, there is approximately 8 MB of data that is not stored in the cache at all. These key-value pairs had one write request but no read requests over the course of the trace, so having those key-value pairs outside the cache reduced the average service time (although only slightly). The graph below shows the allocation in the scenario in which failure costs are not counted. When failures were counted, approximately 2/3 of the database was stored in NVM_1 instead of DRAM, even at the high end of the budget range. These key-value pairs were read only once during the entire trace in contrast with the key-value pairs stored in DRAM which were read on average about 4 times during the trace. Although it was slightly better to have the low frequency key-value pairs in NVM_1, the effect on the cost was almost negligible if they were included in DRAM instead. This illustrates that there can be many substantially different placements that are all close to the optimal in their average service time. These alternatives can be explored by limiting the set of placement options and comparing the average service time under the more restrictive scenario to the average service time in which all possibilities are allowed. The next set of experiments carry out this idea.

We evaluate the cache performance when the cache consists of DRAM in combination with different types of non-volatile memory. Figure 4.12 shows the performance of the cache as a function of budget for the scenario where DRAM is combined with one other storage option. The combination that does well over the broadest range of budgets is NVM_2 and DRAM. NVM_1 and DRAM do the best at the highest price range, but NVM_2 and DRAM is very close. In Figure 4.1, where the cache is limited to one stash, NVM_2 provides the best service time with budgets lower than $100.

Finally, we compare tiering and replication with a cache that includes NVM_2 and DRAM.

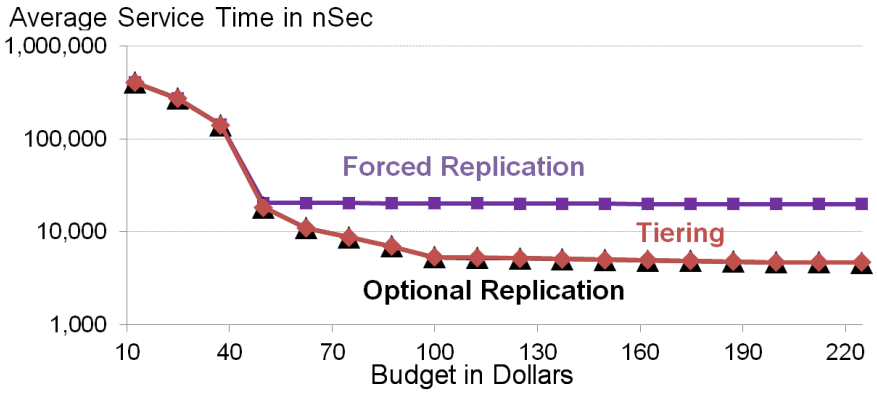


Figure 4.13: Tiering and replication policies with a cache that includes NVM_2 and DRAM. With optional replication, a key-value pair may reside in NVM_2, DRAM or both. With forced replication, every key-value pair in DRAM is also in NVM_2. With tiering, a key-value pair is assigned to one stash.

The graph in Figure 4.13 shows the data for tiering, optional replication and forced replication. Forced replication is significantly worse than the other two as the cost of updating key-value pairs in both stashes is expensive. The optimal placements under tiering and optional replication do differ slightly as the set of key-value pairs that are never updated are stored on both NVM_2 and DRAM under optional replication. However, the difference in performance is so negligible that the two lines cannot be distinguished in the graph. This data is more evidence that under this request distribution, the impact of memory failures is not significant and that tiering is a good choice for allocating key-value pairs to stashes.

4.5 Related work

An overview of the different types of memory including non-volatile memory is provided in [36]. This study motivates the development of both offline and online algorithms for managing storage for database applications, but it does not present specific algorithms.

Several studies have investigated a multi-level cache hierarchy in the context of distributed file servers [59, 77]. These studies observe that LRU may not work well for the intermediate

caches and present alternative online algorithms. The concept of *inclusive* and *exclusive* cache hierarchies is presented in [74]. Inclusive provides for duplication of disk blocks (similar to replication of data items) while exclusive de-duplicates blocks across the caches (tiering of data items). The approach in [74] is to extend LRU with a demote operation to implement an exclusive cache. None of these studies configure a cache by selecting the storage mediums that should participate as a stash in the multi-level hierarchy. Novel features of our approach include its optimality in serving as a measuring yardstick to evaluate alternative on-line algorithms and its consideration of failure rates.

A cache hierarchy consisting of PCM and NAND Flash is analyzed in [45]. While the focus of this study is on PCM and its viability as a stash for use as a host-side cache, it presents an offline algorithm to tier 1 GB extents (consisting of 4 Kilobyte disk pages) across a hierarchy composed of PCM, Flash, and disk. They evaluate the performance of a cache configuration in terms of I/Os per second (IOPS). Their method exhaustively searches all possible combinations of PCM, Flash, and disk to find the one that maximizes (IOPS)/$. (See [29] for additional details excluded here due to space limitations.) Our approach is a superset of theirs as it considers both tiering and replication with an arbitrary mix of storage medium and failure rates. Moreover, our method simultaneously optimizes cache configuration and placement subject to a budget constraint. Finally, our method is provably optimal and can be used as a measuring yardstick to evaluate heuristics.

4.6 Future work

An important next step is to evaluate online replacement policies in conjunction with the offline cache configuration method proposed here. This requires an extension of our model to consider the overhead of moving data items between the stashes. A simple approach may employ a static placement that is recomputed and updated periodically as the popularity

of different items vary over time [45]. Alternatively, a cache replacement policy might be a variant of online algorithms for two-level hierarchies such as CAMP [30].

Another important direction to pursue is to evaluate how robust a cache design is to changes in the workload characteristics. The use of past statistics is bound to be only an approximation of the workload in the future. Therefore, it is important to understand how well a particular cache design does as the set of data items grows over time or as the characteristics of the access pattern change and when it is appropriate to alter the cache configuration.

Chapter 5

The subset assignment problem for data placement in caches

5.1 Introduction

In the previous chapter, we looked at the problem of designing a key-value store from different types of storage media, each with a different set of characteristiics. We gave an algorithm which gave optimal amounts of each type of memory to purchase as well as an optimal placement of data items on subsets of those storage media in order to achieve the best expected time to service each request given a fixed budget and a certain set of workload characteristics. In this chapter, we examine the situation in which the types of storage media and their capacity are given as part of the problem input. Once again, the goal is to place data items in subsets of the memory banks so that the capacities of each memory bank are not exceeded and such that the total cost is minimized. By letting the placement options be subsets of memory banks rather than individual ones, we express the possibility of an item being replicated and stored on different memory banks. The solution to this problem could

be to readjust assignments periodically after the cache configuration is determined.

For each key-value pair p and every subset of the memory banks S, there is a cost associated with storing p on S. The costs do not necessarily exhibit any special properties such as monotonicity, although we do assume that they are non-negative. Chapter 4 gives a detailed description for how the memory parameters and request frequencies translate into costs and uses the model to study a closely related problem in which one is given a fixed budget as well as the price for the different types of memory. The goal is to determine the optimal amount of each type of memory to purchase as well as the optimal placement of key-value pairs to memory banks that minimizes expected service time subject to the overall budget constraint. We consider the situation in which the design of the cache is already determined in that there is a set of memory banks whose capacities are given as part of the problem input. The goal is to place each item on a subset of the memory banks so that the capacities of each memory bank is not exceeded and the total cost is minimized.

We give an abstract formulation of the *subset assignment problem* in which items (key-value pairs) are placed in bins (memory banks). In the context of key-value stores, the number of items n is very large (on the order of billions) and the number of bins is a small constant (e.g., in the range from 3 to 5). We show that there is an optimal solution for the linear programming relaxation in which at most d items are fractionally placed. The effect of not storing the fractionally placed items is negligible as memory banks are typically large enough to store many key-value pairs. Therefore, we focus on an efficient solution to the linear programming relaxation. Our goal is to find a solution whose dependence on n is as small as possible, even at the expense of exponential dependence on d. We give an algorithm that solves the LP relaxation in time $O(3^{d(d+1)}\,\mathrm{poly}(d)n\log(n)\log(nC)\log(Z))$, where Z is the maximum size of any object and C is the maximum cost for storing an item.

