3-23-2021

A Study of Non-datapath Cache Replacement Algorithms

Steven G. Lyons Jr.
Florida International University, slyon001@fiu.edu

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

A STUDY OF NON-DATAPATH CACHE REPLACEMENT ALGORITHMS

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Steven Lyons

2021

To: Dean John Volakis
College of Engineering and Computing

This dissertation, written by Steven Lyons, and entitled A Study of Non-Datapath Cache Replacement Algorithms, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Jason Liu

Giri Narasimhan

Ning Xie

Gang Quan

Raju Rangaswami, Major Professor

Date of Defense: March 23, 2021

The dissertation of Steven Lyons is approved.

Dean John Volakis
College of Engineering and Computing

Andres G. Gil
Vice President for Research and Economic Development
and Dean of the University Graduate School

Florida International University, 2021

DEDICATION

To my family, the new and the old.

ACKNOWLEDGMENTS

Thank you everyone who has been a part of this experience!

ABSTRACT OF THE DISSERTATION

A STUDY OF NON-DATAPATH CACHE REPLACEMENT ALGORITHMS

by

Steven Lyons

Florida International University, 2021

Miami, Florida

Professor Raju Rangaswami, Major Professor

Conventionally, caching algorithms have been designed for the datapath — the levels of memory that must contain the data before it gets made available to the CPU. Attaching a fast device (such as an SSD) as a cache to a host that runs the application workload are recent developments. These host-side caches open up possibilities for what are referred to as non-datapath caches to exist. Non-Datapath caches are referred to as such because the caches do not exist on the traditional datapath, instead being optional memory locations for data. As these caches are optional, a new capability is available to caching algorithms that manage these caches: not caching at all and instead bypassing the cache entirely for an access. With this option, items may not get inserted into the cache, which is an outcome that is beneficial to the lifetime of cache devices that can only sustain a limited number of writes before they degrade and become unusable. We propose several non-datapath caching algorithms that are optimized to make these choices and achieve high hit-rates while reducing writes: mARC, FOMO, and ANX. We also propose a polynomial time optimal solution to non-datapath caches: mOPT. mOPT provides an optimal solution for non-datapath caches that can maximize hits while minimizing writes. This provides further insight into how appropriate non-datapath caches are for specific workloads, if at all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE PAGE

CHAPTER 1

INTRODUCTION

Many applications, either within the enterprise, cloud, or big-data environments have had storage systems become increasingly important both in terms of larger storage systems and, more importantly, the distribution and availability requirements of such systems with the goal of fast, reliable access to data for end-users. Within these storage systems, large capacity storage is often composed of the more cheaply available magnetic disks, with caching allowing for speedup of access to important or recently/often accessed data. While DRAM is typically used within many systems for caching, the complexity of applications running on systems, like distributed computation systems, may effect the amount of space available within such memory [MS14]. On top of this large use of DRAM by the system, DRAM itself is costly and volatile. This cost means that DRAM has to be limited and a slower but less costly medium is necessary for a viable cache for these systems.

Flash devices, such as solid state drives (SSDs) have become increasingly more commonplace within storage systems due to their read and write speeds and comparatively low cost for the memory space when compared to DRAM. These devices are only able to handle a given amount of erasures and writes before they are unable to hold new data, producing errors or potentially making the device corrupt [BD10]. Amongst the latest developments in storage technology, the introduction of NVM (Non-Volatile Memory) devices have produced fervent speculation into not only its benefits but how best to use it. Being faster than flash devices but slower than DRAM, its immediate impact on performance when used is certainly notable [KBC+15]. However, the current estimations for the number of write cycles a NVM device, such as Intel's 3D Xpoint technology, can have before being prone to errors is estimated at a rate similar to that of SSDs [ZHZ+18].

Flash and NVM devices (such as 3D-XPoint) have replaced magnetic disk drives in many systems, having the filesystem located on the devices themselves and improving

boot times and overall system performance [HAWS13]. However, the cost and size of these devices are still comparatively high when compared to magnetic disk drives, a much slower medium. Due to this, many systems where a large, slower storage are required have accommodated flash and NVM devices as large caches to speed up access to frequently requested items by keeping such data from these slower devices [HAWS13]. We henceforth shall refer to the cache configuration where the slower storage is being communicated with over a network while the cache may be either flash or NVM, as host-side caches. A benefit to the use of these host-side caches is the possibility to recover cached data in situations such as a power loss, as unlike DRAM caches, the devices used as caches would be non-volatile [HAWS13]. These host-side caches, when given a request, are capable of a choice impossible for DRAM, or other datapath caches: to not cache a requested item [SLK+15].

The ability for non-datapath caches to not cache becomes even more important when we take into consideration what kind of devices they typically are: flash and NVM devices, devices whose number of write cycles are relatively limited. With this, it has been acknowledged that caching algorithms that know and operate on non-datapath caches have two objectives: high performance by having many hits, and reducing the number of writes to the cache in an effort to expand the cache's usable lifespan. The choice not to cache a requested item brings new revelations and opens new opportunities. Belady's MIN algorithm, a provably best caching algorithm for caches, now has become incompatible when solving for non-datapath caches as MIN always caches on a miss [Bel66]. Other algorithms that are used in practice: CLOCK, FIFO, LRU, ARC, LIRS, all have this same incompatibility with non-datapath caches as they too always cache on a miss [CoT68, DT90, MM03, JZ02].

So, it comes as no surprise that efforts have already been taken to solve this optimality for non-datapath caches. However, many who have approached the problem have not

proposed exact solutions, instead deferring to approximations with the use of intuitive heuristics [CDS+16]. Beyond this, use cases can vary with the write policy of the cache:

- *Write-back:* Caches managed with a write-back policy find hits, even those that lead to writes, valuable.
- *Write-through:* Caches managed with a write-through policy find hits that lead to writes to be evaluated the same as an insertion.
- *Write-around:* Caches managed with a write-around policy find hits that lead to writes to be disadvantageous, as a write invalidates the item in the cache.

This presents an argument for multiple cost models that a non-datapath optimization algorithm would need to accommodate for.

Existing solutions focus on the limitations of non-datapath caches and put forth work to reduce the writes to the cache while attaining reasonable performance. Such solutions have included LARC, a non-datapath caching algorithm that waits until a second access to an item before it is inserted into the cache [HWC+13]. This approach means the loss of several hits, placing priority instead on not caching unique accesses such as scans or random accesses, leading to a lowered hit-rate to achieve a reduction in writes to the cache. Other solutions include those like LFO, a caching algorithm that uses machine learning and a model of OPT (Belady's algorithm) to make decisions to cache or not [Ber18]. As we shall expand on later in Chapter 2, this approach has the problem of not being considerate to reducing writes to the cache and is an online algorithm, not an exact solution to the problem. There is also RL-Cache, a Reinforcement Learning caching algorithm that shows promise, though its use of machine learning through TensorFlow has it require a GPU for further computations [KSGS19].

In this thesis, we explore non-datapath caches in detail and identify several problems. First, we understand that non-datapath caches have the choice to not cache and that many

other approaches either provide compromises to hit-rate or complexity in the pursuit of a solution. To address this problem, we proposed mARC, a non-datapath caching algorithm that identifies the workload state in order to determine whether caching should be optimal or not. Second, with an understanding of mARC, we saw that mARC was able to identify the workload state, however it had a missing transition that unnecessarily inserted items into the cache. To address this problem, we proposed FOMO, an improvement upon mARC that was capable of not only understanding of the workload from a different angle, but was also capable of utilizing any other algorithm as its eviction expert. Third, working with FOMO we found that its main problems were its tunable parameters, its speed to identify workload states, and its potential to get stuck in a local minima. To address these problems, we propose ANX, an experimental algorithm built upon our understanding from the previous work of mARC and FOMO. ANX is a non-datapath caching algorithm that uses a single parameter to drive its decisions, while using randomization to make use of probabilistic understanding of the workload that allows ANX to react to workload state changes much more quickly. In the last part of the thesis, we will focus on the current lack of a polynomial time solution to the non-datapath caching problem and finally propose a conversion of the problem into a min-cost flow graph capable of solving the non-datapath caching problem in polynomial time.

The next chapter outlines the main contributions of this proposal. It starts with a list of the proposed problems, followed by descriptions of our proposed solutions and their significance. Each chapter of this dissertation is written to be as self contained as possible. However, in order to better understand the material here presented, readers are encouraged to get familiar with the terms and concepts described in Chapter 3. The rest of this document dedicates a chapter to each of the contributions we have made to address the proposed problems. We conclude the dissertation with a list of related literature and a detailed description on how our solutions differ those presented previously.

CHAPTER 2

PROBLEM STATEMENT

This chapter introduces the proposed research problems and their significance. In addition, this chapter also identifies the major challenges associated with the problems in question and outlines our unique contributions.

2.1 Thesis Statement

We propose building non-datapath cache solutions that both focus on using an understanding of the workload to make caching decisions and tackle the theoretical optimization questions by:

1. developing a hit-rate based, workload-aware non-datapath caching algorithm,
2. building upon the hit-rate based, workload-aware caching algorithm by making it both generalized and adaptive,
3. developing a new non-datapath caching algorithm that uses individual access indicators for workload-aware capabilities, and
4. providing a reduction of the non-datapath optimization problem to benefit from a polynomial time solution to this known problem.

2.2 Thesis Contributions

This thesis addresses the challenges of determining the correct decision in non-datapath caches when trying to optimize both for hit-rate and for writes, making four distinct contributions.

First, we developed a hit-rate based, workload-aware non-datapath caching algorithm that builds upon the classic ARC caching algorithm. Existing caching algorithms either

do not consider the main flexibility afforded to non-datapath caches (not caching on a miss) or typically employ methods that tended to optimize more for either hit-rate or writes. We found that workloads can be broken down into parts that exhibit definable behaviors. With an understanding of the workload, we found that there were many instances wherein an entire state is defined as benefiting from either always caching or always not caching. In this thesis, we develop mARC, a non-datapath caching algorithm that utilizes this understanding, extending ARC to identify workload states and cache or not based solely off of these identifications, with the idea that a proper balance between caching and not caching leads to optimal non-datapath performance in terms of both hit-rate and writes.

Second, we developed further off of the main ideas of mARC, building a new non-datapath caching algorithm that corrected many of the shortcomings found in mARC. Working on an understanding of mARC's limitations we generalized mARC so that not just ARC, but any datapath caching algorithm can be easily integrated without modification. This generalizability allows us to adopt whichever is the best datapath caching algorithm, doing so allows any datapath caching algorithm to function appropriately in the non-datapath context. Furthermore, FOMO accounted for all possible workload state transitions by simplifying mARC's states and transitions into two states and two transition conditions. In this thesis, we develop FOMO, an adaptive, generalized non-datapath cache admission policy that iterates on mARC to improve performance further.

Third, we revisited the definitions we created and used for mARC and FOMO, looking for new definitions and identifiers that did not require a period of observation to determine the workload state. This reliance on an observation period is something to be avoided, as it slows down mARC's or FOMO's ability to react and adapt to changes in the workload behavior. This is especially harmful if these changes are towards behaviors unfavorable

to mARC's or FOMO's current state, where, until noticed, they will commit to improper actions for the behavior.

The resulting algorithm is a completely unique creation, ANX, an adaptive, non-datapath caching admission policy that explores a new direction to help in understanding the workload at runtime: a miss history. This miss history keeps track of the past decisions the algorithm would have made in regards to a requested page: *Inserted* and *Filtered*, which represent the decisions of *caching* and *not caching*, respectively. This allows the algorithm to identify both bad choices (like not caching a page that would have benefited from being cached) or help identify a workload state before pages were evicted in many cases, further explored in Chapter 6.

Typically, other algorithms are focused on the eviction history in order to improve performance. This use of an eviction history is mostly a technique that comes from an understanding that if a page is found within the eviction history, the algorithm made a mistake in evicting it. This, unfortunately, is an attempt to identify important pages to keep in the cache, whereas the miss history instead focuses on not caching unimportant pages, which can avoid those bad evictions in the first place. Additionally, the miss history focuses on far more recent actions of the algorithm, allowing the algorithm to adapt quickly to recent changes in workload behavior, rather than passively protecting against bad workload behaviors through partitioning (a technique used by both ARC and LIRS). The workload state transition understanding from mARC and FOMO was also reworked with the understanding that important transitions were missing: when Stable transitions to and from Unique Access. ANX uses intelligent identification methods in conjunction with the miss history to ascertain the current workload state rapidly and adapt just as quickly, based on these identifiers.

Finally, we move our focus from the practical applications of our understanding of non-datapath caches to the theoretical. While previous work has gone into producing

solutions for non-datapath caches that optimize for both hits and writes, all of the work on the problem thus far has been empirical, and no optimal solutions to the problem have yet been uncovered. There are also claims that a solution for non-datapath caches has potentially non-polynomial time complexity, though not proven. In this thesis, we propose a way to reduce the non-datapath cache optimization problem to a min-cost flow problem that is capable of solving the non-datapath cache optimization problem in polynomial time. We have named this solution mOPT (modified-OPT).

2.3 Thesis Significance

The increasing availability and deployment of flash and 3D-XPoint based host-side caches represent a huge opportunity to accelerate the slowest component of computing systems — storage. Previously existing solutions, being focused on datapath caches, are not made to make the best use of these caches [Bel66, CoT68, DT90, MM03, JZ02, Li18]. Though these solutions are still capable for achieving good hit-rate, their datapath requirements of always caching on a miss, incur a large number of writes, lessening the lifespan of the host-side caches unnecessarily. The latest solutions explore the new ability of these host-side caches: to *not* cache. However, these solutions either focus too heavily on the reduction of writes or use advanced techniques that are more resource intensive than can be typically allowable in production [HWC$^+$13, Ber18, KSGS19]. In this thesis, we propose solutions that are less resource intensive and, while focusing on the reduction of writes to the cache, perform as well, if not potentially better, as these previously existing datapath caching algorithms.

Along with the previous datapath caching solutions being unsuitable for non-datapath caches, the previous optimization algorithm, Belady's MIN algorithm, has become improper for determining both optimum hit-rate and, a new concern, writes for a cache

[Bel66]. As Belady's MIN suffers from the same datapath cache restriction of always caching on a miss, it incurs more writes, as discussed before, but potentially more important: *the option to not cache can cause the hit-rate to* **improve**! Belady's MIN is often used as a comparison point for datapath caching algorithms to represent how close the algorithm is to optimal, it follows that such an algorithm for non-datapath caches would garner the same importance of comparisons, used as a measurement of the best an algorithm can achieve at a given cache size in this environment. Previous solutions proposed instead the use of heuristics for approximating reasonable comparisons, even if these answers were not true optimal solutions for the non-datapath cache version of the problem [CDS+16]. In this thesis, we propose a polynomial time solution capable of determining the optimal solution for non-datapath caches.

2.3.1 mARC

With an understanding of the capabilities and requirements for non-datapath caches, we made an effort to look for hints in the composition of workloads. From this effort came the classification of three workload states and the recognition that each of these states had a preferred action, where either caching or not caching was preferable in each state. With these workload states identified, the next step was how could one discover them from a caching algorithm context. The answer we arrived at, was using our understanding of these states to identify state changes with the changes in hit-rate. Using ARC as a base and adding non-datapath capabilities, we developed mARC, a non-datapath cache that identified the current state of the workload and would the perform the preferred action of that state. This gave us an opportunity to not compromise hit-rate for reduced writes in the non-datapath cache or great resource requirements.

2.3.2 FOMO

With our previous understanding from mARC, we moved towards an improved solution, one that would have improved capabilities and generalizability, allowing any datapath caching algorithm to become proper for the non-datapath caching environment. Since mARC relied heavily on tunable parameters for much of its workload state transition decisions, it was possible that certain workloads would need tuning in order to have an improved performance. Furthermore, mARC misses a potential state transition between two of its states, requiring not only an extra transition but unnecessary insertions into the cache. Understanding this problem, we worked on simplifying our understanding of mARC, considering a new understanding of the relationship between internal (cache) hit-rate and external hit-rate through the use of a structure we call the miss history. In the end, we developed FOMO, a generalized non-datapath admission policy capable of improving its ability to identify the current state of the workload through a simplified state model.

2.3.3 ANX

While mARC was capable of identifying the state of the workload, it was complex, with many tunable parameters that it would use for its decisions. FOMO improved upon this by introducing a simplified state model, but due to both mARC and FOMO's reliance on cache and filter list (or miss history) hit-rate to identify workload states, they have an observation period that must pass before evaluating these rates and identifying the workload state, delaying changes that may be critical to preventing the cache from evicting important pages while inserting pages of low importance, reducing the resulting hit-rate while incurring several unnecessary writes. In an effort to address these concerns, we revisited the definitions and understandings of workload states and found our revised definitions constantly using two terms: *old* and *new* when referring to pages. This drove us away

from using separate filter and eviction histories and put our focus squarely on misses, or more accurately, recent misses and the relevant non-datapath decision taken in regards to these misses. Though it seemed almost sacrilegious to ignore the eviction history, a miss history allows us to make a relevant distinction between *old* and *new*, where new pages would exist in the miss history and old pages would be in the cache but not in the miss history, and has more recent events recorded that we would want to respond to. This shift to a far more recent event, the miss, gives us a proper opportunity to figure out the workload, rather than to put a page into the cache, see if it gets hits while in the cache and then see it being evicted at a later time in order to finally determine qualities of the workload, while the workload may have changed in the time since then. With this, the focus on limiting the number of parameters, to simplify our solution compared to that of mARC's parameters (and even FOMO's), had us finally reduce everything to a single parameter to drive the cache, which we named *anxiety*. With this model, we developed ANX, a non-datapath caching policy that uses probability and adapting to identifiable state behaviors to determine whether it should cache or not. ANX is able to react quickly to changes in workload state and is incredibly simple for the hit-rate it is able to achieve. On top of all this, ANX is more than capable of reducing the number of writes to the cache significantly, providing evidence that this often ignored pipeline, the miss history, may be far more important to cache performance than the caching community may have imagined. The design of ANX, and in particular the identifiers it uses are an interesting departure from traditional design philosophies that shows promise and merits further exploration and experimentation.

2.3.4 mOPT

While non-datapath caches have clearly stated statistics to optimize for (hits and writes), the way to find the optimal solution for a given cache size had remained an open problem. Having this point of comparison strengthens non-datapath caching algorithms and allows measuring the gap in performance between such an algorithm and the optimal answer. This gap, similar to the one existing when comparing Belady's MIN to other datapath caching algorithms, is important evidence that further improvement is possible. Presently, the solutions provided have mostly come in the form of heuristics that make use of an understanding of the requirements for non-datapath caches to find good enough solutions for these comparisons [CDS+16]. However, the optimal solution has been elusive. In this thesis, we present the mOPT solution to non-datapath caches, capable of finding an optimal solution to a given cache size in polynomial time, providing the missing comparison point with a best possible solution, rather than an approximation as presented so far.

The following chapter will discuss the background details for many of the presented concepts, assisting those previously unfamiliar in the subject matter to understand the significance of our contributions further.

CHAPTER 3

BACKGROUND

In this chapter we define concepts that are prerequisite for understating the problems described in this thesis as well as the proposed solutions.

3.1 Storage Architectures

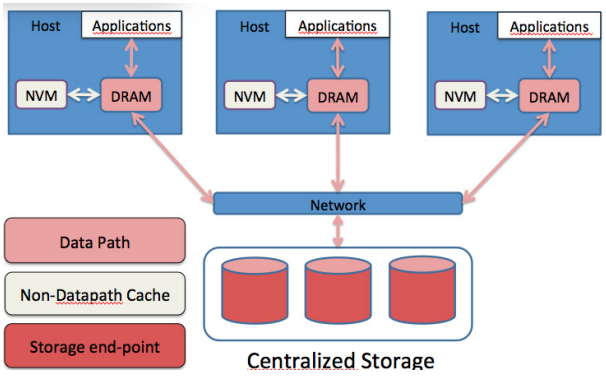

Figure 3.1: Centralized Storage Architecture

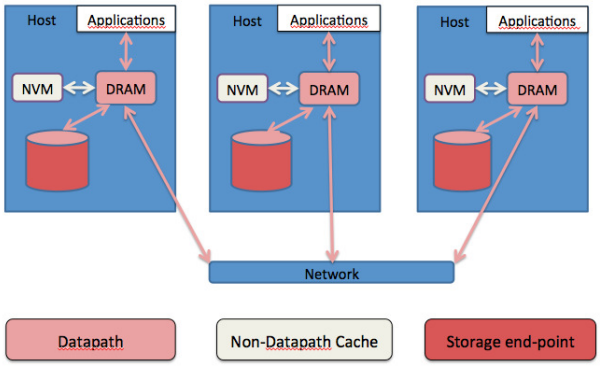

Figure 3.2: Distributed Storage Architecture

Among network storage systems, two particular architectures are common in data-centers: centralized storage and distributed storage. With centralized storage, the host servers communicate over the network to a centralized storage device, such as an array-based storage, as seen in Figure 3.1. With distributed storage, the host servers have their

own storage attached, however, data may also be retrieved from another server over the network, as seen in Figure 3.2. DRAM has been used before to cache this remote data, improving performance as when the data is located in DRAM a request can be served without needing to go over the network, and significantly decreasing the latency of the operation, as the roundtime for a DRAM request is several orders of magnitude faster than the network attached storage [KMR+13].

3.2 Host-Side Caches

While DRAM is used as a cache and more than capable of improving the performance of the servers in network storage systems, the amount of data being used and processed is only ever increasing, increasing the size DRAM would need to be in order to be sufficient for such working set processing requirements. Host-side caches typically use non-volatile memory (NVM) technology such as flash or 3d-Xpoint. They are represented as Non-Datapath caches in Figures 3.1 and 3.2 when used with centralized and distributed storage respectively. Host-side caches, be it flash or persistent memory, are high performance and high capacity caching devices that improve performance when accessing storage over a network while being able to handle larger working sets than DRAM caches using their larger capacities [BLM+12]. While these host-side caches are typically an order of magnitude or two slower than DRAM, the increase in latency by using these slower storage devices still results in a latency far lower that of accessing any network attached storage [WOHL17, BLM+12]. The current estimation for the number of write cycles flash or persistent memory devices, such as 3D Xpoint, are able to have before being prone to errors are roughly the same, making these devices wear faster when writing is more common [ZHZ+18].

These host-side caches, being typically flash or persistent memory devices, are organized into memory cells. While flash devices are block addressable, persistent memory devices tout the capability of being byte-addressable. Despite the method of addressing these devices, blocks are the most common form the data takes when it is being accessed.

These host-side caches, compared to DRAM memory caches, are not a requirement for the datapath memory hierarchy to function properly. As such, when these host-side caches are present, the placement of data within them is optional; meaning that misses to the host-side cache do not carry the datapath requirement that the data *must* then be inserted into the cache. Due to their larger and slower nature compared to DRAM, host-side caches provide another cache level to query prior to other, slower devices. Due to these caches not being a part of the datapath memory hierarchy these caches can be referred to as non-datapath caches.