The subset assignment problem is a natural generalization of the multiple knapsack problem, which is the problem of choosing a subset of the items available and packing them into several

bins without exceeding their capacities in a way that maximizes the total profit. The book by Martello and Toth [52] and the more recent book by Kellerer et al. [43] both devote a chapter to multiple knapsack. The multiple knapsack problem is known to have a polynomial time approximation scheme [13].

If each item can only be stored on a single bin, the linear relaxation of the problem reduces to minimum cost flow on a bipartite graph [1]. Our algorithm is similar in structure to cycle canceling algorithms for min-cost flow. The focus on the scenario in which $n >> d$ is analogous to min cost flow on an imbalanced bipartite graph in which one side has many more nodes than the other. Our algorithm is also inspired by the concept of a bipush, which is central to the tighter analysis [34] and improvement [2] of algorithms for min-cost flow on the imbalanced bipartite graphs.

This chapter is organized as follows. Section 5.2 formally defines the subset assignment problem. Section 5.3 introduces the concepts of augmentations and basic feasible assignments and gives some of their properties. Section 5.4 describes an algorithm for the subset assignment problem and presents an analysis of its running time. Section 5.5 lists open problems.

5.2 Problem definition

There are n items and each item p has a given size(p). There are d bins $\mathcal{B} = \{b_1, \ldots, b_d\}$. Each bin b has a given capacity(b). An item can be replicated and placed on any subset of the bins $S \subseteq \mathcal{B}$. We call S a *placement option* for an item. Placing p on S has cost denoted by cost$(p, S) \geq 0$. A placement of items to bins is described by a set of $n \cdot 2^d$ variables $x(p, S) \geq 0$ with the constraint that for each p

$$\sum_S x(p, S) = \text{size}(p). \tag{5.1}$$

Also the capacity of each bin can not be exceeded, so for each b

$$\sum_{S \ni b} \sum_{p} x(p, S) \leq \text{capacity}(b). \tag{5.2}$$

The goal is to minimize

$$\sum_{p,S} \text{cost}(p, S) x(p, S),$$

subject to the condition that all $x(p, S) \geq 0$, (5.1) and (5.2) above.

The placement option $\varnothing$, corresponding to not placing an item in any of the bins, is an option for every p, so the problem always has a feasible solution. For each bin b, we will add an extra item p whose size is capacity(b). For each added p, $\text{cost}(p, \varnothing) = \text{cost}(p, \{b\}) = 0$. For all other $S \subseteq \mathcal{B}$, $\text{cost}(p, S) = \infty$. We assume that the pages are numbered so that the extra item for bin b_i is p_i. With the additional items, we can assume that every solution under consideration has every bin filled exactly to capacity since any extra space in b_i can be filled with p_i without changing the cost of the solution. So we require that for each b:

$$\sum_{S \ni b} \sum_{p} x(p, S) = \text{capacity}(b).$$

An assignment which satisfies the equality constraints on the bins is called *perfectly filled*.

5.3 Preliminaries

Our algorithm starts with a feasible, perfectly filled solution and improves the assignment in a series of small steps, called augmentations. The augmentations, a generalization of a negative cycle in min-cost flow, always maintain the condition that the current assignment is feasible and perfectly filled. In each iteration the algorithm finds an augmentation that approximates

the best possible augmentation in terms of the overall improvement in cost. An augmentation is a linear combination of moves in which mass is moved from $x(p, S)$ to $x(p, T)$ for some item p. Each move gives rise to a d-dimensional vector over $\{-1, 0, 1\}$ that denotes the net increase or decrease to each bin as a result of the move. We require that the linear combination of vectors for an augmentation equal $\vec{0}$ in order to maintain the condition that the bins are perfectly filled. The profile for an augmentation is the set of vectors corresponding to the moves in that augmentation. In order to find a good augmentation, we exhaustively search over all profiles (introducing a factor of $\exp(d(d+1))$ to the running time) and then find a good set of actual moves that correspond to each profile.

5.3.1 Augmentations

For $S \subseteq \mathcal{B}$, $\vec{S}$ is a d-dimensional vector whose i^{th} coordinate is 1 if $b_i \in S$ and is 0 otherwise. Let $\mathcal{V}$ be the set of all length d vectors over $\{-1, 0, 1\}$. A set $V \subseteq \mathcal{V}$ is said to be *minimally dependent* if V is linearly dependent and no proper subset of V is linearly dependent. If $V = \{\vec{v}_1, \ldots, \vec{v}_r\}$ is minimally dependent, then the values $\alpha_1, \ldots, \alpha_r$ such that $\sum_{i=1}^{r} \alpha_i \vec{v}_i = \vec{0}$ are unique up to a global constant factor. In order to make a unique vector $\vec{\alpha}$, we always maintain the convention that $\alpha_1 = 1$. A minimally dependent set V is said to be *positive* if the associated vector $\vec{\alpha} > \vec{0}$.

A *move* is defined by a triplet (p, S, T) that represents the possibility of moving mass from $x(p, S)$ to $x(p, T)$. The *profile* for a set of moves $\{(p_1, S_1, T_1), \ldots, (p_r, S_r, T_r)\}$ is the set of vectors $\{(\vec{T}_1 - \vec{S}_1), \ldots, (\vec{T}_r - \vec{S}_r)\}$. Note that the vector $\vec{T} - \vec{S}$ represents the net increase or decrease to each bin that results from moving one unit of mass from $x(p, S)$ to $x(p, T)$ for some p. A set of moves is called an *augmentation* if the set of vectors in its profile is minimally dependent and positive. Note that an augmentation contains at most $d + 1$ moves.

An augmentation $\mathcal{A} = \{(p_1, S_1, T_1), \ldots, (p_r, S_r, T_r)\}$ can be applied to a particular assignment

$\vec{x}$ if for every $i = 1, \ldots r$, $x(p_i, S_i) > 0$. Let $\vec{\alpha}$ be the unique vector of values such that $\alpha_1 = 1$ and $\sum_{j=1}^{r} \alpha_j(\vec{T_j} - \vec{S_j}) = \vec{0}$. If the augmentation is applied with magnitude a to $\vec{x}$, then for every $(p_j, S_j, T_j) \in \mathcal{A}$, $x(p_j, S_j)$ is replaced with $x(p_j, S_j) - a \cdot \alpha_j$ and $x(p_j, T_j)$ is replaced with $x(p_j, T_j) + a \cdot \alpha_j$. The cost vector for an augmentation is $\vec{c}$, where $c_j = \text{cost}(p_j, T_j) - \text{cost}(p_j, S_j)$. The cost associated with applying the augmentation with magnitude a is $a \cdot \vec{c} \cdot \vec{\alpha}$. Since the goal is to minimize the cost, we only apply augmentations whose cost is negative.

Let $\mathcal{S}(\mathcal{A})$ be the set of all pairs (p, S) such that for some i, $p = p_i$ and $S = S_i$. For each $(p, S) \in \mathcal{S}(\mathcal{A})$, define

$$\alpha(p, S) = \sum_{i: p_i = p, S_i = S} \alpha_i.$$

The maximum magnitude with which the augmentation $\mathcal{A}$ can be applied to $\vec{x}$ is

$$\min_{(p,S) \in \mathcal{S}(\mathcal{A})} \frac{x(p, S)}{\alpha(p, S)}.$$

The following lemma is analogous to the fact for flows that says there is always a cycle in the network representing the difference between two feasible flows. The proof is given in the appendix.

Lemma 5.1. Let $\vec{x}$ and $\vec{y}$ be two feasible, perfectly filled assignments to the same instance of the subset assignment problem. Then there is an augmentation that can be applied to $\vec{x}$ that consists only of moves of the form (p, S, T) where $x(p, S) > y(p, S)$ and $x(p, T) < y(p, T)$.