3.3 Write Caching Policies

When using a cache, there are multiple write policies that can be used, each with their own benefits. While the models we create for caching writes in non-datapath caches allows for multiple write policies, we assume the implementation of such write policies are orthogonal to the design of the cache replacement algorithm, which is the focus of this dissertation.

A classic write caching policy is write-back, where when a write request has a hit in the cache, the data is written only to the cache; this marks the page as dirty, and only when the page is evicted from the cache is the data written back to its storage. This reduces requests to the storage device and improves performance as the process only has to wait until the data is written to the cache to continue. This write caching policy benefits write

heavy workloads the most. This write policy has the other beneficial effect of reducing the volume of write I/O traffic to the network storage system [KMR+13].

Another classic write caching policy is write-through, which is when a write request has a hit in the cache, the data is written to both the cache and the storage device. This increases the number of requests to the network storage device and hurts performance, when compared to write-back, as the process must wait until the data is written to both the cache and the storage before continuing. However, what is typically gained is a level of durability of the write, since it is guaranteed to be in the storage device, rather than have dirty, or more up to date version, data in the cache should any failures occur [KMR+13].

A less classical write caching policy, but one used in enterprise systems, is write-around, where, should a write request have a hit in the cache, the cached page is marked as invalid, removing it from the cache. This increases the number of requests and hurts performance, just as write-through does, though it can be expected to be even worse as read-after-write behaviors are both misses in a write-around scenario. This write caching policy is mostly beneficial for workloads where there are either a limited number of writes or when written data are not typically read immediately afterwards [KKI+14].

3.4 Consistency

For network storage systems, where a specific instance of data can exist amongst the several hosts, the problem of consistency of this data across these hosts can become a concern. The problem of consistency within network storage systems has been explored thoroughly from multiple angles. One such angle is automating the decision of what kind of consistency is necessary for a system [LLC+14]. These types of consistency consists of weak consistencies (causal, eventual, and timeline) and strong consistencies (serializability and linearizability). Weak consistencies gain performance by having a small number

of replicas to reduce the number of contacts and operations necessary before replying to a user. Strong consistencies attempt to treat a networked system as one server, serializing operations. Another angle explored is the impact of write policies on the consistency and durability guarantees for host-side caches [KMR+13].

In regards to this dissertation, consistency of these caches is an orthogonal concern to the caching algorithm itself.

3.5 Optimality

Belady's MIN Algorithm — also referred to as OPT — is a provably optimal caching algorithm for datapath caches that uses future knowledge of requests in order to make decisions on what to evict when inserting a request into a full cache on a miss. By optimal, we mean that MIN is able to find the maximal number of hits achievable for a given cache size in a datapath cache. The way that MIN is able to do this, is that on eviction, it chooses the page in cache that has the longest time until it would be reused. MIN has been used as a point of comparison for caching algorithms, demonstrating how far any are from this optimal.

CHAPTER 4

mARC

Non-datapath caches are relatively new entrants to the storage stack. All of the previous work on storage caching had been in the context of data path caches. In this chapter, we report on our first attempt at modeling the non-datapath caching problem, building solutions for it, and evaluating it against the state of the art. Specifically, we report on our learnings from analyzing a large set of production storage workloads in order to identify periods of time where there is a common behavior, what we refer to as a workload state. We then demonstrate how we use this knowledge of this state in conjunction with non-datapath caching to reduce writes to host-side caches while attaining reliable performance by developing mARC, a workload state aware non-datapath caching algorithm.

4.1 Introduction

CPU and main memory caches are *datapath caches*. Such caches incur *forced cache updates*; they are required to make a cache update on every cache miss so that the data is accessible by upper-level hardware or software. The widely used cache replacement policies today, such as(e.g. LRU [DT90], FIFO [DT90], ARC [MM03], MQ [ZPL01]) were designed for datapath caches. *Non-datapath caches*, on the other hand, are not required to perform a cache update on every cache miss. One can apply *opportunistic cache updates*, whereby case-by-case decisions can be made whether to perform a cache update. A host-side flash cache is an example of a non-datapath cache. Host-side flash caches are attractive because they can reduce the demands placed on network storage, speed up I/O performance, and provide I/O latency and throughput control [BLM$^+$12, KMR$^+$13, KMR15].

A cache update is composed of two operations: an *eviction* and an *insertion*. Performance wise, evictions can be detrimental. In case of a host-side flash cache, a dirty item

chosen for eviction needs to be written to the backing storage consuming both cache and network storage bandwidth and contending with other items being accessed at the same time. Furthermore, it can add significant latency to the access inducing the eviction. More significantly, since the item being evicted may be more valuable than the item being inserted in its place, cache updates can be detrimental to the cache hit-rate. This problem is especially acute for one-time access items, as in a streaming or random access workload, since they lead to inserting non-reusable items into the cache. Finally, flash-based cache devices have limited write cycles and cache updates also affect device lifetime.

Lazy adaptive replacement cache (LARC) [HWC+13] is a recent proposal that implements opportunistic cache updates. While LARC benefits from avoiding certain cache updates, since the LARC cache filter is always operational, it can also prevent important items from entering the cache in a timely fashion. As we shall demonstrate later, because of this shortcoming, LARC performs worse than ARC for the MSR Cambridge Workloads.

Since LARC, other efforts in online non-datapath caching algorithms has been made. Two such examples would be Reinforced Learning Cache (RL-Cache) [KSGS19] and Learning From OPT (LFO) [Ber18]. RL-Cache is a non-datapath caching algorithm that uses reinforcement learning to figure out whether to cache or not cache items. RL-Cache requires the use and availability of several threads and GPUs in order to achieve its results. Furthermore, RL-Cache is primarily designed for variable sized object caches, as RL-Cache also attempts to reduce the number of objects within its cache. LFO is a non-datapath caching algorithm that uses machine learning supervised by a modified version of OPT to learn whether to cache an item or not based on what it thinks OPT would do. While LFO does well, it uses TensorFlow and has large amounts of overhead that makes it inappropriate for consideration for production environments. In particular, these

environments run workloads with a lot of dynamic behavior and learning needs to be lightweight, online, and continuous.

We propose *multi-modal adaptive replacement cache* (mARC), a non-datapath version of the adaptive replacement cache (ARC) algorithm. mARC is designed to avoid unnecessary cache updates. It identifies three possible states in which a workload may be operating in at any given time — Stable, Unstable, and Unique Access — and selectively disables cache replacement depending on the state.

An evaluation of mARC using a cache simulator for the MSR and FIU block I/O traces from SNIA [VKUR10, SNI19] is encouraging. For the MSR Cambridge Traces, while maintaining a competitive hit-rate compared to ARC (1% worse on average), mARC reduces the number of cache updates by 25% on average. This translates to a significant improvement in flash cache device lifetimes. For the FIU traces, mARC leads to 9% better hit-rate on average while reducing the number of cache updates by 23% on average, when compared with ARC. These results motivate further investigation into replacement algorithms that are specifically designed for non-datapath caches.

4.2 Motivation

In this section we discuss the workloads for storage systems and the states they consist of, define the concept of churning, then identify the problems with ARC in regard to these states, and lastly the discussion of the shortcomings to LARC's design.

4.2.1 Dynamic Storage Workloads

Storage workloads are dynamic. We model this dynamism using a simple Active-Unique (A-U) model that is time-aware and describes the amount of unique data accessed as well as the amount of data that is reused (active data) in the workload. A sample A-U plot is

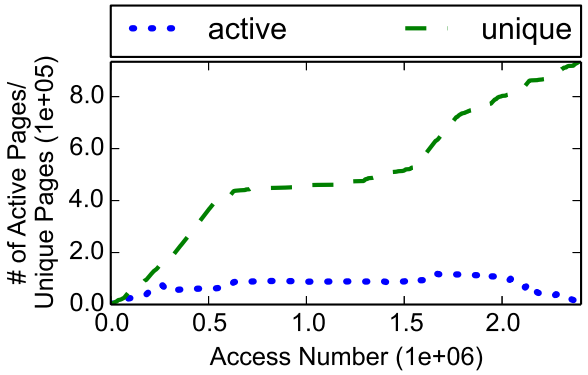

Figure 4.1: A-U plot for one day of the prn0 MSR trace

presented in Figure 4.1 for one day of the *prn0* MSR Cambridge trace. The A-U model tracks the number of *active* and *unique* pages accessed over time. The active pages at any instant are the unique pages accessed previously and that will be re-accessed at some time in the future. As pages get accessed, the unique set of pages never decreases, while the active set may either increase or decrease depending on the future reuse.

Several combinations of A-U states can be identified with corresponding workload behavior. We say that a workload is in a **Stable** state in a given period if the set of items that are currently referenced and their relative importance (e.g. relative frequency of access) remains approximately the same as the previous period. When the set of active and unique pages both remain unchanged, the relative frequency of items being accessed determine if the workload is in a **Stable** state. When the set of unique pages remains the same but the set of active pages decreases, as well as when both the active and unique page sets increase, the working set is changing, and the workload is an **Unstable** state. When the set of unique pages increases but the set of active pages remains unchanged, one time items (either streaming or random) are being accessed and we refer to the workload as being in the **Unique Access** state.

While the active/unique page states provide a good framework to understand behavior, accurately tracking the active/unique page sets and their relative importance is expensive.

Further, it is not possible to identify active pages in practice, given that it would require

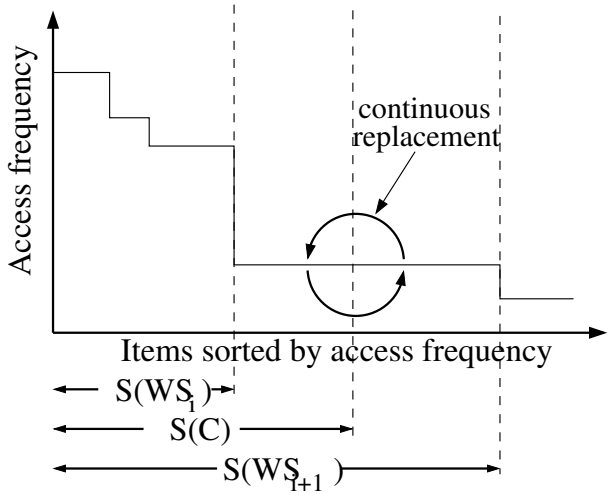

Figure 4.2: Cache churning behavior
S(WS_i) is the size of the working set at granularity i that is entirely contained within the larger working set WS_{i+1} [KVR10]. S(C) is the cache size. Since the cache is not large enough to contain all pages of WS_{i+1}, non cache-resident pages of WS_{i+1} continuously replace cache-resident ones.

knowledge of future accesses. In practice, we found that the *cache hit-rate*, a relatively lightweight metric, can serve as an effective proxy for identifying the above set of workload states and state change events.

4.2.2 The Deceptiveness of Stability

If the workload is in a **Stable** state, a cache is expected to perform well. However, if the cache is not large enough to contain the current set of active pages, the cache contents can *churn* needlessly due to forced cache updates in conventional caching algorithms. Such churning involves the constant eviction of cached pages when pages with relatively equal or lesser importance get accessed. Figure 4.2 illustrates cache *churning* behavior within the **Stable** state. Cache replacement due to such churning is detrimental to cache

hit-rate. The optimal operation in such situations is the ”noop”, i.e., not to perform cache replacement. This phenomenon has been observed for CPU caches earlier [QJP+07].

4.2.3 ARChilles' Heel

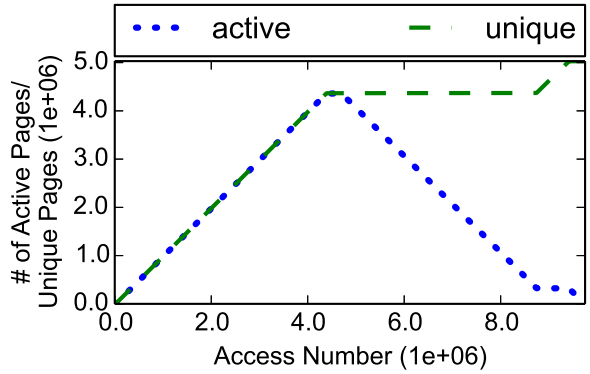

Figure 4.3: A-U plot for one day of the src2-1 MSR trace

ARC [MM03] is a high-performance algorithm for datapath caches. During **Unstable** periods, ARC updates the cache contents efficiently by quickly detecting the important items in the new working set. During **Unique Access** periods, ARC retains the frequently used items while unique items tracked in ARC's T1 list pass through the cache quickly. During a workload's **Stable** state, ARC is designed to retain frequently used items from its T2 list in the cache. However, ARC's cache updating behavior can compromise cache hit ratio when a stable working set does not fit in the cache. Relatively less frequently used items continuously lead to cache misses and cache contents churn needlessly.

Figure 4.3 presents an A-U plot for src2-1 MSR Cambridge trace. In this workload, the number of active and unique pages climbs steadily (introducing a new working set) until about 4.2M accesses. Following this, the workload reuses a subset of these pages for about 0.2M accesses and then accesses a majority of the 4.2M unique pages exactly once again causing a steady decrease in active pages while the number of unique pages stay the

same. If the cache size is smaller than the maximum number of active pages (i.e., smaller than 20GB), ARC would continuously evict pages that are about to be used in this trace. This churning would result in a significantly increased cache miss rate.

4.2.4 Avoiding Forced Cache Updates

This recently proposed LARC [HWC+13] algorithm implements a filtering mechanism that *always* avoids cache updates for items not accessed sufficiently recently. LARC consists of two LRU lists, one for cached items and a *filter list* for tracking items that were not in the cache or in the filter list when last accessed. A cache update is only performed when an item that is not found in the cache *is* found in the filter list. The filter list size grows (*resp.* shrinks) with a decrease (*resp.* increase) in cache hit-rate.

While LARC presents a new approach for avoiding cache updates, it has its own set of weaknesses. LARC's filter is always operational and can prevent important items from entering the cache in a timely fashion. More specifically, LARC populates the cache in a timely fashion. More specifically, LARC populates the cache at least twice as slowly as other algorithms when workload working sets change, negatively affecting cache performance. In such situations, LARC can perform significantly worse than ARC. LARC's filtering mechanism can reduce the probability of content churning in the cache by shrinking the size of the filter. However, it can only be successful in doing so when the cache hit-rate is sufficiently high; in other cases, LARC is unable to avoid churning.

4.3 Design

In this section, we discuss the design and the reasons behind the design decisions. We first focus on workload states, a core driver for the start of our design, continue onto the development onto a state model, and finally on the design of mARC itself. mARC builds

on the strengths of ARC and LARC while addressing their weaknesses, leading to significant benefits for non-datapath caches. The intuition behind mARC is that workloads comprise of multiple states and filtering (or *not* filtering) exclusively is not effective in every workload state.

4.3.1 Workload States

mARC characterizes workloads as a state machine with three states — **Stable**, **Unstable**, and **Unique Access**. By avoiding cache updates when cache contents would otherwise churn, mARC improves cache hit-rate. By avoiding unnecessary cache updates, it also improves data access latency and extends cache device lifetime.

Algorithm	Stable (no Churning)	Stable (Churning)	Unstable	Unique Access
ARC	✔	✗	✔	∼
LARC	✔	∼	∼	✔
mARC	✔	✔	✔	✔

Table 4.1: Algorithms and workload states
Legend: ✔(full support), ∼ (partial support), ✗(no support)

During the **Unstable** state, mARC implements a simple ARC access for each referenced item, i.e., performs no filtering. mARC implements *filtering* in the **Unique Access** and **Stable** states. In these states, items either enter the cache or are registered in a *filter list*, which only stores metadata about the item. On a cache miss, items that are not found in the filter list get added to the filter list, whereas those that are found in the filter list results in an ARC access within the cache. The size of the filter list is maintained and updated as in LARC [HWC+13]. Table 4.1 provides a qualitative comparison of ARC, LARC, and mARC.

4.3.2 The States of mARC

To efficiently identify workload states, mARC uses sampling. It maintains two indicators to track workload state: a running average cache hit-rate that has been accumulated during the current workload state (HR_{state}) and the cache hit-rate during the current *sample period* (HR_{sample}). HR_{sample} is the hit-rate computed over the last n accesses where n is the size of the cache. In practice, n accesses provide us a valuable mean hit-rate by ensuring adequate coverage of items compared to the working set resident in the cache.

State	Condition	Action
Stable	$HR_{sample} \leq 0.7*HR_{state}$	Switch to Unstable
Unstable	$0.9*HR_{state} \leq HR_{sample} \leq 1.1*HR_{state}$	Switch to Stable
Unstable	$HR_{sample} \geq 1.2*HR_{state} \wedge HR_{sample} > 0.2$	Switch to Stable
Unstable	$HR_{state} > HR_{sample} \vee HR_{sample} < 0.1$	Switch to Unique Access
Unique Access	$0.1*HR_{sample} > \text{Filter-}HR_{sample} \vee HR_{sample} > 0.1$	Switch to Unstable

Table 4.2: mARC State Transition Table

mARC operates as follows. Every n accesses, HR_{sample} is compared with HR_{state}. mARC resets the value of HR_{state} only when entering an **Unstable** state to quickly and accurately track the hit-rate of a new working set. Doing so for the other states is not necessary. While in the **Unstable** state, to increase confidence and robustness, HR_{state} tracks at least $2n$ accesses before checking if a change of state should take place. Table 4.2 depicts the state machine that mARC implements. The constants used for each condition were determined by experimenting with a subset of the possible combinations of feasible values across all the 45 I/O workloads from MSR Cambridge and FIU [VKUR10, SNI19]. mARC starts its operation in the **Unstable** state. The rest of this section discusses how the various states of mARC operate, how state transitions occur, andthe resulting impact to the caching mechanism.

Stable State

At the beginning of a stable state, mARC enables the filtering mechanism, configuring the filter to its minimum size. ARC, active during the previous **Unstable** state, is expected to have populated the cache with the workload's working-set. Filtering in this state prevents cache pollution and needless cache updates. Starting with the minimum sized filter reduces the probability of cache churning as well. When signs of workload instability appear, the filtering mechanism is stopped and mARC returns to the **Unstable** state to repopulate the cache.

The only condition needed for mARC to transition out of the **Stable** state is that the performance of the cache is deteriorating over time because of a new working set. As shown in Table 4.2, we used a HR_{sample} that was 70% of the HR_{state} as a robust indicator of instability. In practice, we found that higher values make mRAC prematurely enter the **Unstable** state for a substantial fraction of the workloads.

Unstable State

In an **Unstable** state, mARC uses ARC without filtering with the objective of populating a new working set. As shown in Table 4.2, there are three possible conditions that lead to transitions out of this state:

- HR_{sample} is similar to HR_{state} — within 10% of each other. This implies hit-rate stability and mARC infers that the workload itself is stable. To respond, mARC moves to the **Stable** state to improve the hit-rate by avoiding cache churning and cache updates. We found 10% of HR_{state} to be an acceptable margin of error when detecting stability. If a more rigorous measure of stability were to be used, mARC delays entering the **Stable** state and in starting to filter unwanted items. If a less rig-

orous measure were to be used, the risk of changing state prematurely, and thereby compromising cache hit-rate, increases.

- If HR_{sample} is significantly higher than HR_{state} and also above a threshold, mARC determines that recent performance is better than the historical performance in the current state and moves to the **Stable** state. When HR_{sample} is 20% higher HR_{state}, then the new state is considered significantly better. However, a minimum threshold must be met by HR_{sample} for this increase to avoid really low values.
- When HR_{sample} is significantly lower than HR_{state} or HR_{sample} is below a minimum value, mARC infers streaming or random (one time) access behavior, and transitions to the **Unique Access** state. mARC employs 50% decrease in hit-rate as a good indicator of unique access behavior and this worked well in practice. Furthermore, if HR_{sample} is below 10%, unique access behavior is automatically inferred.

Unique Access State

In a **Unique Access** state, mARC turns on filtering mechanisms to avoid cache pollution due tot he cache updates and stops filtering only after transitioning out to an **Unstable** state. Table 4.2 shows the conditions that bring mARC back to the **Unstable** state. These involve detecting either that (i) a new working set it being introduced or (ii) unique access behavior has terminated. To determine if a new working set is being accessed, mARC samples the hit-rate of items in the filter (Filter-HR_{sample}). If this value exceeds 10%, it concludes that unique access behavior will soon terminate. mARC also uses a minimum HR_{sample} threshold to detect that the **Unique Access** state has terminated and a new working set may already be cache resident. In this case, switching to the **Unstable** state will allow confirming that the working set is indeed cache resident and eventually switch to the *Stable* state.

4.4 Evaluation

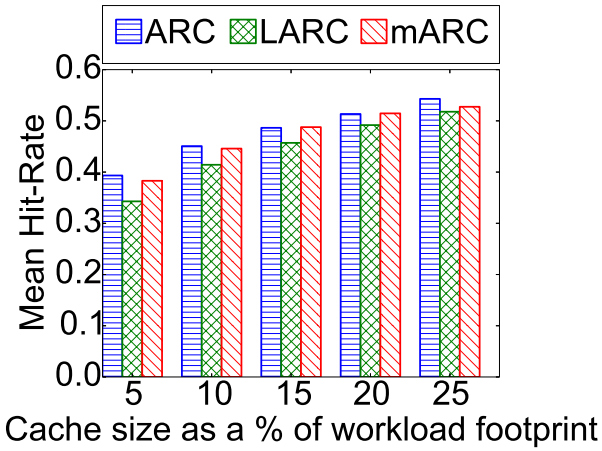

Figure 4.4: MSR Normalized Average Write-Rate

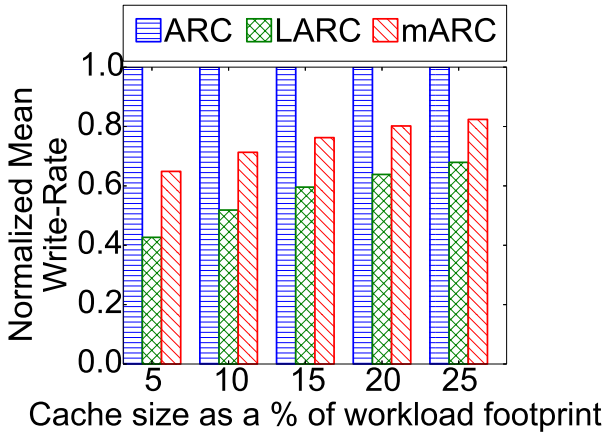

Figure 4.5: MSR Normalized Average Write-Rate

In this section, we observe the hit-rate and write-rate results of mARC. We built ARC, LARC, and mARC cache simulators which process a block I/O trace and report on the number of reads/writes, hits and misses, and clean and dirty evictions. To simulate sufficient cache space as well as I/O activity, we controlled the size of the cache to be a fraction of the *workload footprint*, defined as the combined size of all unique data accessed. We varied this fractional cache size from 5% to 25% in our simulations. Our

evaluation metrics include the mean cache hit-rate and the normalized mean write-rate, greater write-rates indicate lower flash cache device lifetime. For the workloads, we used the FIU and MSR Cambridge block I/O traces from the SNIA IOTTA trace repository [VKUR10, SNI19]. The MSR Cambridge and FIU traces are two large sets of 36 and 9 I/O traces respectively, from a variety of production servers/systems /citenarayanan08, VKUR10.