Proof. The first step is to come up with a linear combination of moves of the form (p, S, T) where $x(p, S) > y(p, S)$ and $x(p, T) < y(p, T)$ that transform $\vec{x}$ into $\vec{y}$. The profile for the set of moves must be linearly dependent because the net change to the load on each bin is 0. However, the resulting profile is not necessarily minimally dependent. The next step is to find a subset of those moves that can be applied to $\vec{x}$ whose profile is minimally dependent

and whose $\vec{\alpha}$ has positive coefficients. The following procedure accomplishes the first step:

Initialize $\vec{z} = \vec{x}$ and $j = 1$.

while $\vec{z} \neq \vec{y}$ **do**

Find a (p, S, T) such that $z(p, S) > y(p, S)$ and $z(p, T) < y(p, T)$.

$\beta_j = \min\{z(p, S) - y(p, S), y(p, T) - z(p, T)\}$

$(p_j, S_j, T_j) = (p, S, T)$

$z(p, S) = z(p, S) - \beta_j$

$z(p, T) = z(p, T) + \beta_j$

$j = j + 1$

Define $\mathcal{S}$ to be the set of pairs (p, S) such that $z(p, S) > y(p, S)$ and $\mathcal{T}$ to be the set of pairs (p, T) such that $z(p, T) < y(p, T)$. In each iteration, if (p, S, T) is the selected move, then either (p, S) drops out of $\mathcal{S}$ or (p, T) drops out of $\mathcal{T}$. Therefore, the process is finite and a move is never selected twice. Let t be the number of moves selected in the process. Applying each move (p_j, S_j, T_j) with magnitude β_j transforms $\vec{x}$ into $\vec{y}$. Since $\vec{x}$ and $\vec{y}$ are both perfectly filled, the net change to the load on each bin is 0:

$$\sum_{j=1}^{t} \beta_j(\vec{T_j} - \vec{S_j}) = \vec{0}.$$

In the second step, we adjust the linear combination of moves selected until its profile is minimally dependent. Let B be the set of indices j such that $\beta_j > 0$. Initially $B = \{1, \ldots, t\}$. Define $V_B = \{\vec{T_j} - \vec{S_j} : j \in B\}$. The following procedure accomplishes the second step:

while there is a proper subset of V_B that is linearly dependent **do**

Select a $\bar{B} \subseteq B$ such that $V_{\bar{B}}$ is minimally dependent.

Let $\{\gamma_j : j \in \bar{B}\}$ be the unique set of values such that: $\sum_{j \in \bar{B}} \gamma_j(\vec{T_j} - \vec{S_j}) = \vec{0}$, and $\min_{j \in \bar{B}} \gamma_j = 1$.

if $\gamma_j > 0$, for every $j \in \bar{B}$ **then**

return $\{(p_j, S_j, T_j) : j \in \bar{B}\}$

else

$c = \min_{j:\gamma_j<0} \frac{\beta_j}{-\alpha_j}$

for each $j \in \bar{B}$ **do**

$\beta_j = \beta_j + c\gamma_j$

return $\{(p_j, S_j, T_j) : j \in B\}$

Note that since $\sum_{j\in\bar{B}} \gamma_j(\vec{T_j} - \vec{S_j}) = \vec{0}$, adding a constant multiple of $\sum_{j\in\bar{B}} \gamma_j(\vec{T_j} - \vec{S_j})$ to $\sum_{j\in B} \beta_j(\vec{T_j} - \vec{S_j})$, maintains the condition that $\sum_{j\in B} \beta_j(\vec{T_j} - \vec{S_j}) = \vec{0}$.

The changes to $\vec{\beta}$ also maintain the condition that $\vec{\beta} \geq \vec{0}$. If $\gamma_j > 0$, then adding $c \cdot \gamma_j$ to β_j can only increase β_j. For j such that $\gamma_j < 0$, $c \leq -\beta_j/\gamma_j$, so

$$\beta_j + c\gamma_j \geq \beta_j + \left(\frac{-\beta_j}{\gamma_j}\right)\gamma_j = 0.$$

Since $c = -\beta_j/\gamma_j$ for some j, at least one β_j becomes 0 and the set B decrease by at least one index. Therefore V_B eventually becomes a minimally dependent set, and the moves corresponding to $j \in B$ satisfy the properties of being an augmentation. □

5.3.2 Basic feasible assignments

An item is said to be *fractionally assigned* if there are two subsets $S \neq S'$, such that $x(p, S) > 0$ and $x(p, S') > 0$. If items can only be assigned to single bins as in the standard assignment problem, then it follows from total unimodularity that the optimal solution is integral, assuming that all the input values are integers. Thus, if all the sizes are uniform and the capacities of the bins are an integer multiple of the item size, the optimal assignment will not have any fractionally assigned items. For the subset assignment problem, the optimal solution may not be integral, even if all the input values are integers. Here is an example

in which the item sizes and bin capacities are all 1, but an optimal solution must have fractionally assigned items: we have two items p and q and two bins b and c with assignment costs

$$\text{cost}(p, \varnothing) = 1, \quad \text{cost}(p, \{b, c\}) = 0, \qquad \text{cost}(q, \varnothing) = \text{cost}(q, \{b, c\}) = C,$$
$$\text{cost}(p, \{b\}) = \text{cost}(p, \{c\}) = C, \qquad \text{cost}(q, \{b\}) = \text{cost}(q, \{c\}) = 0,$$

where C is a large number. The optimal assignment is to equally distribute p over $\{b, c\}$ and $\varnothing$, and to equally distribute q over $\{b\}$ and $\{c\}$.

A basic feasible solution to the linear programming formulation of the subset assignment problem has at most $n + d$ non-zero variables, because the number of constraints is $n + d$. Since for every p, there is at least one S such that $x(p, S) > 0$ and $n >> d$, we know that most of the items will not be fractionally assigned in a basic feasible solution. The number of variables $x(p, S)$ such that $0 < x(p, S) < \text{size}(p)$ is at most $2d$ and therefore the number of fractionally assigned items is at most d. We rephrase the definition of a basic feasible solution in the language of the subset assignment problem and prove the same facts about the new definition.

Consider a feasible assignment $\vec{x}$. Let P_{frac} be the set of data items that are fractionally assigned. Let X_{frac} be the set of variables $x(p, S)$ such that $0 < x(p, S) < \text{size}(p)$. Let P_{int} be the set of items that are assigned to exactly one subset. That is $p \in P_{\text{int}}$ if $x(p, S) \in \{0, \text{size}(p)\}$ for all S.

Definition 5.2. For each $p \in P_{\text{frac}}$ select one S such that $x(p, S) > 0$. Call the selected set for p S_p. Let X be the set of variables $x(p, S)$ such that $S \neq S_p$ and $0 < x(p, S) < \text{size}(p)$. Let V be the set of vectors $\vec{S} - \vec{S}_p$ for each $x(p, S) \in X$. Then $\vec{x}$ is a *basic feasible assignment* (bfa) if and only if V is linearly independent.

Intuitively, the assignment $\vec{x}$ is a *bfa* if there is a unique way to assign the variables in X_{frac} after all the items in P_{int} have been assigned in such a way that all the bins are exactly at capacity.

Lemma 5.3. The condition of being a *bfa* does not depend on the choice of S_p for each $p \in P_{\text{frac}}$.

Proof. Let $p \in P_{\text{frac}}$ and let $\{S_1, \ldots, S_r\}$ be the subsets such that $x(p, S_j) > 0$. Suppose that S_p is chosen to be S_i. Select any two $S_j \neq S_k$. Since $(\vec{S}_j - \vec{S}_k) = (\vec{S}_j - \vec{S}_p) - (\vec{S}_k - \vec{S}_p)$, the space spanned by all $(\vec{S}_j - \vec{S}_k)$ for $S_j \neq S_k$ is equal to the space spanned by all $(\vec{S}_j - \vec{S}_p)$ for $S_j \neq S_p$. The space spanned by all $(\vec{S}_j - \vec{S}_k)$ for $S_j \neq S_k$ is independent of the choice for S_p. □

Lemma 5.4. If $\vec{x}$ is a *bfa*, then the number of variables in X_{frac} is at most $2d$ and the number of fractionally assigned items is at most d.