4.4.1 MSR

We first evaluate mARC for the MSR Cambridge traces, averaging across all of its workloads. Figures 4.4 and 4.5 depict how the cache hit-rate and cache write-rate vary for various cache sizes chosen as fractions of the workload footprint size. We observe that mARC has an average hit-rate that is very competitive with ARC (1% worse on average) and is greater than the LARC hit-rate (5% better on average) across the various cache sizes. LARC incurs the least number of cache writes (43% better than ARC on average), while mARC does 25% fewer writes on average than ARC.

4.4.2 FIU

Figures 4.6 and 4.7 depict results for the FIU traces, averaged across all the workloads. While mARC and LARC both provide higher hit-rate and ARC, LARC's hit-rate degrades as cache size increases form 10% to 15% of the workload footprint. This happens because LARC makes decisions on what to cache based on a secondary hit in the filter; since the filter size is proportional to the cache size, cache churning becomes a possibility. mARC does slightly better than LARC on average (1% better hit-rate) and also performs better with more cache space. LARC incurs the lowest write-rate (33% lower than ARC), whereas mARC also reduces write-rate by 23% compared to ARC.

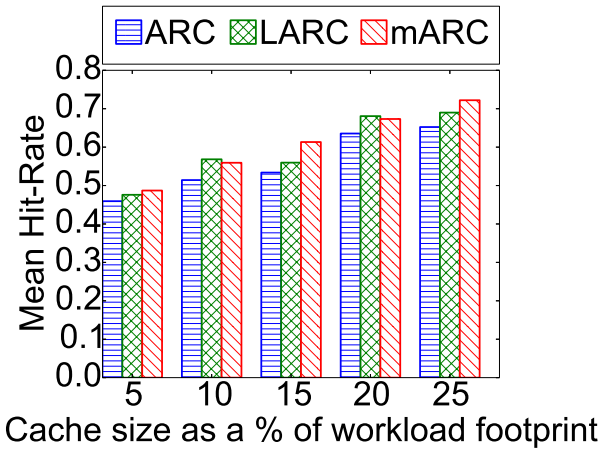

Figure 4.6: FIU Average Hit-Rate

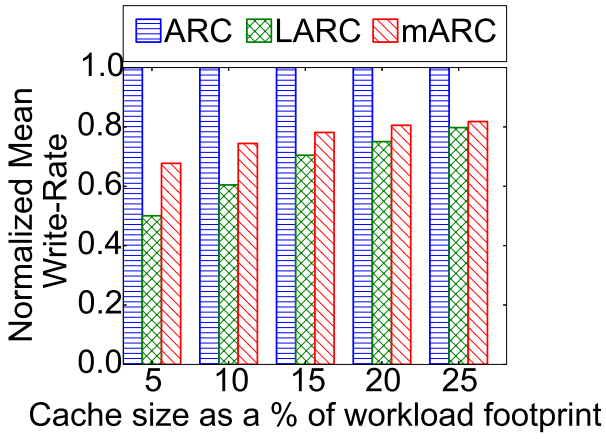

Figure 4.7: FIU Normalized Average Write-Rate

4.5 Summary

Conventional datapath caches have been managed by policies that incur forced cache updates. For non-datapath caches, these caching policies can become detrimental to cache performance and cache device lifetime. We demonstrate that managing datapath caches involves deciding whether and when to filter items from entering the cache, as well as to turn off such filtering when it becomes detrimental to performance. We show that for caching purposes, workloads can be modeled as a simple but useful state machine. mARC modifies the ARC algorithm equaling or exceeding its hit-rate while significantly lowering cache updates. mARC has its limitations though, the conditions are unchanging and don't address the variety of workloads that can be found and ARC may not be the best caching algorithm to use. In the next chapter, we will discuss FOMO, a solution that examines these problems further and works to correct them.

CHAPTER 5

FOMO

Unlike conventional caches, *non-datapath caches*, such as host-side flash caches, have distinct requirements. While every cache miss results in a cache update in a conventional cache, non-datapath caches allow the flexibility of *selective caching*, i.e., the option of not having to update the cache on each miss. This flexibility is important because cache updates are expensive and can be counter-productive to cache performance. Existing non-datapath caching algorithms, such as LARC and mARC, are limited in their ability to balance hit rate performance and cache writes, with LARC imposing misses to reduce writes and mARC imposing writes unnecessarily. We propose a new, bimodal cache, *Fear Of Missing Out* (FOMO), for managing non-datapath caches.

FOMO is a generalized non-datapath cache admission policy, managing which requests are passed along to the underlying cache replacement policy. Being generalized has the benefit of allowing any datapath cache replacement policy, such as LRU, ARC, or LIRS, to be augmented by FOMO to make these datapath caching algorithms better suited for non-datapath caches. FOMO is able to achieve this by taking into account that most storage workloads go through a series of phases during their execution, with access behavior varying significantly across these phases. Operating in two states, FOMO is selective — it selectively disables cache insertion and replacement depending on the learned *behavior* of the workload. FOMO is lightweight and tracks inexpensive metrics in order to identify these workload *behaviors* effectively. FOMO focuses on two metrics: the traditional *cache hit rate*, but also the *reuse rate of recently missed items*, which evaluates how the items outside of the cache are being accessed. FOMO utilizes its Miss-History structure to collect the *reuse rate of recently missed items*, which allows FOMO to understand the workload behavior much more acutely than mARC. This insight allows FOMO to identify *opportunities* for improving cache performance that would have oth-

erwise have been missed. FOMO adjusts its behavior quickly and dynamically based on the performance of the cache and the observed behavior of the workload.

FOMO is evaluated using 3 different cache replacement policies against the current state-of-the-art non-datapath caching algorithms, using 5 different storage system workload repositories (totaling 176 workloads) for 6 different cache size configurations, each sized as a percentage of each workload's footprint. Just as mARC improved upon LARC by writing to the cache more often to improve hit rates, FOMO similarly improves upon mARC, achieving improved hit rate and write rate consistency by using a simple model to avoid the issues present within mARC's state model. The use of a Miss-History gives FOMO a more acute understanding of the workload behavior, compared to mARC, which depends exclusively on cache hit rate to make decisions.

5.1 Introduction

Conventional caching algorithms (e.g LRU [DT90], Clock [Tan07], FIFO [DT90], ARC [MM03], MQ [ZPL01], LIRS [JZ02], etc.) were designed for *datapath caches*. These *datapath caches* are defined by their requirement whereby every cache miss requires that a cache **insertion** occurs and, should that cache be full, a cache **eviction** is also incurred. While this is suitable for CPU and DRAM caches, where the cost of each **insertion** is relatively inexpensive, this no longer holds true in the context of host-side (flash or persistent memory) SSD caches.

When using host-side SSD caches [Lev08, BLM$^+$12, SSZ12], data can get served to the application directly from back-end storage without having to first retrieve it into the cache. This opens up a unique opportunity: *selective updates*, whereby cache updates are not always made upon a cache miss but instead get decided on every access. Caches that allow for this flexibility are *non-datapath caches*. Non-datapath caching algorithms with *selective updates*, can improve the lifetime of cache devices that wear out on account

of device-level writes [BD10, YKH+20]. These algorithms, also have the ability to improve cache hit rate by avoiding cache replacement in cases wherein the evicted item is considered more valuable than the inserted item [CDS+16, ZKAV20].

Prior research has demonstrated that state-of-the-art caching algorithms such as ARC can compromise both the cache hit rate as well as the cache write rate when applied to non-datapath caches [SLK+15, HWC+13]. In particular, the work of Santana *et al.* [SLK+15] demonstrated that workloads are heterogeneous and manifest different characteristics in different phases of their execution. They posited that the existing LARC caching algorithm was insufficient because it always prevented items from entering the cache on their first access. They proposed mARC [SLK+15], a non-datapath caching algorithm that responds to workload phases by either turning *on* or *off* a filter that prevents items from entering an ARC-managed cache upon first access. However, it is not clear whether mARC would generalize to cache replacement policies other than ARC (e.g., LIRS [JZ02]), some of which have been found valuable in addressing the wide array of storage workload types in production. Furthermore, mARC relied on a set of workload-sensitive constants (totaling 8 in all) that defined cache behavior; we demonstrate that this compromises mARC's ability to adapt to the variations within and across workloads.

Since LARC and mARC, other efforts in online non-datapath caching algorithms has been made. Two such examples would be Reinforced Learning Cache (RL-Cache) [KSGS19] and Learning From OPT (LFO) [Ber18]. RL-Cache is a non-datapath caching algorithm that uses reinforcement learning to figure out whether to cache or not cache items. RL-Cache requires the use and availability of several threads and GPUs in order to achieve its results. Furthermore, RL-Cache is primarily designed for variable sized object caches, as RL-Cache also attempts to reduce the number of objects within its cache. LFO is a non-datapath caching algorithm that uses machine learning supervised by a modified version of OPT to learn whether to cache an item or not based on what it

thinks OPT would do. While LFO does well, it uses TensorFlow and has large amounts of overhead that makes it inappropriate for consideration for production environments. In particular, these environments run workloads with a lot of dynamic behavior and learning needs to be lightweight, online, and continuous.

In this paper, we propose the *Fear of Missing Out* (FOMO) admission policy, a generalized non-datapath admission policy that can be used to augment any datapath cache replacement policy. FOMO augments these datapath cache replacement policies to become better suited for non-datapath caches. FOMO exists in one of two possible states at any given time — *Insert* or *Filter*. When in the *Insert* state, FOMO forwards all cache requests to the underlying cache replacement policy while observing the behaviors of both the cache and the workload. When in the *Filter* state, FOMO selectively disables cache updates to improve both cache update rate (cache writes) and the cache hit rate by preventing cache pollution. Cache pollution is the insertion of items into the cache whose ultimate value is less than that of the item evicted from the cache, hence the cache is polluted with worse items.

FOMO decides which state to be in by utilizing information about accesses to items that were in the cache or were recent cache misses. To determine the reuse of recent cache misses, FOMO maintains a Miss-History structure that keeps track of these items and their reuse. Using small periods of observation to come to these decisions, FOMO is capable of reacting quickly to changes in workload behavior. This reaction speed is important for FOMO. FOMO doesn't want to *miss out* on inserting items into the cache for future hits nor does it want to *miss out* on preserving the items in the cache or preventing items lacking reuse from entering into the cache. So, just as someone wishing to keep up with the latest trends, FOMO doesn't want to *miss out* on reacting to the latest workload behavior.

We evaluate FOMO using 3 different eviction policies (LRU, ARC, and LIRS) against other non-datapath caching algorithms on a workload collection comprising of 176 workloads sourced from 5 different storage system workload repositories. For each workload, we evaluate against 6 different cache size configurations, each sized as a percentage of each workload's footprint, defined as the set of unique items accessed by the workload. Just as mARC improved upon LARC by writing to the cache more often to improve hit rates, FOMO similarly improves upon mARC, achieving improved hit rate consistency while reducing writes significantly by using a simple model to avoid the issues present within mARC's state model. The use of a Miss-History gives FOMO a more acute understanding of the workload behavior, compared to mARC, which depends exclusively on cache hit rate to make its decisions.

5.2 Background and Motivation

The most popular caching algorithms in the literature are LRU, LFU, LIRS, and ARC. These approaches attempt to cache items considered important based on different metrics such as recency, frequency and reuse distance. However, they are all datapath caching algorithms and therefore always make cache updates on a cache miss. When these datapath cache replacement algorithms are used on *non-datapath caches*, such as host-side (flash or persistent memory) SSD caches, the amount of cache updates incurred wears out the cache at a alarming rate. Not only that, but these datapath cache replacement algorithms do not utilize a new option possible with these host-side SSD caches: *selective updates*, where cache updates are not always made upon a cache miss but instead are decided on every access. This selectiveness allows non-datapath caching algorithms to improve the lifetime of cache devices that wear out from device-level writes, and improve cache hit

rates by avoiding cache replacement in cases wherein the evicted item is considered more valuable than the inserted item.

5.2.1 Non-datapath caching algorithms

Non-datapath caching algorithms have received attention in the last few years, starting with the LARC [HWC+13] work.

Lazy adaptive replacement cache (LARC): LARC is a non-datapath solution that focuses on reducing forced updates to a non-datapath cache. At a high-level, LARC prevents inserting items not found in the cache in case they have not been accessed sufficiently recently. LARC consists of two LRU lists, one for cached items and a first-in-first-out *filter list* for tracking non-cached items that have been accessed recently. A cache update is only performed when an item that is not found in the cache is found in the filter list. To control how aggressively items are filtered, the maximum filter list size grows (*resp.* shrinks) with a decrease (*resp.* increase) in cache hit rate. While LARC's filtering approach is straightforward, its filter is always operational and, as a result, can prevent important items from entering the cache in a timely fashion. In particular, LARC populates the cache at least twice as slowly as most other algorithms when workload working sets change; this behavior is capable of significantly impacting performance due to the compulsory misses induced by the filter. In addition, LARC considers requests as independent, and makes filtering decisions on an individual basis, missing opportunities that can be afforded by finding patterns within the workload.

Multimodal adaptive replacement cache (mARC): mARC [SLK+15] is a selective caching algorithm that improves performance and endurance by using cache *hit rate* as a metric to selectively turn on/off cache insertions. mARC addresses the limitations of LARC by monitoring and identifying changes in the workload's working set and caching

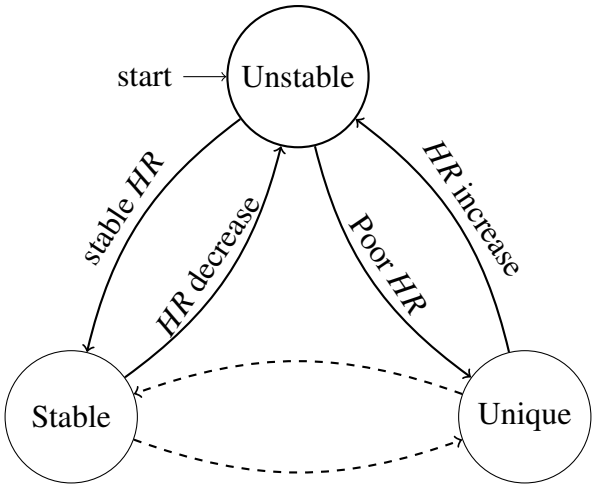

Figure 5.1: Simplified state diagram for mARC. The missing transitions between the Stable and Unique access states are shown using dashed edges. (*HR* refers to the running average Hit-Rate)

the important items in response to these changes. To make this possible, mARC defines three distinct *workload states*: *unstable, stable,* and *unique access* and reflects these states within the mARC state model. In the *unstable* state, the relative importance of items is changing and mARC enables cache insertions. In the *stable* state, the relative importance of items remains about the same and mARC disables cache insertions. In the *unique* state, there is a significant increase in the number of one time accesses — resulting from either streaming or random access — and mARC disables cache insertions. mARC bases its state detection mechanisms by observing a single system metric, the cache hit rate. Figure 5.1 defines a simplified version of how state detection occurs in mARC.

While mARC is able to utilize its knowledge about workload states to allow updates or not in the cache, it has important limitations. First, since it does not account for the *stable* $\rightarrow$ *unique* (and vice-versa) state transitions, it is unable to handle certain workloads. Without these direct transitions, *stable* $\rightarrow$ *unique* becomes *stable* $\rightarrow$ *unstable* $\rightarrow$ *unique*, incurring many cache insertions while in the *unstable* state prior to finally transitioning to *unique access*. The time within the *unstable* state pollutes the cache with items with no

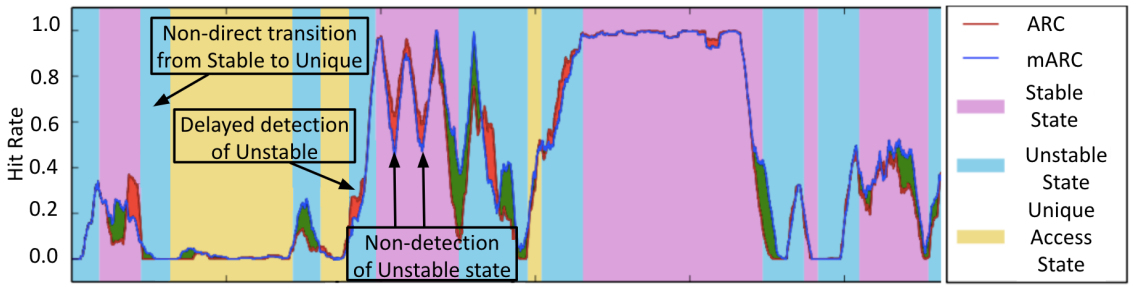

Figure 5.2: Performance comparison using mARC and ARC over time for MSR workload Proj3. Presented are various moments where mARC isn't performing or deciding as well as it could.

reuse. Second, mARC fixes the values of various parameters that compromises its ability to adapt and recognize workload states for arbitrary workloads. The complexity of mARC could allow for it to be tuned for specific workloads, but in the end also prevents it from being properly generalized for all workloads. Thirdly, mARC relies on cache-hits in order to understand the workload behavior, which leaves mARC not analyzing the workload itself, but rather the effects of the workload on the cache. This method of analysis is slow, requiring the item to *first* be inserted into the cache. Workload analysis based on cache-hits are also strongly effected by the cache replacement algorithm, which for mARC would be ARC. Finally, although mARC is able to detect workload state changes, it often does so with significant delay, with mARC detecting state changes only after accesses totalling the cache size are observed. Figure 5.2 shows a missing transition from *stable* to *unique*. mARC also has a delayed reaction to identifying changes in the workload states which can deteriorate the performance due to poor static parameter choices.

5.2.2 The Fear of Missing Out

Non-datapath caching algorithms can experience two potential pitfalls. On the one hand, disabling cache insertions when the workload changes can compromise cache hit rate. On the other hand, allowing cache insertions when the workload is unchanged can lead

to unwanted cache writes. Recent non-datapath caching algorithms [HWC+13, SLK+15] employ limited ability to handle changes in workload behavior. LARC always filters items on first access. While there is significant reduction to cache updates, LARC compromises the algorithms responsiveness to changing workload behaviors. mARC, on the other hand, simply monitors the stability of the cache hit rate to evaluate workload behavior. Unfortunately, cache hit rate does not properly capture workload behavior, but rather how well the cache is prioritizing items in the working set. Non-datapath caching algorithms, owing to the available flexibility of not having to perform cache updates, invariably embody a *fear of missing out* on responding in a timely fashion to changes in workload behavior.

Classic caching solutions are *reactive* and this impedes their ability to react to workload behavior changes. Many incorporate the notion of eviction history to evaluate the importance of an item [MM03, JZ02, Li18]. The non-datapath caching algorithm, mARC [SLK+15], via its internal ARC mechanism, also inherits this approach. Unfortunately, reacting as a consequence of accesses to items that were evicted requires that items must first enter the cache. Furthermore, if the number of cache hits is not increasing, only having information about the evicted items can obscure the reason for low performance with a limited view of the workload. To respond to workload changes quickly, observing accesses to newly requested items are crucial. In particular, keeping track of recent accesses that resulted in cache misses allows us to understand what the cache is "missing out" on. Here, timely knowledge of workload behavior helps improve the accuracy of the filtering mechanism. We assert that the rate of access to recent cache missed items provides crucial, complementary information about the workload that allows for better understanding of short term workload behaviors. FOMO improves upon the qualities of mARC, just as mARC had with LARC. FOMO is a general non-datapath admission policy that is capable of improving hit rate and write rate consistency by utiliz-

ing a simple model that avoids the issues present within the mARC state model. FOMO's Miss-History presents a more acute understanding of the workload behavior through the reuse of recently missed items, which doesn't require the items to be inserted into the cache *first*, unlike mARC, which depends exclusively on the observed cache hit rate. As we shall show in the next section, the design of FOMO focuses on comparing the cache hit rates and rate of reuse of recent cache misses to better understand the general workload behavior.

5.3 Design

Non-datapath algorithms such as LARC can be counter-productive to the hit rate as they do not cache an item until it is reused, incurring a compulsory additional cache miss per item. Additionally, when working sets change frequently, the requirement of proof of reuse can significantly impair hit rates. In the case of mARC, its high level of complexity comes not from using three states, but rather its state transitions. mARC's three states (*unstable*, *unique access*, and *stable*) were meant to align to the satisfied workload behavior. However, in order to do so, mARC employed multiple conditions to be observed before a state transition is allowed. In total, mARC has seven conditions that dictates all the state transitions. Some of these conditions can be considered redundant since they address multiple aspects of workload behavior from the context of the cache hit rate primarily. However, even with these conditions, mARC still excluded the direct state transitions between the Unique Access and Stable states. With this missing transition, mARC was designed to enter the Unstable state as a way to "safely" learn about a changing workload before deciding on either the *Unique Access* or *Stable* state. This method however imposes many unnecessary writes and may cause the removal of "soon-to-be-hit" items from the cache in exchange for "one-hit wonders".

Figure 5.3: Example of FOMO working in Insert state handling the request stream: X, Y, X. (a) Shows the starting state of the cache and Miss History. (b) Has X missing and being inserted into both the cache and the Miss History. (c) Has Y missing and being inserted into both the cache and the Miss History. (d) Has X hitting in the cache and being removed from the Miss History.

Figure 5.4: Example of FOMO working in Filter state handling the request stream: X, Y, X. (a) Shows the starting state of the cache and Miss History. (b) Has X missing and being inserted only in the Miss History. (c) Has Y missing and being inserted only in the Miss History. (d) Has X miss in the cache, but hit in the Miss History. Due to X showing reuse, it is then inserted into the cache and removed from the Miss History.

The following section explains the design of *Fear of Missing Out* (FOMO) that is guided, in principle, by the issues explained above. FOMO is designed to be a generic non-datapath admission policy that could augment any datapath cache replacement algorithm and still provide the write reductions and performance that is expected from non-datapath caches. FOMO incorporates a simple, two state design: *Insert* and *Filter*.