Proof. Since $|X| = |V|$, and V must be linearly independent for any *bfa*, it must be that if $\vec{x}$ is a *bfa*, then $|X| \leq d$. The set of fractionally assigned variables (X_{frac}) includes all the $x(p, S_p)$ for $p \in P_{\text{frac}}$ and X. For each $x(p, S_p)$, there is at least one variable in X. Therefore the number of variables such that $0 < x(p, S) < \text{size}(p)$ in any *bfa* is at most $2d$. □

5.3.3 The algorithm Restore

The algorithm RESTORE takes a feasible, perfectly filled assignment $\vec{x}$ and converts it to a *bfa* that is also perfectly filled. The cost of the resulting assignment is no larger than the cost of the input assignment. The number of iterations is bounded by the number of fractionally assigned variables in the input assignment.

Algorithm 5 PREPROCESS(d)

$\mathcal{P} = \varnothing$
for each subset $\{\vec{v}_1, \ldots, \vec{v}_d\}$ of $\mathcal{V} = \{-1, 0, 1\}^d$ **do**
    Let V be an ordered list whose i-th element is $\vec{v}_i$.
    Let A be the matrix whose i-th column is $\vec{v}_i$.
    Try to find A's inverse.
    if A is not invertible **then**
        Continue.
    for each $\vec{w}$ in $\mathcal{V}$ **do**
        $\vec{\alpha} = A^{-1}\vec{w}$
        for $i = 1, \ldots, d$ **do**
            if $\alpha_i < 0$ **then**
                Continue.
            if $\alpha_i = 0$ **then**
                Remove $\vec{v}_i$ from V.
                Remove α_i from $\vec{\alpha}$.
        Append $-\vec{w}$ to V.
        Append 1 to $\vec{\alpha}$.
        Sort $\mathcal{V}$ lexicographically.
        Reorder $\vec{\alpha}$ to match $\mathcal{V}$'s order.
        Rescale $\vec{\alpha}$ so that the first component is 1.
        Add $(V, \vec{\alpha})$ to $\mathcal{P}$.
return $\mathcal{P}$

Algorithm 6 RESTORE

for each $p \in P_{\text{frac}}$ **do**
  select one S such that $x(p, S) > 0$.
  Call the chosen set S_p.

Let X be the set of variables $x(p, S)$ such that $S \neq S_p$ and $0 < x(p, S) < \text{size}(p)$.
Order the variables in X: $\{x(p_1, S_1), \ldots, x(p_r, S_r)\}$
Let V be the set of vectors $\vec{S} - \vec{S}_p$ for each $x(p, S) \in X$.
while V is linearly dependent **do**
  Let $\beta_1, \ldots, \beta_r$ be such that $\sum_{i=1}^{r} \beta_i(\vec{S}_i - \vec{S}_{p_i}) = \vec{0}$.
  if $\sum_{i=1}^{r} \beta_i[\text{cost}(p_i, S_i) - \text{cost}(p_i, S_{p_i})] > 0$ **then**
    for $i = 1, \ldots, r$ **do**
      $\beta_i = -\beta_i$
  $a = \min\left\{\min_{i:\beta_i<0}\left\{\frac{x(p_i,S_i)}{-\beta_i}\right\}, \min_{i:\beta_i>0}\left\{\frac{x(p_i,S_{p_i})}{\beta_i}\right\}\right\}$
  for $i = 1, \ldots, r$ **do**
    $x(p_i, S_{p_i}) = x(p_i, S_{p_i}) - a \cdot \beta_i$
    $x(p_i, S_i) = x(p_i, S_i) + a \cdot \beta_i$
  if $x(p, S) \in X$ becomes 0 **then**
    remove $x(p, S)$ from X
  if $x(p, S_p)$ becomes 0 **then**
    Select an $x(p, S')$ from X and remove it from X.
    S_p becomes S'.
    Update vectors in V with new S_p.

The process RESTORE, shown in 5, takes an assignment $\vec{x}$ which may not be a *bfa* and restores it to an assignment which is a *bfa*. The process maintains the condition that the current assignment is feasible and perfectly filled. If the set V is linearly dependent, a linear combination of the moves (p, S_p, S) is chosen for each $x(p, S) \in X$ such that applying the linear combination of moves keeps the bins perfectly filled. Since p has some weight on S_p and some weight on S, all the moves can be applied in either the forward or reverse direction. (A negative coefficient denotes applying a move in the reverse direction.) We choose a direction for the linear combination of moves such that the cost does not increase. The combination of moves is applied until either $x(p, S)$ or $x(p, S_p)$ becomes 0 for one of the moves represented in V. Thus, the cost of the assignment does not increase and the number of fractionally assigned variables decreases by at least one. The process continues until V is linearly independent.

Lemma 5.5. There is an optimal solution that is also a *bfa*.

Proof. Start with an optimal assignment $\vec{x}$ which may not be a *bfa*. Apply RESTORE to $\vec{x}$. The resulting assignment is a *bfa*. Moreover, since the cost of $\vec{x}$ does not increase, $\vec{x}$ is still optimal. □

5.4 An algorithm for the subset assignment problem

The algorithm we present proceeds in a series of iterations. In each iteration, we apply an augmentation to the current assignment. Since the resulting assignment may no longer be a *bfa*, we then apply RESTORE to turn the solution back into a *bfa*.

Note that it is possible to find an augmentation that moves from a *bfa* to another *bfa* directly. This is essentially what the simplex algorithm does. However, not every augmentation results in a *bfa*. The augmentations that do result in a *bfa* must include the moves that shift mass between the fractionally assigned items. (These moves correspond to the vectors V

Algorithm 7 MainLoop

$x(p, S) = 0$, for all p and S.
$x(p_i, \{b_i\}) = \text{capacity}(b_i)$, for $i = 1, \ldots, d$. (Fill each bin with the "extra" items.)
$x(p_j, \varnothing) = \text{size}(p_j)$, for $j > d$. (All the "original" items start outside the bins.)
$\mathcal{P} = \textsc{Preprocess}(d)$ $\mathcal{A} = \textsc{FindAugmentation}(\vec{x})$
while $\mathcal{A} \neq \varnothing$ **do**
    Apply $\mathcal{A}$ to $\vec{x}$ with the largest possible magnitude
    $\textsc{Restore}(\vec{x})$. (Transform $\vec{x}$ into a *bfa*.)
    $\mathcal{A} = \textsc{FindAugmentation}(\vec{x})$

described in the definition of a *bfa*). Restricting the augmentation in this way may result in a sub-optimal augmentation. For example, those augmentations could require decreasing a variable that is already very small in which case the augmentation can not be applied with very large magnitude. We therefore allow the algorithm to select from the set of all augmentations in order to get as much benefit as possible and then move the assignment to a *bfa*.

5.4.1 Finding an augmentation that is close to the best possible

The first step is a preprocessing step in which every possible augmentation profile is generated. This consists of generating every minimally dependent subset V of $\mathcal{V}$ and its associated $\vec{\alpha}$. Preprocess (shown in the Appendix) examines each basis made up of vectors from $\mathcal{V}$, in combination with an extra vector $\vec{w}$. It determines the basis vectors that $\vec{w}$ depends on. These basis vectors, together with $\vec{w}$, form a minimally dependent subset of $\mathcal{V}$. To handle the fact that the same minimal subset may be found multiple times, the set data structure storing the minimally dependent subsets uses a canonical representation of the subsets, thus avoiding duplicate storage. Finding the inverse of a matrix using Gaussian elimination is $O(d^3)$ and matrix multiplication is $O(d^2)$. So the running time of Preprocess is $O(\binom{3^d}{d}(d^3 + 3^d(d^2 + d))) = O(3^{d^2+d}d^2)$. Moreover, the size of the returned set of minimally dependent subsets is $O(3^{d(d+1)})$.

Given an augmentation profile $V = \{\vec{v}_1, \ldots, \vec{v}_r\}$, the goal is to find an augmentation whose profile matches V and can be applied with a magnitude that gives close to the best possible improvement. For each vector $\vec{v} \in \mathcal{V}$, we maintain a data structure with every move (p, S, T) such that $\vec{T} - \vec{S} = \vec{v}$ and $x(p, S) > 0$. We will call the set of all such moves $Moves(\vec{v})$. The data structure should be able to answer queries of the form: given x_0, find the move (p, S, T) such that $\text{cost}(p, T) - \text{cost}(p, S)$ is minimized subject to the condition that $x(p, S) \geq x_0$. These kind of queries can be handled by an augmented binary search tree in logarithmic time [14].

For a given *bfa* $\vec{x}$ and augmentation $\mathcal{A}$, one can calculate the maximum possible magnitude a with which $\mathcal{A}$ can be applied to $\vec{x}$. We will make use of upper and lower bounds for the value a for any augmentation and *bfa* combination. Call these values $a_{\max}$ and $a_{\min}$. Round $a_{\min}$ down so that $a_{\max}/a_{\min}$ is a power of 2. The while loop in procedure FindAugmentation runs for $\log(a_{\max}/a_{\min})$ iterations.

Algorithm 8 FindAugmentation($\vec{x}$)

```
BestCost = 0
A = ∅
for each augmentation profile V = {v_1, ..., v_r} and vector α⃗ do
    a = a_max/2
    while a ≥ a_min do
        for i = 1, ..., r do
            Let (p_i, S_i, T_i) be the move with the smallest cost among moves in Moves(v_i)
such that x(p_i, S_i) ≥ a · α_i.
            c_i = cost(p_i, T_i) − cost(p_i, S_i)
        CurrentCost = Σ_{i=1}^{r} a · c_i · α_i
        if CurrentCost < BestCost then
            BestCost = CurrentCost
            A = {(p_1, S_1, T_1), ..., (p_r, S_r, T_r)}
        a = a/2
return A
```

For an augmentation $\mathcal{A}$ that can be applied to assignment $\vec{x}$ with magnitude a, the total change in cost is denoted by $\text{cost}(\mathcal{A}, \vec{x}, a)$. Recall that since we are minimizing cost we will only apply an augmentation if the total change in cost is less than 0.

Lemma 5.6. Let $\mathcal{A}_1$ be the augmentation returned by FINDAUGMENTATION($\vec{x}$). Let a_1 be the maximum magnitude with which $\mathcal{A}_1$ can be applied to $\vec{x}$. Let $\mathcal{A}_2$ be any other augmentation and a_2 the maximum magnitude with which $\mathcal{A}_2$ can be applied to $\vec{x}$. Then $2d \cdot \text{cost}(\mathcal{A}_1, \vec{x}, a_1) \leq \text{cost}(\mathcal{A}_2, \vec{x}, a_2)$.

Proof. Let V_2 be the profile for $\mathcal{A}_2$. Let $\vec{\alpha}$ be the vector associated with the profile V_2. Let $\bar{a}$ be the value of the form $a_{\max}/2^j$ such that $2\bar{a} > a_2 \geq \bar{a}$. There is an iteration inside the while loop of FINDAUGMENTATION($\vec{x}$) in which the augmentation profile is V_2 and the value for a is $\bar{a}$. The augmentation constructed in this iteration will be called V_3. The moves in $\mathcal{A}_2$ are $\{(p_1^{(2)}, S_1^{(2)}, T_1^{(2)}), \ldots, (p_r^{(2)}, S_r^{(2)}, T_r^{(2)})\}$. The moves in $\mathcal{A}_3$ are $\{(p_1^{(3)}, S_1^{(3)}, T_1^{(3)}), \ldots, (p_r^{(3)}, S_r^{(3)}, T_r^{(3)})\}$. Note that since V_2 can be applied to $\vec{x}$ with magnitude a_2, it must be the case that for $i = 1, \ldots, r$, $x(p_i^{(2)}, S_i^{(2)}) \geq \alpha_i a_2$ because applying the moves involves removing $\alpha_i a_2$ from $x(p_i^{(2)}, S_i^{(2)})$. Since $a_2 \geq \bar{a}$, $x(p_i^{(2)}, S_i^{(2)}) \geq \alpha_i \bar{a}$. The move $(p_i^{(3)}, S_i^{(3)}, T_i^{(3)})$ is chosen to be the move with minimum cost such that $x(p_i^{(3)}, S_i^{(3)}) \geq \alpha_i \bar{a}$. Therefore the cost of $(p_i^{(3)}, S_i^{(3)}, T_i^{(3)})$ is at most the cost of $(p_i^{(2)}, S_i^{(2)}, T_i^{(2)})$. The value of the variable *CurrentCost* for that iteration is

$$\begin{aligned}
\mathit{CurrentCost}_3 &= \bar{a} \sum_{i=1}^{r} \alpha_i \left[\text{cost}(p_i^{(3)}, T_i^{(3)}) - \text{cost}(p_i^{(2)}, S_i^{(3)}) \right] \\
&\leq \bar{a} \sum_{i=1}^{r} \alpha_i \left[\text{cost}(p_i^{(2)}, T_i^{(2)}) - \text{cost}(p_i^{(2)}, S_i^{(2)}) \right] \\
&\leq \frac{a_2}{2} \sum_{i=1}^{r} \alpha_i \left[\text{cost}(p_i^{(2)}, T_i^{(2)} - \text{cost}(p_i^{(2)}, S_i^{(2)}) \right] = \frac{1}{2} \text{cost}(\mathcal{A}_2, \vec{x}, a_2)
\end{aligned}$$

Let $\mathit{CurrentCost}_1$ be the value of the variable *CurrentCost* and a' the value of the variable a during the iteration in which the augmentation $\mathcal{A}_1$ is considered. Since $\mathcal{A}_1$ was selected by FINDAUGMENTATION, $\mathit{CurrentCost}_1 \leq \mathit{CurrentCost}_3$. It remains to show that the maximum magnitude with which $\mathcal{A}_1$ can be applied is at least a'/d and therefore the actual change in

cost at most $CurrentCost_1/d$.

Let V_1 be the profile for $\mathcal{A}_1$ and let $\vec{\beta}$ be the vector associated with profile V_1. Since we are now only referring to one augmentation, we omit the subscripts and call the moves in $\mathcal{A} = \{(p_1, S_1, T_1), \ldots, (p_r, S_r, T_r)\}$. We are guaranteed by the selection of the move (p_i, S_i, T_i) that for every i, $x(p_i, S_i)/\beta_i \geq a'$. Define $\beta^{\text{sum}}_{p,S}$ is the sum over all β_i such that $p_i = p$ and $S_i = S$. Define $\beta^{\text{max}}_{p,S}$ is the maximum over all β_i such that $p_i = p$ and $S_i = S$. The value of a_1, the maximum value with which $\mathcal{A}$ can be applied, is equal to $x(p, S)/\beta_{p,S}$ for some pair (p, S). We have

$$a_1 = \frac{x(p,S)}{\beta^{\text{sum}}_{p,S}} \geq \frac{x(p,S)}{d \cdot \beta^{\text{max}}_{p,S}} \geq \frac{a'}{d}.$$

□

5.4.2 Number of iterations of the main loop

The procedure FindAugmentation takes a *bfa* $\vec{x}$ and returns an augmentation that reduces the cost of the current solution by an amount which is within $\Omega(1/d)$ of the best possible augmentation that can be applied to $\vec{x}$. In order to bound the number iterations of the main loop, we need to show that there always is a good augmentation that can be applied to $\vec{x}$ that moves it towards an optimal solution. The idea is that for any two assignments $\vec{x}$ and $\vec{y}$, $\vec{x}$ can be transformed into $\vec{y}$ by applying a sequence of augmentations. Each augmentation decreases the number of variables in which $\vec{x}$ and $\vec{y}$ differ by one. Since the number of non-zero variables in any *bfa* is at most $n + d$, there are at most $2(n + d)$ augmentations in the sequence. Thus, if the difference in cost between $\vec{y}$ and $\vec{x}$ is Δ, one of the augmentations will decrease the cost by at least $\Delta/(2(n + d)$. The idea is analogous to the partitioning the difference between two min cost flows into a set of disjoint cycles. Some additional work is required to establish that the chosen augmentation can be applied directly to $\vec{x}$ with sufficient

magnitude.