5.3.1 Miss-History

FOMO's Miss-History is an LRU structure that primarily holds recently missed items. For fairness against LARC and mARC, the Miss-History is limited to the size of the cache in a similar manner to that of the ghost lists (or filter lists) of its peers. Whenever FOMO encounters a cache miss, FOMO's Miss-History is updated. Should an item that causes a cache hit also exist in the Miss-History, this item will be removed from the Miss-History. When FOMO is in the *Insert* state, as shown in Figure 5.3, the Miss-History

is straightforward, as any item incurring a cache miss, including those already within the Miss-History, are moved to the MRU part of the Miss-History, removing the LRU item as needed. However, when FOMO is in the *Filter* state, as shown in Figure 5.4, FOMO will treat the Miss-History as a method to track the filtered items, similarly to LARC. When FOMO is in the *Filter* state, the Miss-History will treat items that incur a cache miss and exist within the Miss-History the same as a cache hit, removing the item from the Miss-History. Despite the differences in their function, FOMO in both states is utilizing the Miss-History to discover items with reuse, with the state as context. Since the *Insert* state has these items already entering the cache without knowing if they have reuse, the Miss-History is used to verify the reuse of the item *and* that the cache is seeing this reuse. With the *Filter* state, only the Miss-History is seeing this reuse, after which it passes the item along to the underlying cache replacement algorithm, believing that it will see further reuse. In the next section, we go into further detail about FOMO's states and how they are determined.

5.3.2 FOMO States

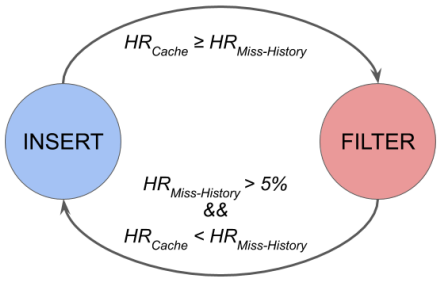

Figure 5.5: FOMO states and transition conditions.

FOMO incorporates a simple, two state design: *Insert* and *Filter*. This simplicity avoids the complexity of mARC's state transitions, while still encompassing the necessary actions when identifying workload behaviors. With these states, the compulsory misses

Algorithm 1 FOMO algorithm
$HR_C = HR_{Cache}$
$HR_{MH} = HR_{Miss-History}$

```
RecordHits(x)
time++
if time % period_size == 0: then
  if state == Insert: then
    if HR_C ≥ HR_MH: then
      state = Filter
    end if
  else
    if HR_MH > 5% and HR_C < HR_MH: then
      state = Insert
    end if
  end if
  CleanUp()
end if
if x in Cache: then
  Cache.Request(x)
  if x in MissHistory: then
    MissHistory.Remove(x)
  end if
else
  insert = state == Insert
  if x in MissHistory: then
    if state == Filter: then
      MissHistory.Remove(x)
      insert = True
    else
      MissHistory.MoveToFront(x)
    end if
  else
    if MissHistory.IsFull(): then
      MissHistory.RemoveLast()
    end if
    MissHistory.AddToFront(x)
  end if
  if insert: then
    Cache.Request(x)
  end if
end if
```

of LARC can be avoided while FOMO is in the *Insert* state and similarly benefit from the protection against cache pollution while FOMO is in the *Filter* state.

When FOMO is in the *Insert* state, all requests are passed to the underlying cache replacement algorithm, where it may cache it and choose what to evict, if needed. FOMO always begins in the *Insert* state to gather information while filling up the cache.

When FOMO is in the *Filter* state, FOMO decides whether or not a request will be passed to the underlying cache replacement algorithm. How FOMO decides this is similar to LARC: the request exists in the cache or reuse is observed for an item *not* in the cache. FOMO tracks this observable reuse with the Miss-History. Should FOMO encounter a cache miss that exists within the Miss-History while in the *Filter* state, it will observe that the item has reuse and pass it on to the cache to insert, as well as remove this item from the Miss-History. This removal from the Miss-History while in the *Filter* state is desirable mainly to avoid the possibilities of *cache churning* [SLK+15]. Cache churning occurs when the working set is a superset of the items in the cache, but the lowest frequency items in the cache and some of those outside the cache are similar. This has meant that the lowest frequency items in the cache are evicted to make way for the similarly (or lower) frequency items outside of the cache, introducing several misses that instead could be hits by protecting the items in the cache from eviction. Figures 5.3 and 5.4 show examples of how FOMO operates within these states and how the Miss-History is affected.

With these two states (*Insert* and *Filter*), the conditions to transition between them must be defined. With the aim to avoid the complexity that mARC had, two conditions that represented each state are defined: $HR_{cache} < HR_{Miss-History}$ and $HR_{cache} \geq HR_{Miss-History}$. The transition from *Filter* to *Insert* was determined to be advantageous where the reuse in the Miss-History was significant, or greater than the hits to the cache. The transition from *Insert* to *Filter* was determined to be advantageous where the reuse of the Miss-History was not significant. In order to calculate both HR_{cache} and

$HR_{Miss-History}$, the last period of requests are observed for hits/reuse, where the period's size is set to 1% of the cache size. This was set with the intention of having FOMO react swiftly to changes and protect the items in the cache.

It was quickly noticed that FOMO would at times encounter situations where both HR_{cache} and $HR_{Miss-History}$ were low and the slightest changes encouraged FOMO too strongly to transition from the *Filter* state to the *Insert* state. This led to an additional condition being added to the *Filter* to *Insert* transition: $HR_{Miss-History} > 5\%$. The addition of this condition prevented this edge-case scenario from improperly using the reuse of a few items in the Miss-History as justification to change state to the *Insert* state. The final state design can be seen in Figure 5.5, while the finalized algorithm can be seen in Algorithm 1.

5.3.3 Overheads

FOMO, augmenting the underlying caching algorithm, adds its own overheads on top of the caching algorithm. As such, FOMO is designed to keep its own overheads low while achieving its goals. All of FOMO's operations are achievable in $O(1)$ time complexity. In terms of space overhead, FOMO's only large requirement is an array of around *cache size* entries large enough to serve as a hash table. Each entry of the array consists of an integer to track the block address and two pointers (next and prev) for its place in the Miss-History.

Should FOMO become more integrated with any algorithm, the hash tables may be merged, along with the entries, to possibly reduce the space overhead by eliminating redundancy. Furthermore, some space and time overhead may be removed by altering FOMO to use a CLOCK or FIFO structure for its Miss-History instead of an LRU list.

5.4 Evaluation

Dataset	# Traces	Details
FIU [Sto, KR10]	10	End user/ developer home directories; Web server for faculty/staff/students; Apache web-server for research projects; Web interface for the mail server; Online course management system
MSR [Sto, NDT+08]	36	User home directories; Hardware monitoring; Source control; Web staging; Test web server
CloudVPS [AZ14]	18	VMs for cloud provider
CloudCache [AZ14]	6	Online course management website; Web server for a CS department user webpages
CloudPhysics [WPGA15]	106	VMware VM block traces. Due to the size of many of the traces within this set, only the first day of each trace was used for tests.

Table 5.1: Sources and descriptions for the 5 storage data sets used in this paper.

Algorithms State-of-the-art non-datapath caching algorithms LARC and mARC are compared against FOMO using LRU, ARC, and LIRS as underlying cache replacement algorithms. Since FOMO was evaluated with LRU, ARC, and LIRS as their underlying cache replacement algorithms, these algorithms are also included to demonstrate FOMO's benefits.

Experimental Setup We built cache simulators for every algorithm that can process block I/O workloads and report on the number of reads, writes, cache hits and misses, and total writes incurred. When possible, we used the original authors' version of the algorithm implementation. To simulate sufficient cache space as well as I/O activity, the size of the cache was set to be a fraction of the *workload footprint*, defined as the total size of all unique data accessed. This fractional cache size was varied from 1% to 20% of each workload's footprint in our simulations (more specifically 1%, 2%, 5%, 10%, 15%, and 20%).

Workloads To have a large, diverse set of I/O workloads, the workloads used for testing include FIU, MSR Cambridge, CloudCache, CloudVPS, and CloudPhysics block I/O workloads as detailed in Table 5.1 [Sto]. The FIU, MSR Cambridge, CloudCache and CloudVPS workloads are all run for their *full duration*, or for the full length of the workloads, with some requiring the merging of individual days into one large, continuous workload. Tests using CloudPhysics workloads are limited to the first day of the workloads in order to reduce the length of the workload and workload footprint so that these are runnable on the available resources within a reasonabe amount of time.

Metrics As previously mentioned, the cache simulators collect information about read, writes, cache hits and misses, and total writes incurred. FOMO aims to not only improve upon its underlying cache replacement algorithms, but also be comparatively better than its non-datapath cache algorithm peers. For the evaluation, metrics include the mean cache hit rate and the mean write rate; with greater write rates indicate lower flash cache device lifetime. To evaluate cache performance, the normalized results are compared across workloads and cache sizes to understand how consistent the performance of FOMO is. Normalized hit rates and write rates are computed for a given workload-cache size combination with respect to the best performing algorithm. This normalization method better shows the general performance of an algorithm amongst its peers. Furthermore, it provides a fair method of comparison, as a total average hit rate comparison alonge would hide how well (or poorly) an algorithm was performing compared to others. As the write rate is not best presented normalized, the write rate is changed to a static rate, the rate at which the cache experiences no changes. The best performer has the highest static rate, as this translates to the fewest writes/updates, which is one of the goals of non-datapath caching algorithms. In order to prevent small differences in metrics leading to large normalized differences, the results where the best performance outcome was less than 5% were not included.

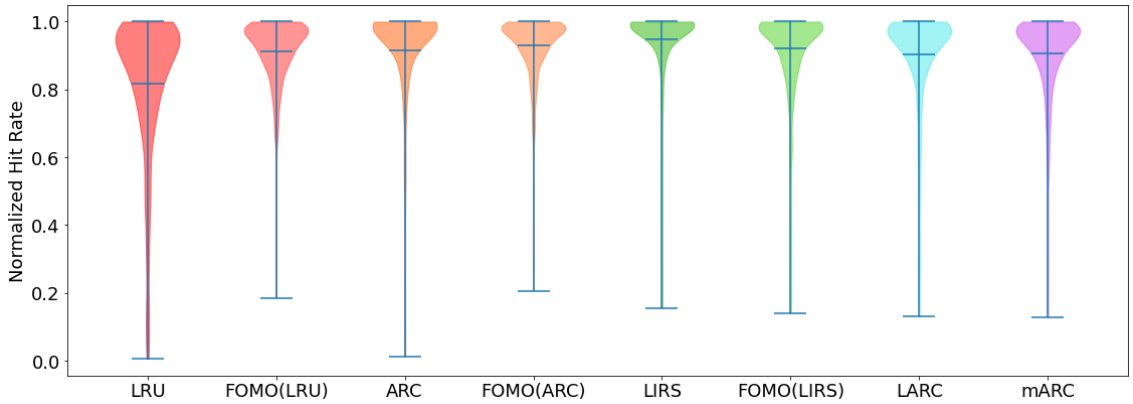

Figure 5.6: Normalized Hit Rate summary results in percentage, including results for all five different workload sources (FIU, MSR, CloudCache, CloudVPS, and CloudPhysics) measured for six different cache size configurations as a fraction of the workload footprint for each algorithm. Each violin plot also has lines to indicate the max, mean, and min of each algorithm's normalized performance. Here it can be seen that FOMO(LRU) and FOMO(ARC) have a notably higher minimum normalized performance across the range of workloads and cache sizes.

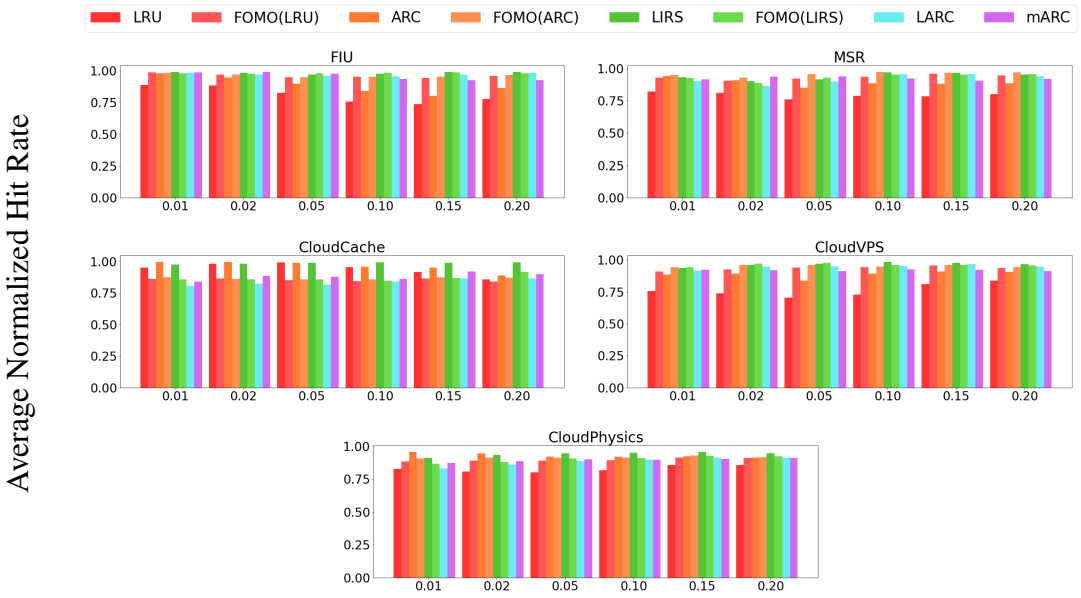

Figure 5.7: Average Normalized Hit Rate results in percentage for the five different workload sources (FIU, MSR, CloudCache, CloudVPS, and CloudPhysics) measured for six different cache size configurations as a fraction of the workload footprint. Notably, none of the non-datapath caching algorithms performs well for the CloudCache workloads. FOMO(LRU) and FOMO(ARC) show that FOMO is typically able to improve their underlying cache replacement algorithm's performance.

Each measurable aspect of FOMO is given its own section. First, we discuss both of the ways FOMO has consistent performance: hit rate and write rate. Of the two, hit rate is evaluated first due to its general importance to cache performance, then write rate, which is of great importance for non-datapath caches. Second is a discussion of the impact that the generality of FOMO's design has on the performance improvements FOMO can offer when augmenting datapath cache replacement algorithms. Third, we analyzed how FOMO's Miss-History is able to detect patterns of reuse within the workload, using a particular workload for a case study. Finally, we discuss the *adversarial workloads* to FOMO and the other non-datapath caching algorithms.

5.4.1 Consistency

The focus of our evaluation is FOMO's consistency in both hit rate and write rate performance. This focus on consistency is for one primary reason: since neither always updating nor always filtering is correct, FOMO must be able to balance the two options effectively. To clarify, there exists within the many workloads and cache sizes evaluated, tests where the penalty for being exclusively updating or exclusively filtering is a significant hinderance. As such, FOMO, making decisions to update or filter, has a responsibility to avoid these penalties as best it can. Hence, this paper's measure of success for performance is focused on consistent hit rates and consistent write rates, with reasonable or improved averages compared to its peers.

To measure consistency of both hit rate and write rate performance, normalized metrics are used to draw fair comparisons. This normalization is with respect to the best performer for a workload at a given cache size, and not with respect to the results for any particular algorithm. Results are presented both as a violin plot of all normalized results as well as bar graphs which break down the results by workload and cache size.

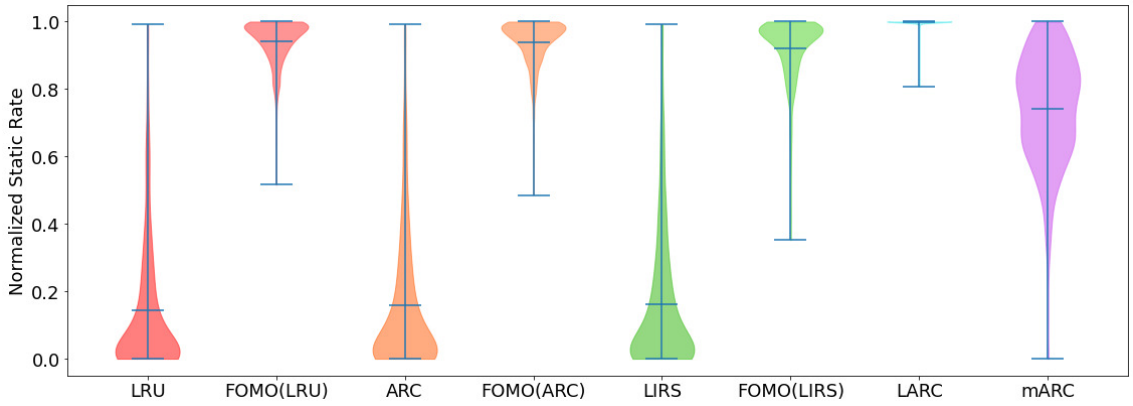

Figure 5.8: Normalized Static Rate summary results in percentage, including results for all five different workload sources (FIU, MSR, CloudCache, CloudVPS, and CloudPhysics) measured for six different cache size configurations as a % of the workload footprint for each algorithm. LARC is indisputably the algorithm with the fewest cache updates. mARC aimed to write more for performance gains, but incurs a great deal more than that of LARC and FOMO, even to levels matching a datapath caching algorithm. Notably, FOMO significantly reduces the number of cache updates of its underlying cache replacement algorithms to levels similar to that of LARC.

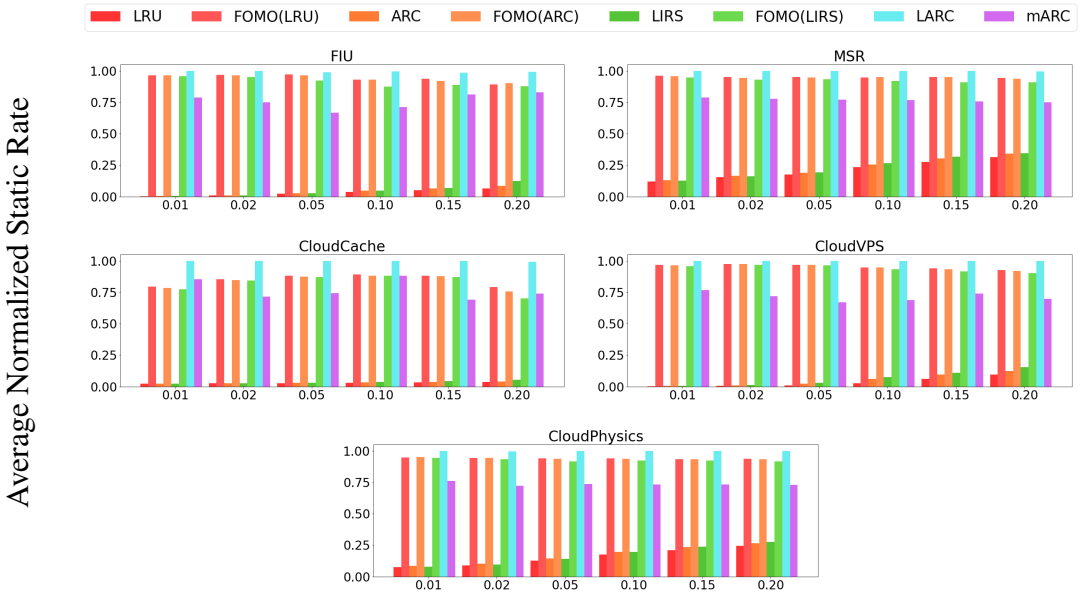

Figure 5.9: Average Normalized Static Rate results in percentage for the five different workload sources (FIU, MSR, CloudCache, CloudVPS, and CloudPhysics) measured for six different cache size configurations as a % of the workload footprint. As noted in Figure 5.8, LARC is consistently having the fewest cache updates, typically followed by FOMO, then mARC.

Consistency is best seen in the distribution of results presented using violin plots, with narrower distributions indicating a greater degree of consistency in comparative performance. Since normalized results where the best performance is still poor can have huge normalized differences from a small absolute value difference, any results where the best performing candidates did not exceed 5% in their target metric were excluded.

Hit Rate

Despite the early focus of non-datapath caching algorithms on the reduction of writes, hit rate performance continues to be of vital importance for many applications that utilize such caches. This results in a summary of each algorithm's hit rate performance, shown through violin graphs in Figure 5.6, where it can be seen that a good degree of narrowing of the algorithm's normalized hit rate distribution contributing to improved consistency in performance has occurred. Both FOMO's normalized average performance (shown by the middle line in Figure 5.6) and normalized minimum performance (shown by the bottom line in Figure 5.6) show clear improvement from their underlying cache replacement algorithms. In fact, FOMO's normalized minimum performance is substantially higher than that of the other algorithms tested. Excluding FOMO(LIRS), FOMO has a significantly higher minimum normalized hit rate (FOMO(LRU) (18.56%) and FOMO(ARC) (20.62%)) compared to the other non-datapath algorithms(LARC (12.94%) and mARC (12.82%)), with the next closest being that of LIRS (15.44%). This improvement upon the normalized minimum performance is confirmation that FOMO is adjusting well to the workloads and adapting to the worst-case scenarios better than other, stricter algorithms. One such case will be discussed in greater detail in Section 5.4.3.

Datapath cache replacement algorithms augmented with FOMO show clear improvement when compared with their unaugmented counterparts, as seen in Figure 5.7. This improvement for normalized minimum hit rate indicates that FOMO is capable of adapt-

ing to worst-case scenarios better compared to the existing non-datapath cache replacement policies.

Write Rate

While it is expected that all of the non-datapath caching algorithms would have significant improvements over their datapath counterparts, its is clear that LARC, by always filtering, would see the largest reduction in writes overall. As with the hit rate analysis, the write rate results are normalized, with write rate being the number of requests that incurred a write/update (cache insertion or write hit) over all requests. However, using write rate itself would result in the "best" performer having the lowest value, which does not have a preferred presentation of how well algorithms are doing compared to the best. To be able to present normalized results, we define a metric called *static rate* is used for our normalized analysis. Just as hit rate is the opposite of miss rate, static rate is the opposite of write rate, where write rate measures the rate of cache updates, the static rate measures the rate of not updating the cache. This means that results with a higher static rate have the fewest writes made to the cache, and therefore indicate longer cache device lifetime.

Figure 5.8 and Figure 5.9 both show FOMO strongly improving the normalized static rate of their underlying datapath cache replacement policy to levels similar to that of LARC, with LARC having the best static rate performance, as expected. These results show FOMO drastically improving the static rate performance of their underlying datapath cache replacement policies to the levels expected to that for non-datapath caching algorithms that aim to drastically reduce writes to the cache. Interestingly, the normalized static rate performance of mARC has high variance (as seen in Figure 5.8), in the worst case being similar to a datapath caching algorithm. This distribution in mARC's performance is attributed to mARC's *Unstable* state. mARC always transitions to or from the

Unstable state and, when transitioning to it, remains in the *Unstable* state for a significant amount of time before its next chance to change state.