Lemma 5.7. Let $\vec{x}$ be a *bfa* for an instance of the subset assignment problem and let Δ be the difference in the objective function between $\vec{x}$ and the optimal solution. Then there is an augmentation $\mathcal{A}$ such that when $\mathcal{A}$ is applied to $\vec{x}$ with the maximum possible magnitude, the cost drops by at least $\Delta/2(n+d)$.

Proof. Let $\vec{y}$ be an optimal solution that is also a *bfa*. We define a sequence of assignments $\vec{z}_0, \vec{z}_1, \ldots, \vec{z}_t$. We start with $\vec{z}_0 = \vec{x}$ and describe how to obtain $\vec{z}_{j+1}$ from $\vec{z}_j$. Let $\mathcal{S}_j$ be the set of pairs (p, S) such that $z_j(p, S) > y(p, S)$. Let $\mathcal{T}_j$ be the set of pairs (p, T) such that $z_j(p, T) < y(p, T)$. We know from Lemma 5.1 that there is an augmentation $\mathcal{A}_j$ that can be applied to $\vec{z}_j$ such that all the moves are of the form (p, S, T), where $(p, S) \in \mathcal{S}_j$ and $(p, T) \in \mathcal{T}_j$. Apply augmentation $\mathcal{A}_j$ to $\vec{z}_j$ with magnitude a_j to get $\vec{z}_{j+1}$, where a_j is the largest possible magnitude with which $\mathcal{A}_j$ can be applied such that $z_{j+1}(p, S) \geq y(p, S)$ for every $(p, S) \in \mathcal{S}_j$ and $z_{j+1}(p, T) \leq y(p, T)$ for every $(p, T) \in \mathcal{T}_j$. Note that after the augmentation is applied, it must be true that for some $(p, S) \in \mathcal{S}_j$, $z_{j+1}(p, S) = y(p, S)$ or for some $(p, T) \in \mathcal{T}_j$, $z_{j+1}(p, T) = y(p, T)$. Therefore $\mathcal{S}_{j+1} \cup \mathcal{T}_{j+1}$ is a proper subset of $\mathcal{S}_j \cup \mathcal{T}_j$. Continue the process until $\mathcal{S}_t \cup \mathcal{T}_t = \varnothing$ which means that $\vec{z}_t = \vec{y}$.

Since any *bfa* has at most $(n+d)$ non-zero variables, $|\mathcal{S}_0 \cup \mathcal{T}_0| \leq 2(n+d)$ and therefore $t \leq 2(n+d)$. The t augmentations cause the cost of the assignment to drop by Δ, so there is at least one augmentation that causes the cost to drop by at least $\Delta/2(n+d)$. Suppose that the largest drop in cost happens when $\mathcal{A}_j$ is applied with magnitude a_j to $\vec{z}_j$. We need to establish that $\mathcal{A}_j$ can be applied to $\vec{z}_0$ with magnitude at least a_j.

Consider a pair $(\bar{p}, \bar{S})$ such that $z_j(\bar{p}, \bar{S})$ decreases when $\mathcal{A}_j$ is applied to $\vec{z}_j$. Then $(\bar{p}, \bar{S}) \in \mathcal{S}_j$. Furthermore since each $\mathcal{S}_j \subseteq \mathcal{S}_{j-1} \subseteq \cdots \subseteq \mathcal{S}_0$, then $(\bar{p}, \bar{S}) \in \mathcal{S}_i$ for any i in the range from 0 to j. The augmentations $\mathcal{A}_0, \ldots, \mathcal{A}_j$ can only take mass off of $z_i(\bar{p}, \bar{S})$ or leave it the same. Therefore $z_0(\bar{p}, \bar{S}) \geq z_j(\bar{p}, \bar{S})$ for any pair $(\bar{p}, \bar{S})$ such that $\mathcal{A}_j$ causes $z_j(\bar{p}, \bar{S})$ to decrease.

Thus, if $\mathcal{A}_j$ can be applied to $\vec{z}_j$ with magnitude a_j, then $\mathcal{A}_j$ can also be applied to $\vec{z}_0$ with magnitude at least a_j. □

In order to bound the number of iterations in the main loop, we need to know the smallest difference in cost between two assignments that have different cost. If none of the items are fractionally assigned, all of the variables have integer values and the cost of an assignment is an integer. But if there are fractional assignments, we must bound the granularity of a solution. First, we need a technical lemma.

Lemma 5.8. If A is an invertible $d \times d$ matrix with entries in $\{-1, 0, 1\}$ and $\vec{b}$ is a d-vector with integer entries, then there is an integer $\ell \leq d^{d/2}$ such that the solution $\vec{x}$ to $A\vec{x} = \vec{b}$ has entries of the form k/ℓ where k is an integer. Moreover, if $\vec{b}$ also has entries in $\{-1, 0, 1\}$, then the entries of x are at most equal to d.

Proof. As a consequence of the Laplace expansion of the determinant, the inverse of A is equal to $\text{adj}(A)/\det(A)$, where $\text{adj}(A)$ is the adjugate matrix of A or transpose of the cofactor matrix of A. Since A has integer entries, so does its adjugate. This implies that the elements of x are all of the form $k/\det(A)$ for some integer k. But by Hadamard's inequality, the determinant of A is bounded by $d^{d/2}$. So this proves the first statement.

Now if the entries of $\vec{b}$ are in $\{-1, 0, 1\}$, then $k \leq d$. Since A is invertible, its determinant is non-zero. Since its entries are integers, the absolute value of its determinant is at least 1. This proves the second statement. □

The following bound comes from the fact that the fractionally assigned values are the solution to a matrix equation with a $d \times d$ matrix over $\{-1, 0, 1\}$.

Lemma 5.9. If $\vec{x}$ is a *bfa*, then there is an integer $\ell \leq d^{d/2}$ such that every $x(p, S) = k/\ell$ for some integer k.

Proof. As in the definition of a *bfa*, let P_{int} denote the items that are integrally and P_{frac} those that are partially assigned. Also, let S_p be the subset or one of the subsets to which p is assigned, let $F = \{(p, S) : x_{p,S} > 0 \text{ and } S \neq S_p\}$, and let $\vec{c}$ be the vector of bin capacities. Since $\vec{x}$ is perfectly filled, we have

$$\sum_{p \in P_{\text{int}}} x(p, S_p)\vec{S}_p + \sum_{p \in P_{\text{frac}}} x(p, S_p)\vec{S}_p + \sum_{(p,S) \in F} x(p, S)\vec{S} = \vec{c}.$$

Now $x(p, S_p) = \text{size}(p) - \sum_{S \neq S_p} x(p, S)$ for all p, and in particular $x(p, S_p) = \text{size}(p)$ for $p \in P_{\text{int}}$. So we have

$$\sum_{p} \text{size}(p)\vec{S}_p + \sum_{(p,S) \in F} x(p, S)(\vec{S} - \vec{S}_p) = \vec{c}.$$

By definition of a *bfa*, the vectors $\vec{S} - \vec{S}_p$ are linearly independent. Therefore, if we extend this set to a basis of $\{-1, 0, 1\}$-vectors, we can view this equation as the matrix equation

$$A\vec{x} = \vec{c} - \sum_{p} \text{size}(p)\vec{S}_p$$

where A's columns are the basis vectors. Since A has entries in $\{-1, 0, 1\}$ and the right hand vector has integer entries, an application of Lemma 5.8 yields the result. □

Lemma 5.10. The number of iterations of the main loop is $O(nd^2 \log(dnC))$.