5.4.2 Generality

The design of FOMO is focused on being as generalized as possible, with FOMO treating the underlying cache replacement algorithm as a black box. This design leads FOMO to act as the admission policy, filtering when needed, and have the underlying cache replacement algorithm responsible for managing the items in the cache. This control of admission to the cache, with the aid of the Miss-History, which is generalized and primarily acts the same regardless of the underlying cache replacement algorithm, is a robust technique for improving both hit rate and write rate of these algorithms when augmented with FOMO. This approach lets FOMO augment any datapath cache replacement algorithm. Within our tests, FOMO augmented LRU, ARC, and LIRS to much success, as FOMO was able to have results that align strongly with those expected of non-datapath caching algorithms.

FOMO improves upon the results of the underlying cache replacement algorithm for every workload and cache size, as seen in Figure 5.7, with the exception of CloudCache. The CloudCache workloads did not appear to have any of the non-datapath caches perform well. The investigation into why this is can be found in Section 5.4.4.

Excluding FOMO(LIRS), the average normalized hit rate of underlying cache replacement algorithm improved overall when augmented with FOMO (LRU (81.79%), FOMO(LRU) (91.06%), ARC (91.59%), FOMO(ARC) (92.85%), LIRS (94.83%), FOMO(LIRS) (91.96%)). Additionally, FOMO achieves an average normalized hit rate above other non-datapath caching algorithms (LARC (90.17%) and mARC (90.65%)), regardless of the underlying cache replacement algorithm. Ultimately though,

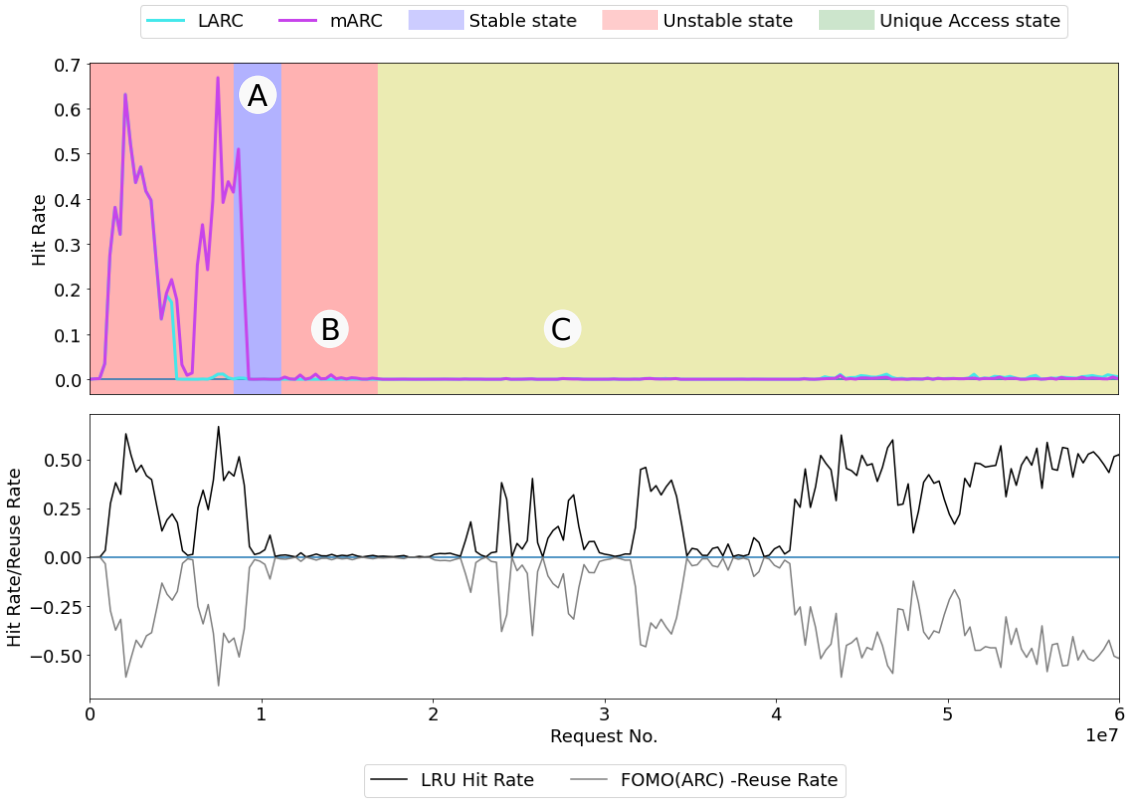

Figure 5.10: (a) The hit rates of both LARC and mARC for the CloudPhysics workload w54_vscsi2.itrace, in which LARC and mARC do not perform well. The background has colors that indicate the state of mARC where red=Unstable, blue=Stable, and green=Unique Access. LARC stops seeing a significant amount of hits after it fills the cache. mARC stops seeing a significant amount of hits after it switches to the *Stable* state Ⓐ. mARC incurs unnecessary writes by transitioning to the Unstable state before transitioning to the Unique Access state Ⓑ. mARC doesn't see any cache hit rate activity and therefore cannot find a reason to change state Ⓒ, even though plenty of opportunities for cache hits exists, as can be seen in Figure 5.10b.
(b) The cache hit rates of LRU and the reuse rates of FOMO's Miss-History for the CloudPhysics workload w54_vscsi2.itrace. The caching algorithm that FOMO augments does not matter here, as the focus is on the Miss-History. To be able to see both hit rates more clearly, the hit rate of the Miss-History has been mirrored over the horizontal axis (negated). This, compounded with the hit rates of LARC and mARC, show both that LRU is capable of having hits and that FOMO's Miss-History is seeing the same hits. Taken together, these plots demonstrate that the workload is mostly composed of items limited to second accesses. As such, both LARC and mARC (when acting similar to LARC) are only caching items when they have the second access, but do not get any benefit in doing so.

FOMO(LIRS) does bring LIRS in line with expectations for non-datapath caching algorithms by not being a significant hindrance to its hit rate performance while drastically improving LIRS's write rate performance, as seen in the next section.

The reason for the hindered hit rate of FOMO(LIRS), as seen in Figure 5.6, comes from the design of LIRS more than the design of FOMO, as LIRS itself already has a built-in filtering mechanism: the Q stack. This stack, while aggressively giving newly inserted items a small time frame to be hit in order to be moved to LIRS' S stack, works to filter further the items passed to LIRS by FOMO, which utilizes its own filter. This double filtering leads to situations where FOMO reasons about items that are passed to LIRS, but are evicted from LIRS prior to being hit, decreasing performance. Despite this, FOMO is still helpful when using LIRS for a non-datapath cache, as it still greatly improves the write rate, as seen in Figure 5.8. The implications of this result and understanding indicate that datapath cache replacement algorithms that feature aggressive filtering mechanisms, like those of LIRS, may have degraded hit rate performance when augmented with FOMO while still seeing significant write rate improvements. Overall, FOMO offers generality for any datapath caching algorithm.

5.4.3 FOMO's Miss-History: A Case Study

FOMO's hit rate and write rate results show the benefit of FOMO's design. However, the structure central to FOMO's understanding of the workload: *the Miss-History* has not been analyzed for effectiveness. We present a case where the Miss-History reveals a pattern that FOMO takes advantage of, but neither LARC nor mARC do — the first 60 million requests of the CloudPhysics workload w54_vscsi2.itrace with a cache size of 1% of the number of unique addresses requested. Some of the previously noted shortcomings of mARC are present within Figure 5.10. These particular shortcomings are the lack of a

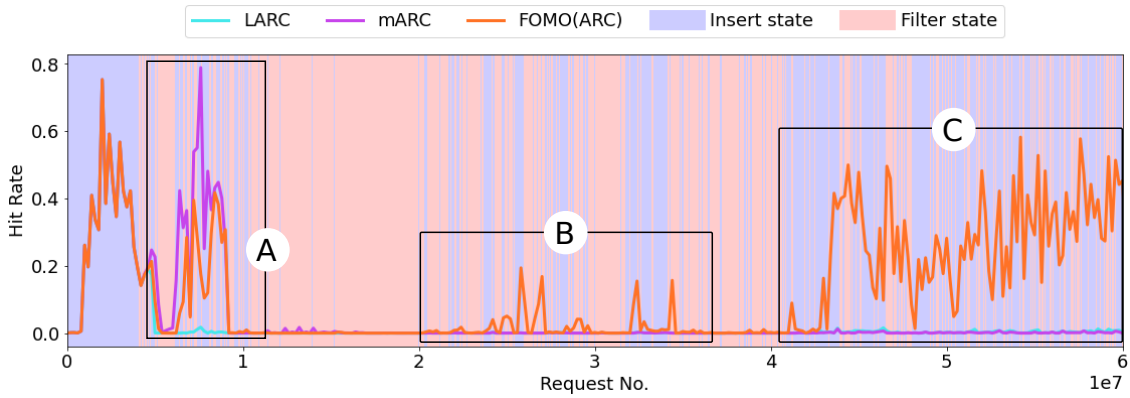

Figure 5.11: The hit rates of LARC, mARC, and FOMO(ARC) for the CloudPhysics workload w54_vscsi2.itrace. FOMO(ARC) is shown as it performs the worst among FOMO(LRU) and FOMO(ARC) in this instance. The background is colored with the state of FOMO, where red=Filter and blue=Insert. As FOMO state switches, it adapts to the workload for the chance to improve the hit rate and is able to achieve much more than both LARC and mARC due to FOMO's Miss-History. Ⓐ FOMO(ARC) started filtering prior to mARC, missing out a some hits. Ⓑ FOMO(ARC)'s changing states captures some opportunities for hits by switching to the Insert state quickly. Ⓒ FOMO(ARC) is able to recognize a pattern of reuse and is able to promptly respond and have many cache hits that LARC and mARC instead miss.

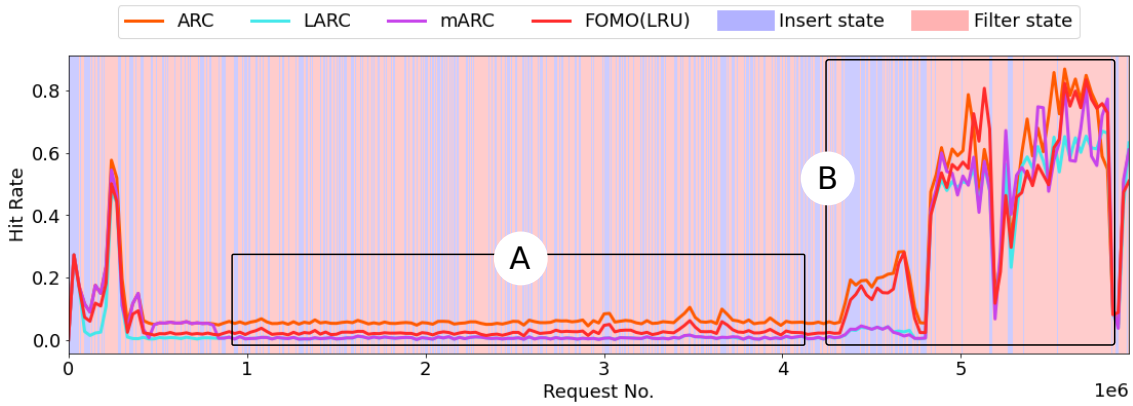

Figure 5.12: Hit rate plot of CloudCache workload webserver-2012-11-22-1.blk, focusing on ARC as the datapath caching algorithm to compare the performance of the non-datapath caching algorithm to (LARC, mARC, and FOMO(LRU)). The background colors correspond to the state of FOMO(LRU) at the time, with blue=Insert and red=Filter. From around one million to four million requests ARC is achieving hits that LARC, mARC and FOMO(LRU) aren't able to get, though FOMO(LRU) gets the most amongst the non-datapath caching algorithms, as seen at Ⓐ. Even as reuse ramps up at Ⓑ, FOMO(LRU) is able to achieve many more cache hits compared to LARC and mARC, while performing close to ARC. Afterwards, the algorithms perform similarly.

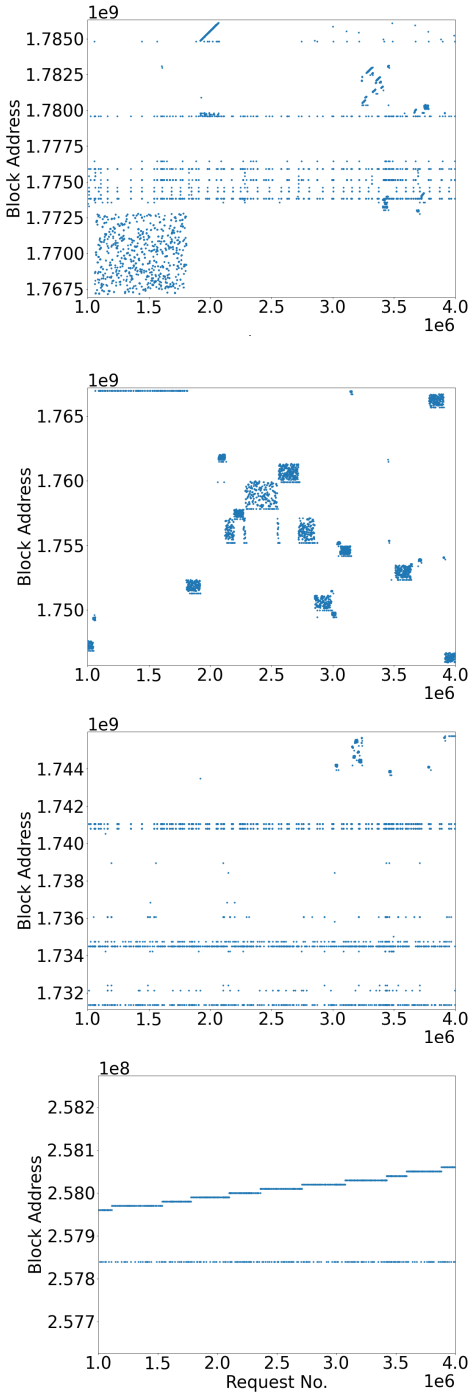

Figure 5.13: Address plots for CloudCache workload webserver-2012-11-22-1.blk that highlight the various patterns that could be seen simulataneously during the time period where non-datapath caching algorithms had poorer hit rate compared to their datapath counterparts. Among them we can notice scans, random accesses, and loops all occurring concurrently. These concurrent behaviors are why the non-datapath caching algorithms did not perform well.

direct transition between the *Stable* and the *Unique Access* states, incurring extra writes to the cache, and mARC's analysis of the workload using cache hit rate, which prevents it from noticing opportunities for hits.

What is interesting about this workload is that the majority of repeated accesses are limited to the second access of an item, with very few third or fourth accesses within a reasonable time frame. This can be seen in Figure 5.10, where the LRU hit rate mostly mirrors that of the Miss-History, which, in general, removes items from its structure on the item's second access.

FOMO is capable of finding patterns within the workload, and makes reasonable decisions based on these patterns, which can be seen in Figure 5.11. FOMO's decisions here capture many opportunities for hits that go unrecognized by both LARC and mARC, whose hit rate (and mARC's states) can be seen in Figure 5.10. This is unfortunate for mARC, which places emphasis on identifying workload states. mARC, due to its focus on using cache hit rate to identify workload state instead of something similar to FOMO's Miss-History, cannot see the pattern within this workload, leaving mARC to stay in the *Unique Access* state.

5.4.4 Adversarial Workloads

FOMO improves upon the results of the underlying cache replacement algorithm for every workload and cache size, as seen in Figure 5.7, with the exception of CloudCache. In fact, CloudCache workloads did not appear to have any of the non-datapath caches perform well compared to their datapath caching algorithm counterparts. When investigating the reason for this behavior, several things of note were observed, which will be highlighted with a focused discussion on one of these CloudCache workloads: webserver-2012-11-22-1.blk.

Webserver-2012-11-22-1.blk has an instance that shows a large time frame where non-datapath caching algorithms do not perform well compared to the datapath cache algorithms, shown in Figure 5.12. In particular, the time period between one million and four million requests has this behavior. When observing the block address access pattern plots of this CloudCache workload(Figure 5.13) it is noticeable that the workloads include several concurrent working sets and patterns (combinations of scans, random accesses, looping and repeated accesses). This mixing of patterns increases the likelihood of both FOMO and mARC observing reuse and discerning patterns in the workload based on them. This is why, as the cache size increases, the difference between the datapath and non-datapath caches begins to decrease.

Each of the non-datapath cache algorithms have their own reasons for why these access patterns were problematic. For LARC, which observes reuse on an individual basis, an item being reused often would not be reused again prior to being evicted, leading to many missed opportunities that the datapath cache algorithms can take advantage of. mARC, with its dependence on cache hit rate for decisions, with its use of the *Unstable* state as a intermediary transition between *Stable* and *Unique Access* states, and the long evaluation times, finds itself within the *Unstable* state more often and gains some level of advantage over both LARC and FOMO because of it. Lastly, FOMO, looking for patterns in the workload, would periodically find a pattern of reuse, change the state to *Insert*, get some cache hits that eventually overtake the reuse found in the workload, change state to *Filter* and so on repeatedly during such highly overlapping periods. Finally, we note that when particular patterns, or reuse in general, were more significant within these workloads, all of the non-datapath cache algorithms would identify and react to achieve cache hits.

5.5 Summary

We've demonstrated that FOMO is a viable non-datapath admission policy that is able to adapt to the specifics of workload's behaviors, choosing to admit an item into the cache or not based on FOMO's understanding of the workload's current behavior. Furthermore, FOMO has also demonstrated its capability to augment existing datapath caching algorithms to provide non-datapath cache benefits, doing so well enough to bring their performance close to that of existing non-datapath caching algorithms. Unfortunately, while FOMO is able to adapt faster than mARC, it still relies on a window of recent statistics in order to make decisions. The time period waiting for the statistics to be deemed ready and relevant in order to properly observe changes in the hit-rate and reuse-rate necessary for making a decision on which state of FOMO is appropriate for the the current behavior of the workload. Additionally, FOMO still has numerous tunables that have been set through testing; tunables that are a consequence of using rates and comparison of rates in order to make decisions for FOMO's state through the understanding of the workload behavior that these rates and comparisons provides. Hence, in an ongoing effort to remove tunables, a movement away from these methods that made mARC and FOMO work is necessary: foregoing rates and an internal state transition model, especially one relying on information gathered over a time period before making a decision. In Chapter 6, we explore how to understand workload states without rates or internal state transitions and develop a new non-datapath admission policy that is able to react quickly to changes in workload states: ANX.

CHAPTER 6

ANX

FOMO provides an adaptive, general, and workload state aware non-datapath admission policy, but is still hindered through limitations caused by its design. FOMO had improved upon mARC, with a new state model that consolidated mARC's lack of any transitions from the stable state to the unique access state, with a reduction of parameters that were tunable and set through testing, a reduction in the observation period's length, and a new rate that assists the understanding of the workload state. In particular, FOMO, taking an approach similar to that of mARC, relies upon rates such as the hit-rate to identify the workload state with confidence. The reliance on rates for state transitions has two main limitations: the potential to get stuck in a local minima, and the non-detection of behaviors by virtue of using rates over the entire observation period. In this section we propose a non-datapath caching algorithm called ANX that follows many of the same requirements of FOMO, but reconsiders the method used to identify the workload state as well as whether confidence should hinder the speed of identification. To do this ANX doesn't rely upon any rates, which require a collection of data over a period of time for confident decisions, but rather upon immediate behaviors of the workload and its relation to new and old pages. Furthermore, ANX utilizes uncertainty in its understanding of the workload to keep options available with a chance to occur based on ANX's level of certainty for an action. In other words, ANX measures certainty and utilizes randomness in order to avoid worst-case scenarios as much as possible, while additionally gaining multiple chances to react much more quickly to changes in the workload state compared to

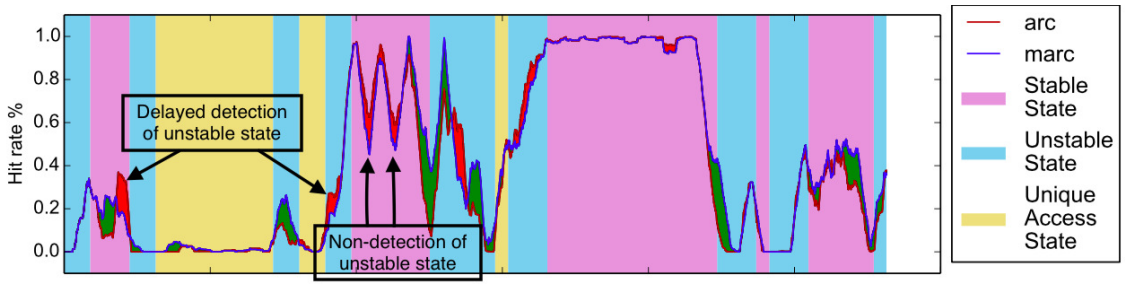

Figure 6.1: mARC State and ARC Hit-Rate Timeline for MSR Trace proj_3 for 1 Day

algorithms such as mARC and FOMO, which require more certainty before committing to decisions.

6.1 Introduction

The process of identifying the workload state reliably is difficult. The approach of taking a time period and analyzing it, perhaps in comparison with its predecessor, to determine the state of the workload is used by both mARC and FOMO. Their shared limitations include the use of tunable parameters that are set through testing, the use of an observation period in order to gather rate information for decision making, and the potential for them to be stuck in a local minima due to the non-detection of false negatives. Furthermore, mARC is missing a transition from the stable state to or from the unique access state. Some of the limitations of mARC are visible in the MSR trace of proj_3, detection of states is delayed or states are not detected at all, as shown in Figure 6.1. FOMO improved upon these flaws, reducing the number of states and state transitions to two and introducing a Miss History to have more a proactive approach to understanding the workload instead of being reactive to cache evictions. FOMO unfortunately still requires several tunables, with the most unfortunate being the window (an observation period) between decisions. This window, which scales with the cache size in FOMO, slows the potential reaction time of FOMO to changes in the workload state. We observed these problems and set several different goals as we began developing the next non-datapath solution: be capable

of moving directly between stable and unique access states, use as few parameters as possible, avoid requiring an observation period before reacting to state changes, and it should be able to correct itself quickly should it make any wrong decision. The proposed solution, ANX, satisfies all of these goals.

6.2 Workload States Revisited

While we had previously defined the workload states when developing mARC, we decided to revisit these definitions in order to find a new method of identification that would allow us to move away from windows such as those used to gather hit-rate information in mARC and FOMO. Presented next are the previous workload state definitions followed by our new definitions.

6.2.1 Stable

The stable state was previously defined as *a period of the workload where the working set frequency distribution is stable*. The way that mARC approached identifying this state using hit-rate was to check if the cache hit-rate itself was stable, or changed little, asserting that the working set itself must be stable to achieve this hit-rate stability. FOMO, while not directly using the stable state in its design, has it implicitly part of its methodology to filter when the hit-rate of the cache set is higher than that of the external set. To find a new definition, we re-evaluated what we were truly looking for and came to the simple conclusion that the stable state is when *old items are important*. In other words, identifying the important subset of a working set that has been seen previously and is being seen right now.

A subtype of the stable state, *churning*, defined as *a period of the workload where the working set frequency distribution is stable, but the working set is larger than the cache*.

The churn results in low frequency items being inserted into the cache then being evicted prior to having a cache hit. Problematically, when churning, the item being inserted encounters a similar scenario to the item it just replaced, creating a cycle of cache inserts and evictions that provide no benefit. This, in part, is the reasoning used to justify the activation of the filtering mechanism during moments of stable state in both mARC and FOMO. With this understanding, we realized churn can be identified with a new definition: *cache misses on new, recently inserted items*. As an aside, this behavior is strongly tied to the cache eviction policy rather than the workload state.

6.2.2 Unstable

The unstable state was previously defined as *a period of the workload where the working set frequency distribution is unstable, either having a slight or great change in the working set frequency distribution.* The way that mARC would identify this state using hit-rate was to see if the cache hit-rate was unstable, changing greatly between observation periods, asserting that the importance of items in the cache are changing and needs the internal cache to be activated in order to determine what items are important again. FOMO would implicitly find the unstable state when it observes a higher external hit-rate than the cache hit-rate. Upon its consideration, we came to the realization that an unstable state, was composed primarily of a new working set, and therefore the unstable state is when *new items are important*. By this, we are saying that the items in the cache are not as important and so it follows that we must cache these new items that are being requested.

6.2.3 Unique Access

The unique access state was previously defined as *a period of the workload where the frequency of the working set is either just one or is so low as to reasonably be consid-*

ered one. The way that mARC and FOMO identify this state using hit-rate is to see if the cache and external hit-rates are very low, asserting that if these hit-rates are so low, we must have a mostly unique access behavior occurring. To find a new definition, we attempted to determine what we were truly looking to identify for this state, coming to the conclusion that this state *has new items that are not reused within a given amount of time*. This definition makes sense, as many datapath caching algorithms such as ARC, LIRS, and even the non-datapath caching algorithm LARC, attempt to avoid unique accesses or low frequency items by giving new items a certain amount of time (often proportional to the size of the cache) to be reused before either being evicted or no longer being considered for insertion [MM03, JZ02, HWC+13]. However, in the interest of avoiding these windows and reacting quickly, we simply consider all new misses to be indicators of a unique access state, requiring the state and item to prove otherwise by having reuse later.

6.2.4 Individual Identifiers For Workload States

Looking at these new definitions, we can see two key words that repeatedly appear: *new* and *old*. Taking the next appropriate step, we developed a method for determining how to qualify items as new or old. The way we can determine items are *new* is whether we've filtered or inserted them recently, so we could just have a data structure to record a history of misses, or rather an LRU history of item misses. With this, we now have a way to determine if items are *old* or *new*, where *new items* are items found within the miss history, and *old items* are items found in the cache but not in the miss history.

By combining the information about whether a item is old or new and whether it was found in the cache or not, we establish a truth table to identify the workload state:

In Table 6.1, we can see that all possibilities are accounted for and how each is able to be an identifier for a workload state, with the included identifiers for impossible entries,

Hit (in cache)	New	Old	Workload State Identified
✗	✗	✗	Unique Access
✗	✗	✔	*Impossible*
✗	✔	✗	Unstable or Stable (Churning)
✗	✔	✔	*Impossible*
✔	✗	✔	Stable
✔	✔	✗	Unstable
✔	✔	✔	*Impossible*

Table 6.1: Workload State Identification using New and Old

these are situations that break the exclusivity of old and new or the definition of old by not being in the cache. However, we also see that one of these conditions identifies two possibilities: a miss on a new item could either be an indicator for an unstable or a stable state. Looking at the definitions, we see that unstable is focused only on new, but stable, in particular churning, is interested only in the items that have been *recently inserted*. So, let us revisit this truth table again, focusing on the new, or recently accessed items, identifying whether the item was recently filtered and/or inserted instead.

Hit	Inserted	Filtered	Workload State Identified
✗	✗	✗	Unique Access
✗	✗	✔	Unstable
✗	✔	✗	Stable (Churning)
✗	✔	✔	Stable (Churning)
✔	✗	✗	Stable
✔	✗	✔	*Impossible*
✔	✔	✗	Unstable
✔	✔	✔	Unstable

Table 6.2: Workload State Identification Truth Table used by ANX

In Table 6.2, we present the workload state identification truth table used by ANX. We now see all of our states accounted for cleanly, and we are able to identify the stable state implicitly by using the definition of *old items* here. We modify our miss history so that

the metadata items are flagged with inserted or filtered, in order to properly distinguish an unstable state from a stable state.

6.3 ANX's Design

A miss history that can flag items as being inserted or filtered is all that is needed to identify workload states. Next, we need an algorithm that uses this information to make the appropriate decisions regarding inserting or filtering into the cache. The foundation of the ANX algorithm is a single parameter that drives it: *anxiety*. *Anxiety* is simply a probability to not cache (i.e., filter), a item on a miss. By using a single probability value, instead of an explicit state machine, ANX allows for all possible workload state transitions implicitly. Second, since anxiety gets updated on every access, ANX itself is quick to react to workload changes, entirely eliminating the need for an observation duration. Finally, by using probability-based state identification and by reacting on every access, it becomes difficult for ANX to become stuck at a local minima without the workload behavior itself driving that conclusion. The only theoretical case would be an adversarial workload for a given cache size, where the workload would only have reuse distances larger than the cache and would initialize the cache with single use items, a combination that would be both short-lived and unlikely in production storage workloads.

Adjusting *anxiety* is currently very simple. When finding an indicator for either a stable or unique access workload, increase *anxiety* to protect what's in the cache, finding the items being requested to be less important. Otherwise, when finding an indicator for an unstable state, decrease *anxiety* to cache what is assumingly a good, new working set. There is a single exception however: the indicator for a stable workload on a cache hit that was *not* recently inserted or filtered does not change anxiety, instead deferring changes to its surroundings. For this case, while this indicator can be observed during workload

behaviors where a working set is larger than the cache (as in the case for churning), it can also occur as part of a often reused working set that can exist within another working set. The latter situation is that of a telescoping working set, where within larger scale working sets smaller working sets can exist with their own access patterns [KVR10].

The rate anxiety increases and decreases is greater when at their opposing extremes to allow for rapid reevaluations of the workload state. Anxiety increases at a rate of $\frac{1-anxiety}{3}$. Anxiety decreases at a rate of $\frac{2}{3}anxiety$.

The design of ANX is composed of *anxiety*, the miss history, and the internal caching algorithm that manages the cache, which is currently LRU.

6.4 Evaluation

Dataset	# Traces	Details
FIU [Sto, KR10]	10	End user/ developer home directories; Web server for faculty/staff/students; Apache web-server for research projects; Web interface for the mail server; Online course management system
MSR [Sto, NDT+08]	36	User home directories; Hardware monitoring; Source control; Web staging; Test web server
CloudVPS [AZ14]	18	VMs for cloud provider
CloudCache [AZ14]	6	Online course management website; Web server for a CS department user webpages
CloudPhysics [WPGA15]	106	VMware VM block traces. Due to the size of many of the traces within this set, only the first day of each trace was used for tests.

Table 6.3: Sources and descriptions for the 5 storage data sets used in this paper.

Algorithms In order to evaluate ANX's performance, it will be compared to the state-of-the-art non-datapath caching algorithms LARC, mARC, and FOMO(ARC). Of the

variations of FOMO, FOMO(ARC) was chosen as it was the best performing of the three options (LRU, ARC, and LIRS).

Other efforts in online non-datapath caching algorithms exist, such as Reinforced Learning Cache (RL-Cache) [KSGS19] and Learning From OPT (LFO) [Ber18]. RL-Cache is a non-datapath caching algorithm that uses reinforcement learning to figure out whether to cache or not cache items. RL-Cache requires the use and availability of several threads and GPUs in order to achieve its results. Furthermore, RL-Cache is primarily designed for variable sized object caches, as RL-Cache also attempts to reduce the number of objects within its cache. LFO is a non-datapath caching algorithm that uses machine learning supervised by a modified version of OPT to learn whether to cache an item or not based on what it thinks OPT would do. While LFO does well, it uses TensorFlow and has large amounts of overhead that makes it inappropriate for consideration for production environments. In particular, these environments run workloads with a lot of dynamic behavior and learning needs to be lightweight, online, and continuous.

Experimental Setup Every algorithm tested had a cache simulator built that would process block I/O workloads and report the number of reads, writes, cache hits, misses, and total writes to the cache. When possible, we used the original authors' version of the algorithm implementation. To simulate sufficient cache space, the size of the cache was set to a faction of the *workload footprint*, which consists of the total size of all unique data accessed. This fractional cache size was varied from 1% to 20% of each workload's footprint in our simulations. The sizes tested were 1%, 2%, 5%, 10%, 15%, and 20% of each workload's footprint.

Workloads To test along a large, diverse set of I/O workloads, we used traces from several sources including FIU, MSR Cambridge, CloudCache, CloudVPS, and CloudPhysics block I/O workloads, each detailed further in Table 6.3 [Sto]. All workloads, with the exception of CloudPhysics, were run for the *full duration* (the full length of the workloads).

To do this, some workloads had separate day workloads merged into one large, continuous workload in order to do so. Cloudphysics, begin as large as it is, was limited to the first day of the workloads in order to reduce the length of the workload and its footprint in order to have tests run on the resources we had available within a reasonable time frame.

Metrics Aswith its non-datapath cache algorithm peers, the metrics critical in ANX's evaluation are hit rate and write rate; with greater write rates contributing to lesser flash cache device lifetimes. Results have been normalized to be compared across workloads and cache sizes in order to confirm the consistency of ANX's performance. Normalized statistics are normalized against the *best* performance of all the algorithms tested for a given workload and cache size combination. The use of normalization is to better show the general performance of an algorithm compared to its peers. In order to keep a consistent "higher is better" visual approach, write rates have been converted to static rates, or the rate at which the cache experiences no changes. For non-datapath caching algorithms, a high static rate is preferable as it indicates that few writes/updates were made, which prolongs the lifespan of the flash cache device. In order to prevent small performance differences from leading to large normalized differences, the results where the best performance was less than 5% were excluded from the results.

Currently, the performance of ANX is being evaluated to determine whether its concept has promise or merit for further exploration and experimentation. To do this, we evaluated ANX's performance in its current state and compared it to its peers, noting not only its workload and cache size results, but also the overall consistency of its performance. As such, the results are presented both as a violin plot of the normalized results as well as averaged bar graphs to break down the results by workload and cache size.

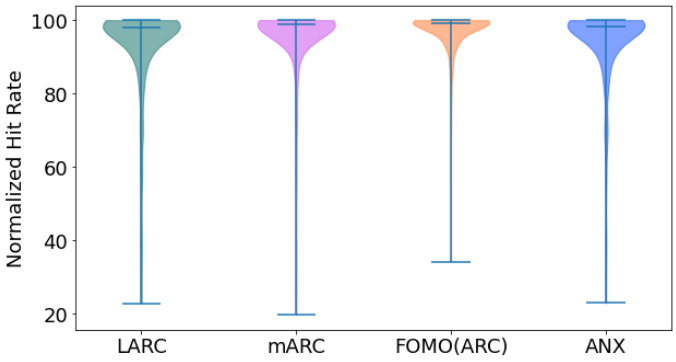

Figure 6.2: Normalized Hit-Rate summary results in percentage, including results for all five different workload sources (FIU, MSR, CloudCache, CloudVPS, and CloudPhysics) measured for size different cache size configurations as a fraction of the workload footprint for each algorithm. Each violin plot also has lines to indicate the max, median, and min of each algorithm's normalized performance. Here it can be seen that ANX has a normalized hit-rate distribution very similar to that of LARC.

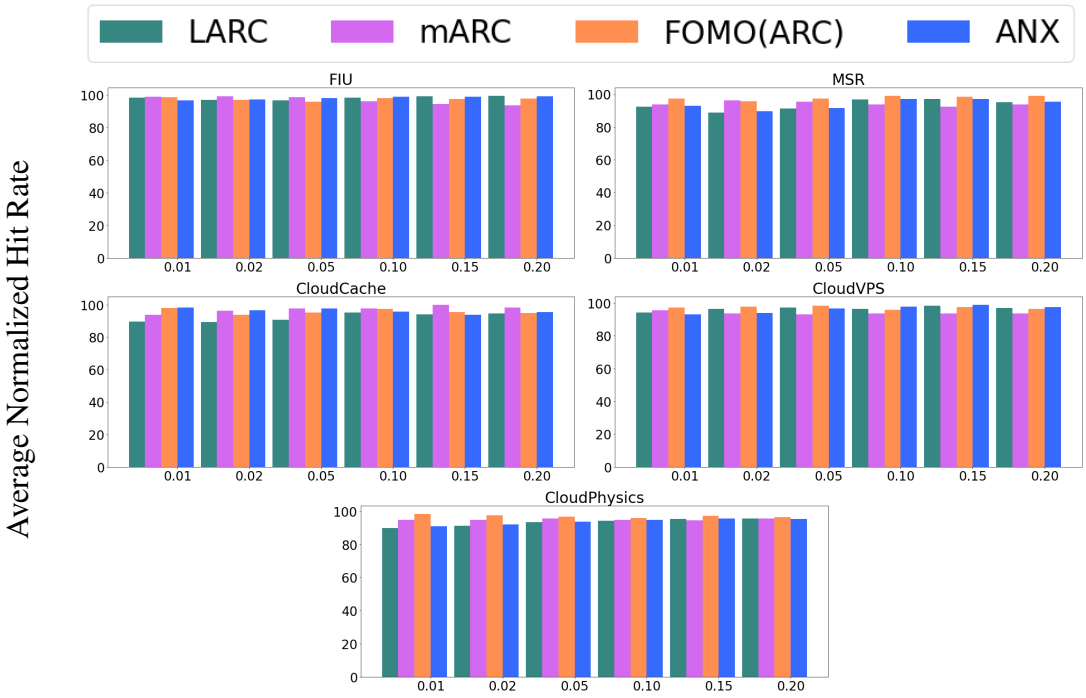

Figure 6.3: Average Normalized Hit Rate results in percentage for the five different workload sources (FIU, MSR, CloudCache, CloudVPS, and CloudPhysics) measured for six different cache size configurations as a fraction of the workload footprint. Notably, of the non-datapath caching algorithms, ANX appears to perform well for the smaller cache sizes in the CloudCache workload.

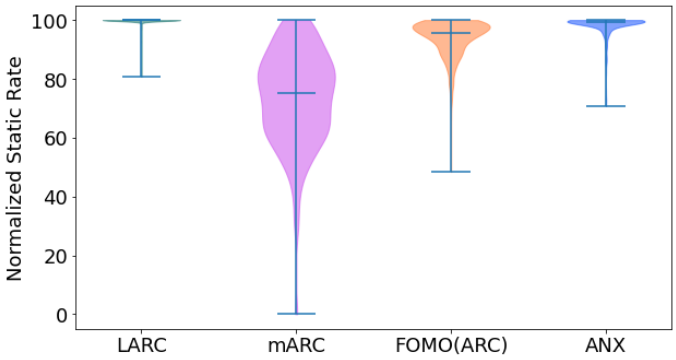

Figure 6.4: Normalized Static-Rate summary results in percentage, including results for all five different workload sources (FIU, MSR, CloudCache, CloudVPS, and CloudPhysics) measured for size different cache size configurations as a fraction of the workload footprint for each algorithm. Each violin plot also has lines to indicate the max, median, and min of each algorithm's normalized performance. Here it can be seen that ANX has a normalized static-rate distribution is mostly similar to LARC, which is consistently the algorithm with the best static-rate.

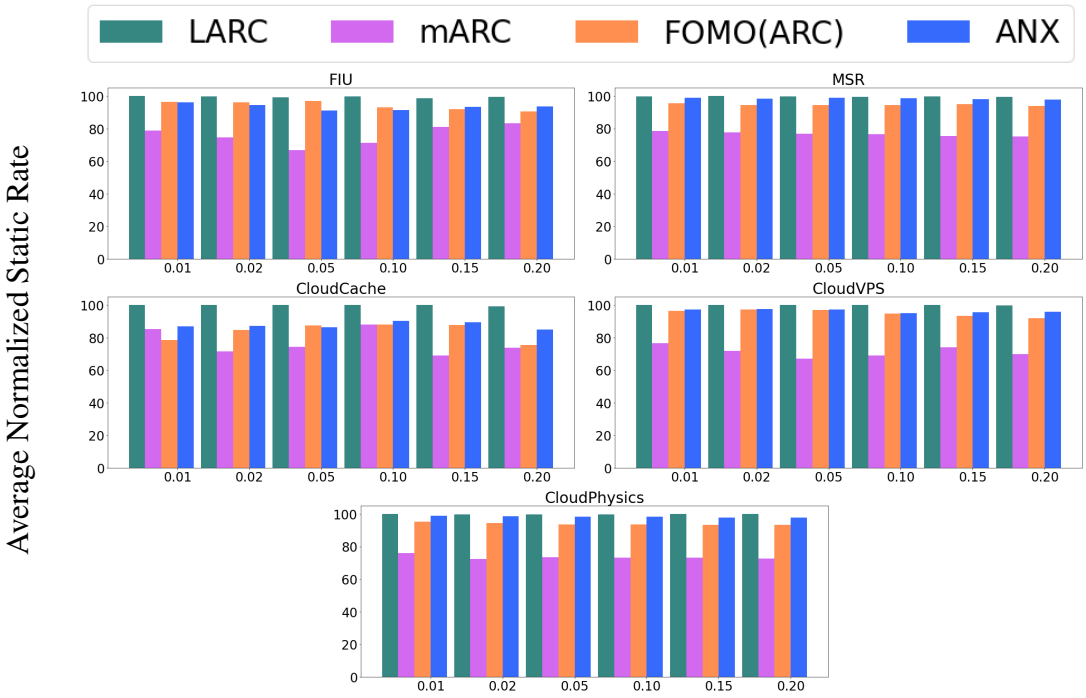

Figure 6.5: Average Normalized Static Rate results in percentage for the five different workload sources (FIU, MSR, CloudCache, CloudVPS, and CloudPhysics) measured for six different cache size configurations as a fraction of the workload footprint. Notably, ANX writes more often than FOMO in CloudCache, which is a workload ANX had an improved hit rate performance on, as seen in Figure 6.3.

6.4.1 Hit Rate

To summarize the hit-rate results in Figure 6.3, we found ANX to perform similarly to LARC for most workloads and cache sizes, with a few exceptions. One notable exception is the smaller cache sizes of CloudCache, a workload known for not being favorable to non-datapath caching algorithms in the FOMO paper. The other is that LARC appears to perform slightly better for the smaller cache sizes of CloudVPS. Otherwise, FOMO(ARC) is generally the best performing non-datapath caching algorithm, with no cases where it is the worst performing amongst its peers.

This can be seen in Figure 6.2, where LARC and ANX's violin plots are similar, with both having a worst case of 22.9% of the best performing. This current version of ANX does have a slightly higher median than LARC, with ANX's being 98.2% compared to LARC's median of 97.9%. Additionally, FOMO(ARC) can be seen to have a worst case of 34.2% of the best performing and a median of 99.2%. This is interesting, as ANX does not have a similar policy to LARC's policy of inserting an item into the cache should a missed item be found in its ghost list (or in ANX's case: the Miss History), leaving this decision instead up to ANX's decision table, randomness and *anxiety*.

6.4.2 Static Rate

To summarize the static-rate results in Figure 6.5, we found ANX to generally write less often than mARC and FOMO(ARC), but slightly more often than LARC. ANX writes considerably more than LARC for the FIU and CloudCache workloads, with the former not resulting in a considerable improvement in hit-rate and the latter producing an improvement to ANX's hit-rate on the smaller cache sizes.

The general similarity of ANX and LARC's static-rate is visualized well in Figure 6.4, showing a similar distribution of results, with ANX having notably more writes.

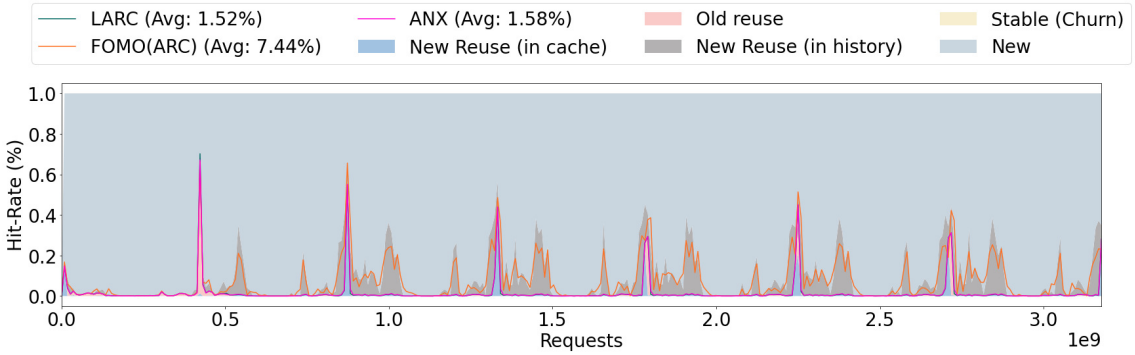

Figure 6.6: Hit-rate of LARC, FOMO(ARC), and ANX over requests. Each point in the graph is separated by 9 million requests. Under ANX's hit-rate curve is a breakdown of these hits: New Reuse (in cache) and Old Reuse. Above of ANX's hit-rate curve is a breakdown of its misses: New Reuse (in history), Stable (Churn), and New. We can see that many of the areas FOMO(ARC) is achieving hits, ANX is encountering New Reuse in its Miss History.

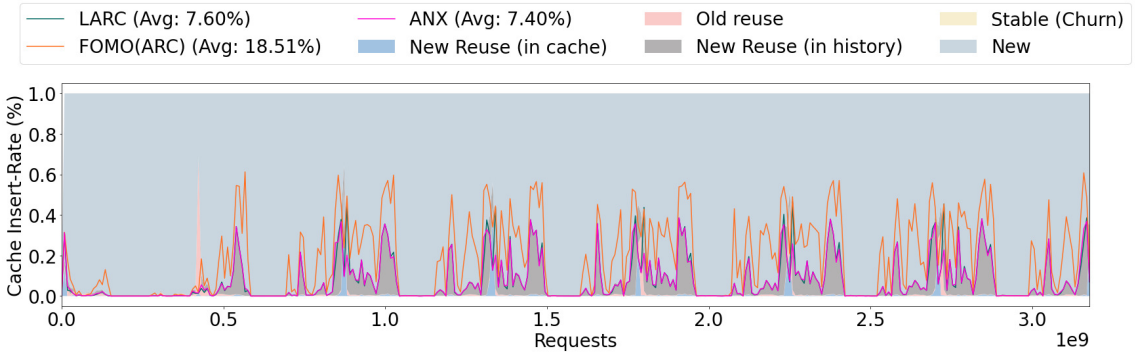

Figure 6.7: Cache insert-rate of LARC, FOMO(ARC), and ANX over requests. Each point in the graph is separated by 9 million requests. While ANX's identifications are not organized to match its line plot, we can still see that majority of ANX and LARC's cache inserts occur when reuse is seen in ANX's Miss History. Alternatively, we can see that FOMO(ARC) is inserting more than both LARC and ANX, and when compared to its hit-rate plot in Figure 6.6 show additional inserts occurring before FOMO(ARC) achieves additional cache hits.