Proof. The cost of the inital assignment is at most nC, where $C = \max_{p,S} \text{cost}(p, S)$. Since the costs are non-negative, the difference in cost between the initial assignment and an optimal assignment is at most nC.

By Lemma 5.9, for any *bfa* $\vec{x}$, every $x(p, S)$ is an integer multiple of some $1/\ell$ where ℓ is an integer bounded by $d^{d/2}$. Since the costs are integers, the cost of $\vec{x}$ is also an integer multiple of $1/\ell$. Consider two *bfa*'s, $\vec{x}$ and $\vec{y}$ with different costs. The cost of $\vec{x}$ is a multiple of $1/\ell$

and the cost of $\vec{y}$ is a multiple of $1/\ell'$, where ℓ and ℓ' are both integers bounded by $d^{d/2}$. If $\ell = \ell'$, then the difference in costs between $\vec{x}$ and $\vec{y}$ is at least $1/d^{d/2}$. If $\ell > \ell'$, the difference in cost is at least

$$\frac{1}{\ell'} - \frac{1}{\ell} = \frac{\ell - \ell'}{\ell\ell'} \geq \frac{1}{d^d}.$$

Lemmas 5.7 indicates that if the difference in cost between the current assignment and the optimal assignment is Δ, there is an augmentation that reduces the cost by at least $\Delta/2(n+d)$ and Lemma 5.6 indicates that the augmentation returned by FindAugmentation reduces the cost by at least $1/2d$ times the best possible. Therefore, each iteration reduces the difference in cost between the current assignment and the optimal assignment by at least a factor of $1 - 1/(4d(n+d))$. The number of iterations is the smallest t such that

$$nC\left(1 - \frac{1}{4d(n+d)}\right)^t < \frac{1}{d^d},$$

which is $O(nd^2 \log(dnC))$. □

5.4.3 Analysis of the running time

The running time of Preprocess is dominated by the running time of the main loop, so we just analyze the running time of the main loop. To bound the size of the augmented binary search trees $Moves(\vec{v})$, observe that for each S, there is at most one T such that $\vec{T} - \vec{S} = \vec{v}$. Therefore, the number of moves (p, S, T) that can be stored in a single tree is $O(2^d n)$. Updates are handled in logarithmic time, so the time per update to an entry in one of the trees is $O(d \log n)$. Every time a variable $x(p, S)$ changes, there are 2^d subsets T such that the move (p, S, T) must be updated. In each iteration of the main loop there are $O(d)$ variable changes, resulting in a total update time of $O(d^2 2^d \log n)$.

By lemma 5.4, the *bfa* at the beginning of an iteration has at most $2d$ fractionally assigned variables. An augmentation consists of at most $d+1$ moves and therefore changes the value of at most $2(d+1)$ variables. Thus, the input to RESTORE is an assignment with $O(d)$ fractionally assigned variables. Each iteration of RESTORE reduces the number of fractionally assigned variables by at least one. Therefore, the number of iterations of RESTORE is bounded by $O(d)$ and the total time spent in RESTORE during an iteration is poly(d).

The inner loop of FINDAUGMENTATION requires $O(d)$ queries to one of the augmented binary search trees resulting in $O(d^2 \log n)$ time for each iteration of the inner loop. The number of times the inner loop is executed is $\log(a_{\max}/a_{\min})$ times the number of augmentation profiles which is bounded by $3^{d(d+1)}$. Therefore the running time of FINDAUGMENTATION dominates the running time of an iteration of the main loop which is $O(3^{d(d+1)}d^2 \log n \log(a_{\max}/a_{\min})$. By Lemma 5.10, the number of iterations of the main loop is $O(nd^2 \log(dnC))$, resulting in a total running time of

$$O(3^{d(d+1)}d^4 n \log n(\log n + \log C) \log(a_{\max}/a_{\min})).$$

It remains to bound $a_{\max}/a_{\min}$:

Lemma 5.11. The values of a are bounded above by $a_{\max} = d^{d/2}Z$ and below by $a_{\min} = 1/d^{d/2+1}$, where $Z = \max_p \text{size}(p)$.

Proof. Recall that the entries of $\vec{\alpha}$ were obtained as the solution to the equation $A\vec{\alpha} = \vec{w}$ where the columns of A and the vector $\vec{w}$ have entries in $\{-1, 0, 1\}$. By Lemma 5.8, $1/d^{d/2} \leq \alpha_i \leq d$.

An upper bound on a is the ratio of the maximum possible value for $x(p, S)$ over the minimum possible value for α_i. The highest value that $x(p, S)$ can achieve is $Z = \max_p \text{size}(p)$ because of (5.1). So let $a_{\max} = d^{d/2}Z$.

Similarly a lower bound on a is the ratio of the minimum possible $x(p, S)$ over the maximum

possible value for α_i. By Lemma 5.9, $x(p, S)$ is least $1/d^{d/2}$. So we can take $a_{\min}$ to be $1/d^{d/2+1}$. □

Putting it all together, we get that $\log(a_{\max}/a_{\min})$ is $O(d^2 \log d \log Z)$ and the total running time is bounded by $O(3^{d(d+1)} \operatorname{poly}(d) n \log(n) \log(nC) \log(Z))$.

5.5 Future work

In this chapter, we gave an algorithm for the linear programming relaxation of the subset assignment problem, which has complexity $O(3^{d(d+1)} \operatorname{poly}(d) n \log(n) \log(nC) \log(Z))$. Is there a faster algorithm that minimizes the dependency on d? Our algorithm is a generalization of the maximum cycle canceling algorithm for the minimum-cost flow problem. However, there are other, more efficient algorithms for this problem, including ones that take advantage of the bipartite structure inherent in assignment problems. Is there a more efficient algorithm that is a generalization of, say, the cost-scaling algorithm? Finally, the subset assignment problem was motivated by the problem of data placement in the memory banks of a multi-level cache. The next step is to use the algorithm presented here to perform a trace-driven simulation to evaluate how robust a cache configuration is to changes in workload characteristics.

Bibliography

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1 edition, 2 1993.

[2] R. K. Ahuja, J. B. Orlin, C. Stein, and R. E. Tarjan. Improved algorithms for bipartite network flow. *SIAM J. Comput.*, 23(5):906–933, 1994.

[3] M. Altinel, Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, B. G. Lindsay, H. Woo, and L. Brown. DBCache: Database Caching for Web Application Servers. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 612–612, New York, NY, USA, 2002. ACM.

[4] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. LinkBench: a Database Benchmark Based on the Facebook Social Graph. In *SIGMOD*, pages 1185–1196. ACM, 2013.

[5] S. Barahmand and S. Ghandeharizadeh. BG: a benchmark to evaluate interactive social networking actions. In *CIDR*, January 2013.

[6] S. Barahmand, S. Ghandeharizadeh, and J. Yap. A comparison of two physical data designs for interactive social networking actions. In *Proceedings of the 22Nd ACM International Conference on Conference on Information & Knowledge Management*, CIKM '13, pages 949–958, 2013.

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

[8] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX ATC 13*, pages 49–60, San Jose, CA, 2013.

[9] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer. Mercury: Host-side Flash Caching for the Data Center. In *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2012.

[10] R. Cáceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich. Web Proxy Caching: The Devil is in the Details. *SIGMETRICS Perform. Eval. Rev.*, 26(3):11–15, Dec. 1998.

[11] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[12] P. R. Center. World Wide Web Timeline. `http://www.pewinternet.org/2014/03/11/world-wide-web-timeline/`, 2014. Last visited on 11/8/2015.

[13] C. Chekuri and S. Khanna. A PTAS for the Multiple Knapsack Problem. In *Proc. of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 213–222. ACM, 2000.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, third edition, 2009.

[15] Dell. Dell Fluid Cache for Storage Area Networks. `http://www.dell.com/learn/us/en/04/solutions/fluid-cache-san`, 2014.

[16] P. J. Denning. The Locality Principle. *Commun. ACM*, 48(7):19–24, July 2005.

[17] P. J. Denning and S. C. Schwartz. Properties of the Working-set Model. *Commun. ACM*, 15(3):191–198, Mar. 1972.

[18] M. Dyer. An $O(n)$ Algorithm for the Multiple-Choice Knapsack Linear Program. *Mathematical Programming*, 29(1):57–63, 1984.

[19] L. Fan, P. Cao, W. Lin, and Q. Jacobson. Web Prefetching Between Low-bandwidth Clients and Proxies: Potential and Performance. *SIGMETRICS Perform. Eval. Rev.*, 27(1):178–187, May 1999.

[20] A. Feldmann, R. Caceres, F. Douglis, G. Glass, and M. Rabinovich. Performance of Web Proxy Caching in Heterogeneous Bandwidth Environments. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 107–116 vol.1, Mar 1999.

[21] M. Fink. Beyond DRAM and Flash, Part 2: New Memory Technology for the Data Deluge, HP Next. `http://www8.hp.com/hpnext/posts/beyond-dram-and-flash-part-2-new-memory-technology-data-deluge.VCb6VRbCfE8`, 2014.

[22] B. Fitzpatrick. changelog: livejournal. `http://changelog.livejournal.com/637455.html`, 2003. Last visited on 11/8/215.

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

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

[25] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

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

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

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

[29] S. Ghandeharizadeh, S. Irani, and J. Lam. Memory Hierarchy Design for Caching Middleware in the Age of NVM. Technical Report 2015-01, USC Database Laboratory, 2015.

[30] S. Ghandeharizadeh, S. Irani, J. Lam, and J. Yap. CAMP: a Cost Adaptive Multi-Queue Eviction Policy. In *Middleware 2014*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014.

[31] S. Ghandeharizadeh, J. Menon, G. Kotzur, S. Sen, and G. Chawla. Host Side Caching: Solutions and Opportunities. Technical Report 2015-01, USC Database Laboratory, 2015.

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

[33] S. Ghandeharizadeh, J. Yap, and H. Nguyen. Strong Consistency in Cache Augmented SQL Systems. *Middleware*, December 2014.

[34] D. Gusfield, C. Martel, and D. Fernández-Baca. Fast Algorithms for Bipartite Network Flow. *SIAM J. Comput.*, 16(2):237–251, 1987.

[35] D. A. Holland, E. Angelino, G. Wald, and M. I. Seltzer. Flash Caching on the Storage Client. In *USENIXATC*, 2013.

[36] H. Hunter, L. Lastras-Montano, and B. Bhattacharjee. Adapting Server Systems for New Memory Technologies. *IEEE Computer*, 47(9):78–84, Sept 2014.

[37] S. Irani, J. Lam, and S. Ghandeharizadeh. Cache Replacement with Memory Allocation. *ALENEX*, 2015.

[38] N. N. G. Jakob Nielsen. The Need for Speed. `http://www.nngroup.com/articles/the-need-for-speed/`, 1997. Last visited on 11/8/2015.

[39] S. Jiang and X. Zhang. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. *SIGMETRICS Perform. Eval. Rev.*, 30(1):31–42, June 2002.

[40] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 439–450. Morgan Kaufmann, 1994.

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

[42] S. Kavalanekar, B. L. Worthington, Q. Zhang, and V. Sharda. Characterization of Storage Workload Traces from Production Windows Servers. In *4th International Symposium on Workload Characterization IISWC, Seattle, Washington, USA, September*, pages 119–128, 2008.

[43] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.

[44] H. Kim, I. Koltsidas, N. Ioannou, S. Seshadri, P. Muench, C. Dickey, and L. Chiu. Flash-Conscious Cache Population for Enterprise Database Workloads. In *Fifth International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2014.

[45] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu. Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014.

[46] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. In *8th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 23-26, 2010*, pages 211–224, 2010.

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

[48] P.-A. Larson, J. Goldstein, and J. Zhou. MTCache: transparent mid-tier database caching in SQL server. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 177–188, March 2004.

[49] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A Spectrum of Policies That Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Comput.*, 50(12):1352–1361, Dec. 2001.

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

[51] D. Liu, J. Tai, J. Lo, N. Mi, and X. Zhu. VFRM: Flash Resource Manager in VMware ESX Server. In *IEEE Network Operations and Management Symposium*, 2014.

[52] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., 1990.

[53] Y. Matias, S. C. Sahinalp, and N. E. Young. Performance Evaluation of Approximate Priority Queues. In *Proceedings of Fifth DIMACS Implementation Challenge*, 1996.

[54] N. Megiddo and D. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *In Proceedings of the 2003 Conference on File and Storage Technologies (FAST*, pages 115–130, 2003.

[55] N. Megiddo and D. Modha. Outperforming LRU with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, April 2004.

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

[57] D. Mituzas. Flashcache at Facebook: From 2010 to 2013 and Beyond. `https://www.facebook.com/notes/facebook-engineering/flashcache-at-facebook-from-2010-to-2013-and-beyond/10151725297413920`, 2010.

[58] C. Muller, L. Courtade, C. Turquat, L. Goux, and D. Wouters. Reliability of Three-Dimensional Ferroelectric Capacitor Memory-Like Arrays Simultaneoulsy Submitted to X-Rays and Electrical Stresses. In *Non-Volatile Memory Technology Symposium*, 2006.

[59] D. Muntz and P. Honeyman. Multi-level Caching in Distributed File Systems -or- Your cache ain't nuthin' but trash. In *In Proceedings of the Winter 1992 USENIX*, pages 305–313, 1992.

[60] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association.

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

[62] E. J. O'Neil, P. E. O'Neil, and G. Weikum. An Optimality Proof of the LRU-K Page Replacement Algorithm. *J. ACM*, 46(1):92–112, Jan. 1999.

[63] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional Consistency and Automatic Management in an Application Data Cache. In *OSDI*. USENIX, October 2010.

[64] M. Rajashekhar and Y. Yue. Twemcache 2.6.0. `https://twitter.com/twemcache`, 2012–2013.

[65] M. Reddy and G. P. Fletcher. An adaptive mechanism for web browser cache management. *IEEE Internet Computing*, 2(1):78–81, Jan. 1998.

[66] S. Daniel and S. Jafri. Using NetApp Flash Cache (PAM II) in Online Transaction Processing. NetApp White Paper, 2009.

[67] P. Sinha and A. A. Zoltners. The Multiple-Choice Knapsack Problem. *Operations Research*, 27(3):pp. 503–515, 1979.

[68] W. Stearns and K. Overstreet. Bcache: Caching Beyond Just RAM. `https://lwn.net/Articles/394672/,http://bcache.evilpiepirate.org/`, 2010.

[69] STEC. EnhanceIO SSD Caching Software. `https://github.com/stec-inc/EnhanceIO`, 2012.

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

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

[72] Wikipedia. memcached. `https://www.mediawiki.org/wiki/Memcached`. Last visited on 11/8/2015.

[73] A. Wolman, M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. *SIGOPS Oper. Syst. Rev.*, 33(5):16–31, Dec. 1999.

[74] T. M. Wong and J. Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC)*, pages 161–175, 2002.

[75] N. Young. On-line Caching As Cache Size Varies. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '91, pages 241–250, Philadelphia, PA, USA, 1991. Society for Industrial and Applied Mathematics.

[76] E. Zemel. An $O(N)$ Algorithm for the Linear Multiple Choice Knapsack Problem and Related Problems. *Inf. Process. Lett.*, 18(3):123–128, Mar. 1984.

[77] Y. Zhou, Z. Chen, and K. Li. Second-Level Buffer Cache Management. *IEEE Trans. Parallel Distrib. Syst.*, 15(6), June 2004.