6.4.3 Case Study: src1_1

Analyzing ANX's hit-rate and static-rate results, we next looked into ANX's worst case in order to better understand ANX's performance. ANX's worst case wound up being the MSR Cambridge workload, src1_1, with the cache size being 1% of the workload footprint. Upon analyzing the hit-rate graph in Figure 6.6, we noticed that LARC also doesn't do well in src1_1, but that FOMO(ARC) fares much better. Additionally, it can be noted that ANX's identification types are shown, letting us see that ANX identifies New Reuse (in history) at the same times that FOMO(ARC) is encountering hits, suggesting that FOMO(ARC)'s gains are due to it inserting items without prior reuse into the cache to have benefits later. We are able to confirm this when we bring our attention to the insertions made to the cache (Figure 6.7, as we can notice that FOMO(ARC) is inserting considerably more prior to experiencing cache hits, with LARC and ANX inserts primarily aligning to the number of hits to the miss history are made (New Reuse in history). Credit for these insertions would appear to be FOMO(ARC)'s use of rates and therefore observation periods to continue to insert items into the cache in amounts much greater than the hits to its Miss History.

With this case study analysis, we can conclude that ANX is too quick to raise *anxiety*, suggesting that introducing a method of slowing its rate of increase may prove beneficial in cases such as this.

6.5 Summary

ANX goes in a radically different direction for adaptive non-datapath caching algorithms, away from evaluation periods and reliance on rates and toward miss histories utilizing a new evaluation of workload states and the use of uncertainty in making decisions. The exploration of this unintuitive direction has appeared to given it the validity necessary

for justifying further improvements and exploration. Further exploration can determine the appropriate rates or method that ANX could use to increase and decrease its anxiety. In particular, as seen in the case study, the rate of increase of anxiety can appear to be too high in some cases, leaving possible items to insert and have future hits on excluded from the cache. Another avenue of exploration is in regards to the question of whether it is appropriate to identify a new miss as an indicator of a Unique state when it is still possible for that same miss to be part of a new working set. Another direction for exploration for ANX, which is currently using the LRU eviction policy, is whether ANX should include further protection for items in the cache by using partitions for higher and lower frequency cached items, such as those present in ARC and LIRS. In the next chapter, we will discuss the method of finding the optimal solution for non-datapath caches.

CHAPTER 7

mOPT

Primarily, focus has been on the development of online non-datapath caching algorithms thus far. While these online algorithms focused on retaining or improving the hit-rate compared to their datapath cache counterparts, they drastically improve upon the number of writes made to the cache. However, in thoughts of evaluating these non-datapath caching algorithms, we came upon an unknown: *ideally, how much can we reduce the writes to the cache yet still perform optimally when it comes to the hit-rate*?

The offline algorithm MIN, a common measure for finding the optimal miss rate, does not consider the option available to these non-datapath caches to not cache on a miss request and does not consider optimizing for writes at all. Several online algorithms that use simple heuristics to alleviate the problem tend to use strategies that sacrifice hit-rate in order to significantly reduce write-rate. This approach is one that is seemingly antithetical to the purpose of caching, but provides an acknowledgement that the non-datapath caching problem has multiple areas for evaluation: both in respects to performance and device longevity. As such, an optimization algorithm for non-datapath caches should account for different levels of sensitivity to writes when finding the optimal hit-rate. We propose mOPT, a non-datapath cache optimization algorithm that provides the optimal hit-rate and write-rate for a given write sensitivity. With mOPT, the relation between the hit-rate and write-rate for a cache when adjusting the cache's write sensitivity can demonstrate the appropriateness of the workload for a non-datapath cache.

7.1 Introduction

Belady's MIN algorithm — also known as OPT — is a provably optimal algorithm in regards to datapath caches as it follows the requirement of *always inserting a missed item into the cache*. MIN has often been applied as a useful comparison for caching algo-

rithms, as the gap in performance between MIN and these algorithms represents room for improvement, with the bleeding edge making efforts to move ever close to MIN. Furthermore, while we could apply MIN to the non-datapath caching problem, its propensity to always insert is in conflict with another requirement of non-datapath cache optimality: reducing the number of writes. This reduction of writes is important, not only due to non-datapath caches having the option to *not insert on a miss*, but also due to the storage media used for these caches: flash and persistent memory, both of which are known to have limited write cycles before becoming unusable [BD10, ZHZ+18]. Furthermore, as we shall elaborate later, applying MIN to non-datapath caches also results in a sub-optimal outcome for hit-rate since MIN may insert a less important item (because it *must* insert on misses) while evicting a more important item from the cache.

Heuristics have been previously proposed to have reasonable comparisons for non-datapath caches without using computationally expensive methods that would be required for a true solution [CDS+16]. However, the approximation these heuristics are able to achieve is unknown other than a notable decrease in the estimated number of writes incurred in their approximations to use in comparisons while tending towards having a hit-rate approximately equivalent to MIN. mOPT, on the other hand, derives a correct *optimal* solution for non-datapath caches in polynomial time.

7.2 Background and Motivation

Host-side SSD caches, a popular non-volatile caching solution, introduces a large, fast cache to improve request latency [Lev08, BLM+12, SSZ12]. While these caches are slower than DRAM, they are significantly faster than both HDDs and requests made over a network. However, SSDs and other non-volatile storage devices, such as Intel Optane (3D-XPoint), have a limited write lifespan [ZLL+18, BD10, YKH+20]. This means

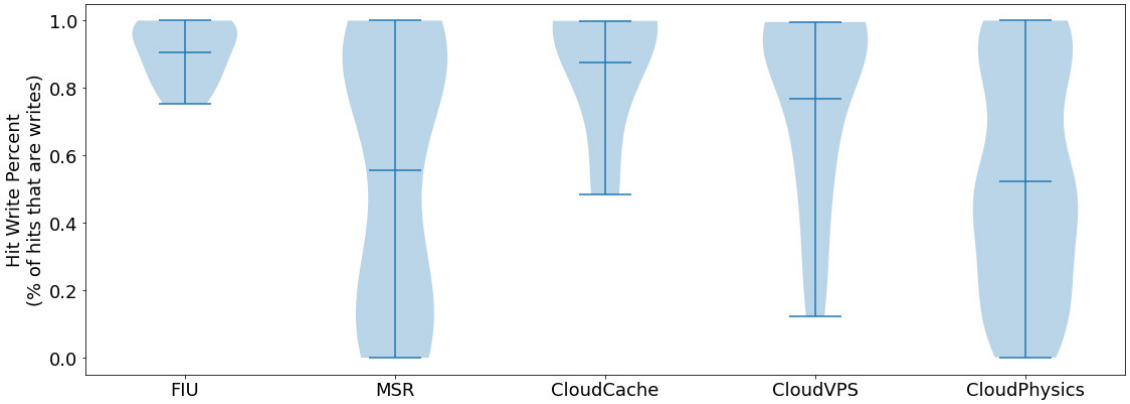

Figure 7.1: Violin graphs showing the breakdown of what percent of each trace's requests that are potentially hits are writes. The whiskers represent the maximum, median, and minimum write percentages among the traces within each trace collection. It can be seen that a significant portion of each trace collection is composed of traces which have 50% or more of their requests being write requests.

that cache updates and insertions are costly in more ways than just performance, and should be minimized while trying to keep latency as low as possible. This focus presents a target for caching algorithms to get the most performance throughout the lifetime of the non-volatile caching solution. Conveniently, these non-volatile caches are not on the traditional datapath, and as such do not require every request to then exist within the cache immediately following the request. This provides a unique and useful option: the choice to not insert an item into the cache.

Two particular events lead to writes to the cache: *cache insertions* and *cache hit on write requests*. The number of *cache insertions* that occurs are influenced by several factors, with the two most significant to this paper being the size of the cache and the number of uniquely requested items. These problems are only exacerbated for traditional datapath caches, due to every cache miss resulting in a cache insertion. Similarly, *cache hits on write requests* result in an update to the item present in the cache, which is effectively a write.

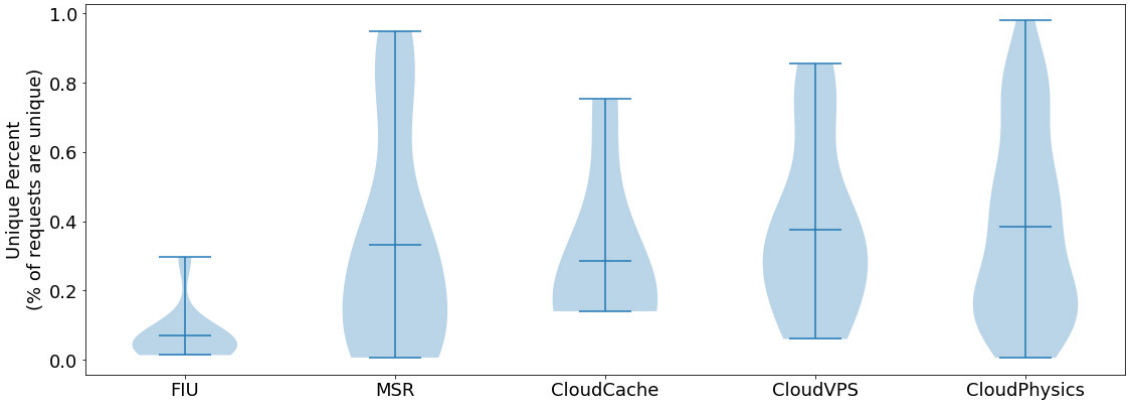

Figure 7.2: Violin graphs showing the breakdown of what percent of each trace's requests are unique. The whiskers represent the maximum, median, and minimum write percentages among the traces within each trace collection. It can be seen that while most of these traces are composed of less than 50% of their requests are to unique items. However, a significant portion of each trace collection is composed of traces which have 50% or more of their requests being write requests, with a few almost entirely composed of unique items.

When analyzing several publicly available workloads, we found that a large number of them have a significant number of potential hits are write requests, as seen in Figure 7.1. This means that when a non-volatile cache is used, a large number of items that are cached primarily to provide performance benefits with a cache hit are likely to encounter a cache write hit significantly more often than a cache read hit. This presents situations where little benefit is found for non-volatile caches and writes continue to shorten its lifespan. Additionally, when looking at the number of unique requests we found that some traces consist almost entirely of unique items, as seen in Figure 7.2. This suggests that a great number of the publicly available workloads show plenty of signs that traditional caching strategies are not appropriate for non-datapath caches. Thus, in order to reduce writes to a non-datapath cache, an optimal non-datapath caching algorithm would both need to *not cache unique requests* and *prefer read cache hits*.

	OPT				M+				Optimal			
(**R**)ead/(**W**)rite	W	R	W	R	W	R	W	R	W	R	W	R
Requests	A	B	A	B	A	B	A	B	A	B	A	B
Cache	A	B	A	B	-	-	-	-	-	B	B	B
Hit	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✔
Write	✔	✔	✔	✔	✗	✗	✗	✗	✗	✔	✗	✗
Results	HR: 0%		WR: 100%		HR: 0%		WR: 0%		HR: 25%		WR: 25%	

Table 7.1: The shown example uses a simple request stream of ABAB and a cache size of 1 to illustrate the different the different shortcomings of MIN and M+. The use of such a small cache is not important, as theoretically more can be packed into the examples and keep the most important moments, the insert-evict decisions the same. MIN is shown to not be capable of getting a hit within this example. M+ is shown to not be capable of getting any hits as well, but opts to not insert any of the items into the cache in order to not incur any writes that do not result in hits. In the theoretical optimal for this example, we can see that not only is one hit found, but by opting to cache B instead of A, the extra write imposed by a write hit is also avoided.

7.2.1 Need For A New Optimal

Previous offline solution efforts to this caching problem have noted that Belady's MIN (OPT) [ZS15] is inappropriate for the non-datapath cache problem as it neither accounts for cache writes nor considers the option to not cache a request that does not improve performance. The M+ offline heuristic [CDS+16] improves the write-rate of MIN by removing wasted cache insertions that do not improve the hit-rate. This approximation is good, but as noted by the authors, additional optimization decisions are left for future work. The need for such optimization decisions is shown in the example in Table 7.1, where M+ is capable of reducing writes, but has trouble finding particular beneficial choices.

To illustrate not only the methodology of MIN and M+ but also how they are lacking compared to the theoretical optimal, we focus on a few simple examples, as seen in Table 7.1. Belady's MIN evicts the item whose next access is the farthest in the future. This strategy has been found to be optimal in calculating the maximum hit-rate for conventional caches that must insert a requested item on a cache miss. M+'s strategy

is to run MIN first, noting times where a cache insertion occurred that did not lead to a cache hit. Following this, M+ then runs MIN once more while avoiding insertions that are known to provide no benefit to the hit-rate. While this strategy is not an optimal for non-datapath caches, as seen in Table 7.1, it cannot have a hit-rate that is worse than MIN while providing a reasonable method of measuring the reduction of writes. Nonetheless, both have room for improvement with respect to improving hit-rate and reducing writesfor non-datapath caches; such as acknowledging write updates (write hits) as less preferred over read hits or finding hits and other trade-offs that are typically not captured with the strategy MIN employs when given the option to not cache on a miss. As such, an offline optimal for non-datapath caches should maximize hit-rate while minimizing the writes to the cache.

7.3 mOPT

Observing the current gap in the optimal for non-datapath caches, we introduce mOPT. mOPT is an offline optimal algorithm non-datapath caches that prioritizes minimizing the miss-rate and write-rate. mOPT is designed to consider the three possible options a non-datapath cache is afforded: cache hit, inserting on a cache miss, and not inserting on a cache miss. With this algorithm, those designing online algorithms can understand how much room exists to improve upon. Similarly, non-datapath cache hosts would be able to evaluate the potential appropriateness of non-datapath caches for their respective workload.

7.3.1 Generalized Model and Objective

When considering the typical non-datapath cache configuration, there exists both the *non-datapath cache* (consisting of a flash device) and the *backing store*. Within this model,

we assume that the cache has a capacity of N blocks, that the cache supports both *cache bypass* and *cache invalidation*, and that all data can be served from the backing store.

We define a workload[1] as a sequence $(\vec{x,w}) = (x_t, w_t)_{t=1}^{T}$ of T unit-size block accesses where $x_t \in I$ for all t and I denotes the set of all blocks that can be requested and w_t denotes the request type (a read or write). w_t represents a read request with 0 and a write request with 1. A data placement algorithm processes the sequence $(\vec{x,w})$ in order, and for each request x_t that is not already in the flash device makes an online decision whether to:

1. bring x_t into the flash device; note that this may potentially evict another block on demand if the flash device is full, or
2. serve x_t directly from the backing store without storing it in the flash device.

Note that (2) represents a *cache bypass* decision. Additionally, as a data placement algorithm processes the sequence $(\vec{x,w})$ in order, and for each request x_t that is already in the flash device makes an online decision whether to:

1. read from or write to the cache block containing x_t based on the request type w_t, or
2. invalidate the cache block containing x_t, serving x_t from the backing store without reading, writing, or otherwise storing it in the flash device.

Note that (2) represents a *cache invalidation* decision.

For mOPT, we focus on a variation of the generalized uniform cache model, where all object sizes and request times are assumed to be uniform, while not considering the time between requests to be significant. mOPT diverges from the generalized uniform cache model only by making a distinction between read and write requests, treating each differently. As such, mOPT is focused on two primary statistics that it minimizes: misses

[1]The workload definition is based on that provided in the paper "*Optimal Data Placement for Heterogeneous Cache, Memory, and Storage Systems*" [ZKAV20]

and writes. To properly optimize for misses and writes in a sensible manner, the write sensitivity is represented with a variable (α) that we call the *write penalty*. This *write penalty* is applied to all writes to affect mOPT decisions. This primarily manifests with α being set to a very small non-zero value that influences mOPT to minimize the number of writes while never choosing to increase the number of misses to reduce the number of writes.

The general cost model mOPT adopts is $C((\vec{x,w})) = \sum_{t=1}^{T} m_t + \alpha w_t$. $C(\vec{x})$ is the cost imposed over the entirety of the workload. m_t is the cumulative miss cost of the entire cache (all N cache blocks) at time t, which is typically N on a cache miss or $N-1$ on a cache hit, where N represents the size of the cache. As previously discussed α is the *write penalty* assigned to each write that occurs to the cache. Lastly, w_t is the number of writes that occurs to the cache at time t, which is typically just 0 or 1, which is a read or write respectively. Hence, each operation incurs different costs:

- **Cache write hit:** $(N-1)+\alpha$
- **Cache read hit:** $(N-1)+0$
- **Cache miss:** $N+\alpha$
- **Cache bypass:** $N+0$
- **Cache invalidation:** $N+0$

7.3.2 Designing the Offline Optimal Algorithm mOPT

We now introduce mOPT, an optimal offline algorithm for non-datapath caches using the previously defined cost model. mOPT represents each of the N blocks of cache block with a network flow using vertices that represent requests and edges to represent cache operations.

mOPT then translates the problem of making optimal caching decisions into a Min-Cost Flow (MCF) problem. The resulting flows of mOPT would be the optimal decisions the non-datapath cache can make to minimize both misses and writes for a given workload, cache size, and write penalty. The translation of optimal caching problems to MCF problems is shown to be practical and useful [BBH17, ZKAV20, ASWB20]. In Section 7.3.2, we'll discuss the specifics on how this translation is done.

Within our discussions, we will be using a tuple representation when discussing the costs of operations, decisions, or flows, such as $(4,2)$. The (M,W) tuple representation is shorthand for the cost formula of

$$E(x_{t_a}, x_{t_b}) = \sum_{t=t_a}^{t_b} m_t + \alpha w_t$$

where the *edge flow cost* (E) is determined by the number of *misses* (m) and *writes* (w) a flow across an edge would impose. While appearing similar to the cost model defined in 7.3.1, there is an important caveat: E is a cost for a flow, or a cache block, whereas C is the cost for the entire cache. This manifests in all previously given costs of cache operations to not include $N-1$ misses (the misses imposed by the other flows at the same time instance). The *write penalty* (α), when using the shorthand representation, is ideally so small that misses are not affected directly by the presence of writes. As such, in the discussions, mOPT will be solving primarily for the minimal hit-rate, then the minimal write-rate, hence the ordering of the tuple being misses first, then writes.

mOPT Construction

Previous cache optimization algorithms that use MCF, such as FOO [BBH17], CHOPT [ZKAV20] and Belatedly [ASWB20], have tiered graph designs that represent a single request for an item with multiple vertices. Within this design, these multiple vertices are representations of an item existing on a particular tier following the request.

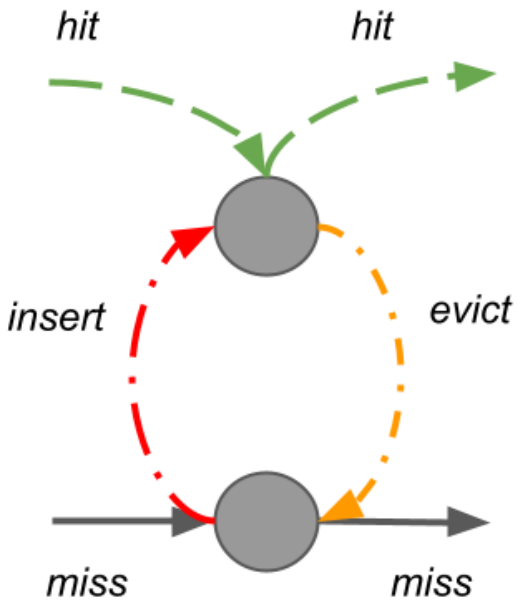

Figure 7.3: The simplified anatomy of how a request is represented in mOPT. Each request exists in mOPT as two vertices, one (top) existing as a representation of the requested item in the cache, the other (bottom) existing as a representation of the requested item not in the cache. As such, the edges connecting the two represent inserting and evicting the item from the cache. Similarly, the implied connections to other items on the top represent hits, while the implied connection to other items on the bottom represent misses. Notably, hits are connections only from/to other requests for the same item; whereas misses are edges from the previous and to the next requests.

We found the two-tiered design to be rather elegant and easily adaptable towards solving the presented problem. Within our design, the top tier in examples represents the request being in cache while the bottom tier is the item existing only in the backing store, or more simply not being in the cache.

With mOPT using a two-tiered design, each request is represented by two vertices: a_t and a'_t. Each of these vertices representing the item either being present in the cache or not. The following description of the request model, focusing on its vertices and edges has an accompanying visual representation with Figure 7.3. As such, the edge from the not-in-cache vertex to the in-cache vertex costs a write, due to it representing a cache insertion. The edge from the in-cache vertex to the not-in-cache vertex costs nothing,

representing a cache eviction, which presents no costs in our model. Both of the edges connecting the not-in-cache vertex and the in-cache vertex have a capacity of one. All requests have the previous request's not-in-cache vertex connected to their not-in-cache vertex with an edge. This edge costs a miss with a capacity equal to the cache size, as any flow going between these not-in-cache vertices is missing whatever the request is and since the entire cache can miss on the request the capacity of the edge must be the equivalent to the cache size.

If an accessed item had been previously requested, an edge is created from the most recent previous request's in-cache vertex to the current request's in-cache vertex to represent a hit. As such, the miss cost is calculated as one less than the number of requests between them. This provides preference for flows that can produce multiple hits within the same time frame as another with a single hit. The write cost is either zero or one based on whether the request is respectively a read or write request. The capacity of this *hit edge* is one.

As with many MCMF graphs, we will have a single source and a single sink for flows to come from and end at. The source only connects to the vertex representing the first request. Only the vertex representing the final request is connected to the sink. The edge the source connects with has a cost of a single miss or $(1,0)$ and a capacity equal to the cache size, just as all of edges connecting to all not-in-cache vertices. The edge connected to the sink have no cost and have a capacity equal to the cache size.

After construction, an MCMF solving algorithm, such as Bellman-Ford's algorithm can be used to find an optimal solution, as shown in Appendix A. With this optimal solution, we can determine the cache state and decisions by moving along the not-in-cache vertices and taking note when flows went into or out of the in-cache vertex. As such, we can write an algorithm that can read a solved graph and determine both whether to insert an item on a miss and what to evict when performing a cache insert on a full

cache. While evictions in the graph would be prior to the cache insertion that would cause it, such an algorithm would be creating a queue of items that are ready to be evicted. This queue would then be used when performing a caching insert on a full cache to provide an item for eviction as needed.

7.3.3 Optimization

As the runtime of MCMF solving algorithms are heavily dependent on the number of vertices and edges, removing any superfluous ones can provide performance benefits. An optimization employed by mOPT is the removal of any requests that are not referenced more than once from consideration. The reasoning for this is that there is no way that mOPT would choose to cache any item that would not be referred to again in the future, thus its removal from the graph provides a worthwhile optimization.

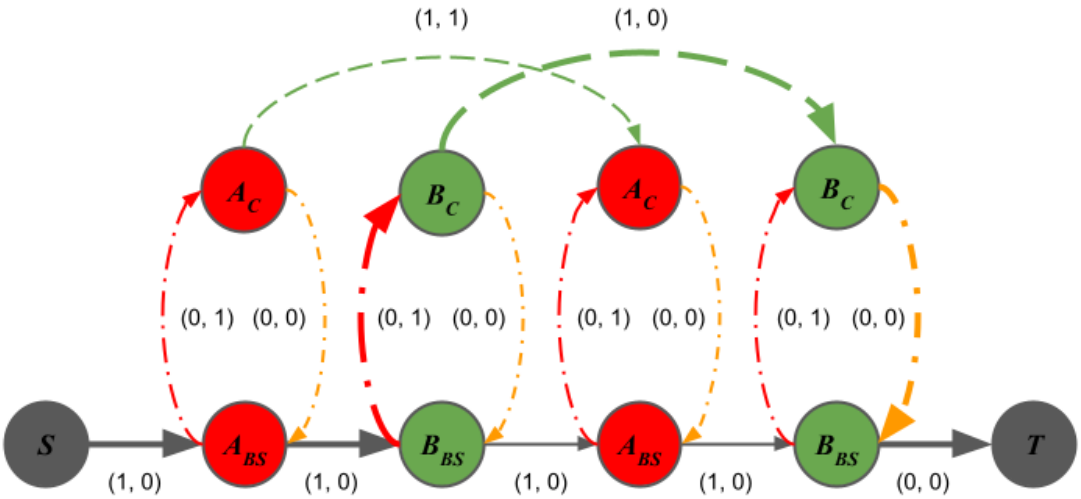

Figure 7.4: Here we revisit the example stream provided in Table 7.1, using mOPT to solve for the optimal shown there. As such, the optimal path is represented by path with bold edges. The total cost of this path is $(3, 1)$, or a hit-rate of 25% and a write-rate of 25%, an exact match for the optimal within the previous example.

7.3.4 Illustrative Example

Figure 7.4 shows that when solving for the previous example shown in Table 7.1. With a request stream of *ABAB*, with a cache size of 1, mOPT is able to find the optimal choice that was previously shown. Here, we can see how when considering the two options of caching *A* or *B*, mOPT determines which is preferred by comparing the number of writes incurred by each option.

7.4 Evaluation

Experimental Setup We built cache simulators for the MIN, M+, and mOPT algorithms in C++ that can process block I/O workloads and report on the total number of misses and writes they incurred. In order to simulate sufficient cache space as well capture a sufficient amount of I/O activity, the size of the cache was set to be a fraction of the *workload footprint*, which is defined as the total size of all unique data accessed. This fractional cache size was varied from 1% to 20% of each workload's footprint for our simulations (with the cache sizes specifically being set to 1%, 2%, 5%, 10%, 15%, and 20%). The following results, unless specified otherwise will have a write penalty (α) of a small non-zero value.

Workloads To have a large, diverse set of block I/O workloads, we used the FIU [iod10, Sto], MSR Cambridge [NDR08, Sto], CloudCache [AZ14], CloudVPS [AZ14], and CloudPhysics [WPGA15] block I/O workloads for testing. Originally, due to their length, CloudPhysics workloads were limited to the first day. These workloads were then limited further along with the rest of the workloads. Due to the runtime of mOPT, all workloads were limited to their first 1 million requests. A reliable spatial sampling method [ZKAV20, WSAP17] was employed to reduce the number of computations necessary for solving for the workloads used for our simulations. These papers also noted

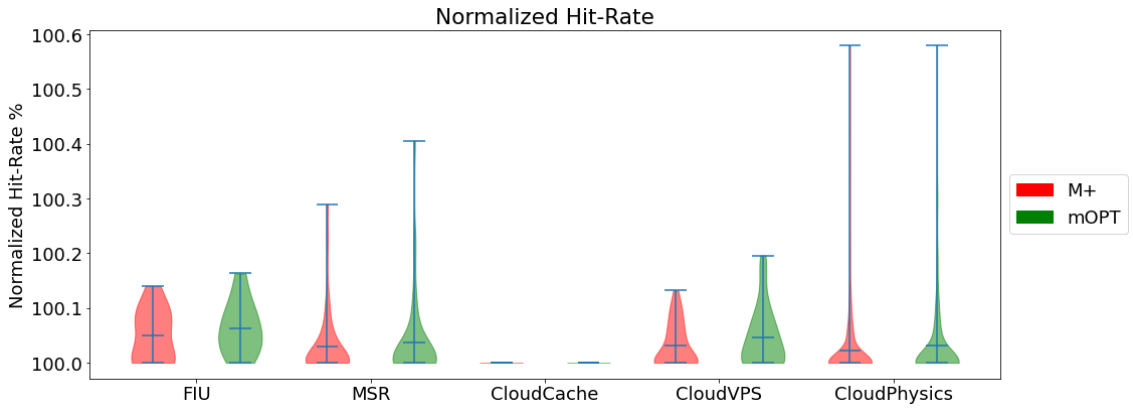

Figure 7.5: This violin graph shows the hit-rate results of M+ and mOPT, normalized to the hit-rate of MIN. We can see here that M+ will perform slightly better than MIN, and mOPT will perform very slightly better than M+. An interesting outlier is that of CloudCache, where neither M+ nor mOPT was able to find opportunities to improve the hit-rate further.

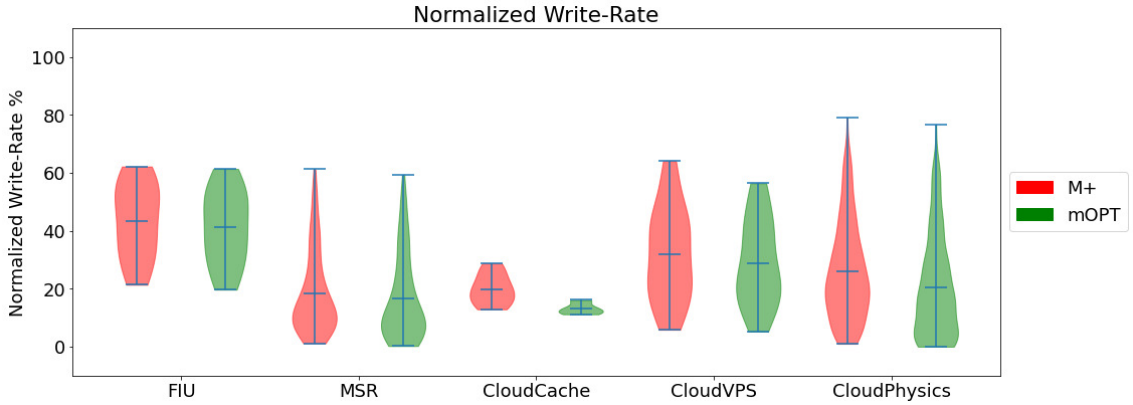

Figure 7.6: This violin graph shows the write-rate results of M+ and mOPT, normalized to the write-rate of MIN. We can see here that M+ will reduce the write-rate a great deal better than MIN, but that mOPT can still find a significant number of opportunities to reduce writes further.

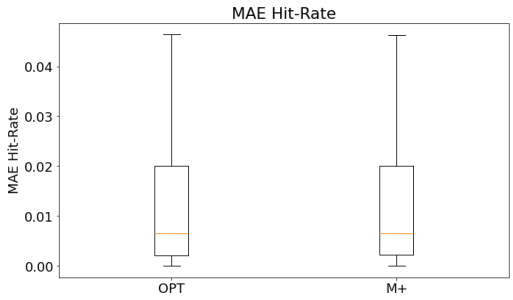

Figure 7.7: Here we can see the Mean Absolute Error (MAE) that MIN and M+ have due to the sampling of the trace. These errors are determined by the difference in performance compared to their unsampled counterparts, hence why mOPT was not capable of having its error measured similarly. We can see that for the most part, the sampled hit-rate results are typically accurate, with a number of outliers having an MAE of at most 0.26.

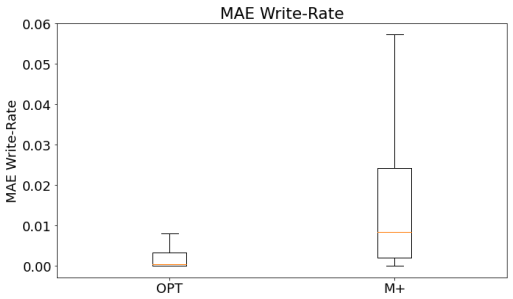

Figure 7.8: Here we can see the Mean Absolute Error (MAE) that MIN and M+ have due to the sampling of the trace. These errors are determined by the difference in performance compared to their unsampled counterparts, hence why mOPT was not capable of having its error measured similarly. We can see that for the most part, the sampled write-rate results are typically accurate. We can see that MIN is less error prone, due to its always caching nature. M+, making a variety of choices presents more opportunities for errors, but even so still has typically accurate sampled write-rate results, with a number of outliers having an MAE of at most 0.242.

that the amount of error increases when the sampling rate is less than 0.001 or the cache size is less than 100. As such, any test that was attempted to be reasonably simplified for runtime improvements, but surpassed these limits are not included. There was a subset of these tests that did not complete within our 20 hour time frame, despite the techniques used to improve runtimes and are also not included.

7.4.1 Offline Algorithms

We compared mOPT to MIN and M+, normalizing our results to MIN. As seen in Figure 7.5, there is only a small amount of room for hit-rate improvements within these traces compared to MIN, and less when compared to M+. The maximum normalized hit-rate increase mOPT brings is a mere 0.579% over MIN. The bulk of the improvements mOPT brings lies with the write-rate, which, as can be seen in Figure 7.6, are significantly less than MIN and still shows a marked improvement compared to M+. Here, we can find a minimal improvement mOPT brings over MIN of 13.156% less writes, while M+ at worst reduces writes 11.056% over MIN. At best, we find M+ reducing writes a full 99% and mOPT reducing writes the whole 100%. This is likely in cases where no reuse is found, where M+ will still warm up the cache prior to using the OPT eviction algorithm for decisions and mOPT will optimize this cache warm up period away.

With these results, we then set out to measure the Mean Absolute Error of these results compared to their unsampled counterparts. Mean Absolute Error is a measure of the absolute error between the expected and the actual results, the measures are in the same unit as the statistic the error is being measured. As we sampled our traces in order to improve test runtimes, we can not include mOPT within this error measurement. As seen in Figure 7.7 and 7.8, the error is mostly negligible. At worst, we found that the measured error got as bad as 0.26 Mean Absolute Error for MIN and M+ hit-rate. When

considering write-rate, MIN had a worst case Mean Absolute Error of 0.083, whereas M+ had 0.242 in the worst case. These worst case scenarios are outliers, with the majority of cases having a much smaller error.

7.5 Summary

While Belady's MIN is a wonderful measure for optimal hit-rate, it lacks both the consideration for *selective caching* as well as optimizing for writes. M+ improves upon MIN, but is not advanced enough to optimize properly for both hit-rate and write-rate. mOPT properly considers *selective caching* and optimizing for writes and is able to optimize for both the miss-rate and the write-rate within non-datapath caching scenarios. While mOPT is able to only improve upon MIN's miss-rate a miniscule amount, the write-rate measures provide an invaluable understanding of how little writes are necessary in order to achieve such hit-rates. This provides a new measurement for algorithm designers to compare against in their drive towards designing algorithms that perform as close to optimal as possible. In the next chapter we will discuss related works.

CHAPTER 8

RELATED WORK

In this chapter we examine the body of related work for mARC, FOMO, ANX, and mOPT. In each subsection, we discuss the relevant work for a topic of the solutions proposed in this dissertation and describe the difference between our solutions and previous works, highlighting our unique contributions.

8.1 Datapath Caches

The processing speed of the central processing unit (CPU) has continuously improved, but the latency to dynamic random access memory (DRAM) still serves as a bottleneck to performance, referred to as the memory wall [WM95]. However, while the CPU itself has static random access memory (SRAM) caches in order to side-step this latency, the datasets of several applications, such as big-data, are simply too large to be captured solely by the small SRAM caches, and thus the latency and size of DRAM become important as DRAM is more capable of holding these larger working sets [JVF13]. The operating system manages this DRAM using the memory management unit, the Linux kernel will use DRAM as a cache, attempting to keep hot pages in the cache while evicting cold pages when necessary [HQS16].

With the management of DRAM being so important, several strategies for managing this memory were developed. Least recently used (LRU) was among the first and best, using an understanding of temporal locality, that what has been accessed is likely to be accessed again in the near future, to develop a stack-based algorithm for determining hot and cold pages [DT90]. As LRU's overhead was too great, a 1-bit approximation of LRU was developed that was called CLOCK, using a circular queue to manage hot and cold pages in an approximately LRU-like behavior [CV65]. Looking to improve upon LRU, the adaptive replacement cache (ARC) was developed, introducing a two tiered caching

algorithm that was able to better hold onto reused pages while also being capable of resisting scanning workload behaviors while adapting to the workload using its eviction history [MM03]. Low inter-reference recency set (LIRS), was also built to improve upon LRU, using a two tiered caching algorithm that was able hold onto reused pages very well, providing a great amount of correction using its eviction history while resisting scanning workload behaviors [JZ02]. These strategies are designed for the datapath cache, making them improper for non-datapath caches. The solutions proposed in this dissertation (mARC, FOMO, and ANX) are designed for non-datapath caches and they achieve significantly lower cache write-rate and a comparable hit-rate when compared to the body of datapath caching algorithms.

8.2 Flash in Storage Systems

Solid State Hybrid Drives (SSHDs) place themselves between SSDs and HDDs in terms of performance and cost while having storage volumes more in line with HDDs [Sch14]. These SSHDs internally are choosing where data should go, having important data reside on the SSD portion and others on the slower HDD portion of the device, mARC, FOMO, and ANX do not control where data goes but just acts as a cache for the data instead. Another point of hardware development for flash in storage systems is the introduction of the Flash Cache by NetApp, a PCI-E flash device that uses intelligent caching to improve performance without moving data from a slower existing hard drive [Net]. Flash Cache offers a caching solution that is similar to the design of the non-datapath caches we envision, however it is a piece of specialized hardware running a specialized, intelligent algorithm that determines how it caches. mARC, FOMO, and ANX on the other hand are designed for general purpose flash drives and persistent memory with the use of kernel modules such as dm-cache [dm-]. Another solution proposed the use of flash devices

in storage systems by using a tier system like the Extent-based Dynamic Tiering (EDT) system that intelligently satisfies performance requirements while reducing power/operating costs [GPG+11]. EDT is a solution that is built to improve storage systems with the inclusion of different storage technologies. Our work addresses host-side improvements that are targeted towards the same goal. This class of solutions are built to improve storage systems by making use of different device technologies. EDT is a solution that focuses on the management of resources rather than caching, similarly to SSHD. Our work, in contrast, addresses host-side improvements that are targeted towards the same goal.

8.3 Host-Side Caches

Host-side caching devices are typically non-volatile, being flash or persistent memory (such as 3D-XPoint) meaning that data written to the cache endures through complications such as node and power failures. This opens them up to potentially be used for caching writes [KMR+13]. Most host-side caches, however, use a write-through policy that makes writes to both the cache and backing storage, making the write to the cache somewhat redundant, but understandable for strictly fail-safe writes [BLM+12]. While writes can be cached for performance, there are still many scenarios where the avoidance of writing to the cache can keep the cache performing well and extend the lifespan of the flash or persistent memory device, which is the focus of mARC, FOMO, and ANX. Host-side caches can be a powerful part of delivering a certain amount of quality-of-service for workloads [KMR15]. The performance speed up from the inclusion of the host-side cache is enough to accommodate such quality-of-service needs [THL+15]. The work proposed in this dissertation supports the use of multiple write caching policies.

8.4 Non-Datapath Caching Algorithms

With the introduction of non-datapath caches and the acknowledgement of their particular limitations and the shortcomings of using existing caching algorithms, have opened the doors to new approaches.

Lazy Adaptive Replacement Cache (LARC), is a non-datapath caching algorithm that focuses on not caching unique accesses or very low frequency items by not caching anything until its second occurrence [HWC+13]. With how LARC focuses first on avoiding writes, it suffers additional misses that could be otherwise avoided by caching on particular first occurrances, like the unstable state in mARC and FOMO or when *anxiety* is low in ANX.

Reinforced Learning Cache (RL-Cache), is a non-datapath caching algorithm that uses a subset of machine learning, reinforcement learning, to figure out whether to cache or not cache items [KSGS19]. RL-Cache requires the use of several threads and GPUs in order to achieve the results it has, a large overhead that is avoided with our solutions.

Learning From OPT (LFO), is a non-datapath caching algorithm that uses machine learning supervised by a modified version of OPT to learn whether to cache an item or not based on what it thinks OPT would do [Ber18]. While LFO does well, it uses TensorFlow and has large amounts of overhead that make it inappropriate for consideration for production environments, overheads avoided with our solutions.

Other approaches avoid writing to the cache by using deduplication to reuse existing data, such as D-LRU and D-ARC [LJBR+16]. While we haven't explored deduplication, the amount of complexity added and potential overhead should a write-back write policy be used to evict dirty data suggest that D-LRU and D-ARC may not be appropriate for certain caching environments where write-back is used to improve performance.

Nevertheless, deduplication as a general technique for host-side cache management is orthogonal to our proposed work.

8.5 Optimality for Non-Datapath Caching

While solutions do not exist for non-datapath caches, heuristics have been proposed to approximate the solution, they are as follows: M^+, a MIN variant which determines to not cache an item if the item would be evicted before reuse, M_N, a variant of M^+ that does not insert items with less than N accesses, and lastly, M_T, a variant of M^+ that gets the best hit-rate possible while having a given max number of erasures [CDS+16]. With these heuristics, it is not currently known how closely they approximate the optimal for non-datapath caches, though that could potentially be answered following the proposal and evaluation of mOPT for comparison. The proposed mOPT, on the other hand, is proven to be the offline optimal algorithm for non-datapath caches with fixed sized items, as seen in Appendix A.

Previous work on optimality of caching decisions has been focused on the datapath cache, as seen with MIN. More recent works have included a translation of MIN into a Min-Cost Max Flow problem called FOO [BBH17]. FOO is primarily designed to extend upon MIN by solving for the non-uniform size caching problem. FOO was also later used by the same author for LFO's [Ber18] MIN model that was used for LFO's machine learning. mOPT instead solves for the uniform caching problem, which includes items of the same items, for the non-datapath cache context.

Other works that used Min-Cost Max Flow to solve for cache-based optimality problems, includes both CHOPT [ZKAV20] and Belatedly [ASWB20]. CHOPT aims not only to optimize for latency, but to optimize for data placement among multiple tiers of memory. Belatedly similarly solves for latency, but accounts for an event not typical

in most optimizations: delayed hits. Delayed hits are hits that occur as the item is being brought into the cache. While typically reported in most optimizations as hits, these cache hits must still wait for the item to be in the cache before the request is handled. mOPT does not currently solve for latency, is currently restricted to two tiers (backing-store and cache) and does not account for delayed hits. mOPT instead accounts for misses and writes, though it doesn't seem impossible to adapt mOPT to account for latency instead of misses within its optimization.

CHAPTER 9

CONCLUSIONS

In this thesis we studied the non-datapath caching problem from two perspectives: the development of online caching algorithms and the development of an offline optimal algorithm. We focused on the new option non-datapath caches provide: *selective caching*, where caching algorithms are capable of choosing whether to insert an item on a cache miss or not. This single additional choice, when compared to datapath caches, has introduced an interesting new angle of complexity towards the decisions a non-datapath caching algorithm can make. Furthermore, the devices used for non-datapath caches (flash and other non-volatile devices), introduces a secondary interest in reducing the number of writes made to the devices as they act as caches in order to stretch out their lifetime of usefulness.

We began by introducing a non-datapath caching algorithm that was capable of utilizing hit-rate information to identify workload states that benefit from inserting items on a cache miss or not. Existing solutions were either datapath caches [MM03, JZ02] that did not consider the option to not cache an item on a cache miss, or was a simple heuristic for non-datapath caches that did not adapt with the workload [HWC+13], missing out on opportunities for hits that were otherwise achievable. This adaption to the workload allowed mARC to provide performance improvements of both reducing writes compared to datapath caching algorithms, while having improved hit-rate over LARC.

Next, we focused on the areas mARC could improve. In particular, the reduction of mARC's parameters and transition conditions provided an opportunity for improvement. With FOMO, not only were we able to develop a workload identification model that involved only two states, but also two transition conditions. Additionally, FOMO was generalized to allow for any datapath caching algorithm to benefit from non-datapath caching options without modification. These changes resulted in the FOMO version of

these datapath caching algorithms having write-rates similar to that of LARC while finding hit-rate performances generally improved from their datapath counterparts.

Next, we focused on where FOMO could be further improved. One particular point present in both mARC and FOMO is the use of an observation period before calculating rate metrics relevant for making decisions. Through an exploration of options and ideas, we settled upon what would later be called *anxiety*. Through this constantly changing metric, ANX was more than capable of finding opportunities that otherwise would go unnoticed within the observation periods of mARC and FOMO. ANX has shown great promise in the ideas used in its design: anxiety, the miss history, the identifiers it uses to adjust anxiety. With this promise, there exists merit in further experimentation and exploration of ANX's design space.

Finally, we developed mOPT to understand both how much room for improvement was possible for the known online non-datapath caching algorithms. Previously there existed no non-datapath equivalent to MIN [ZS15], which has often been used to judge how closely a datapath caching algorithm can perform to optimal.

CHAPTER 10

FUTURE WORK

10.1 Online Non-Datapath Caching Algorithm

Online caching algorithms, regardless of the application, are still a constantly evolving area with new ideas and innovation. For non-datapath caches, there is still a wealth of areas left uncovered for others to venture and experiment. Whether through a more intimate knowledge of the workload states, or improved designs, the next best algorithm is an idea away. Beyond heuristics however, is the field of Machine Learning, which is currently being explored in several caching contexts and performs well, but generally requires additional resources to attain.

10.2 Offline Non-Datapath Caching Algorithm

The foundations of mOPT can be further expanded to solve further problems from the context of a non-datapath cache. A primary example is that of latency, which is a fundamentally different statistic from misses, but is essential in performance analysis. Another example still incorporates latency, but now is aware of delayed hits. mOPT also does not include different cache write policies and can be expanded to further consider them.

BIBLIOGRAPHY

[AMO93] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Upper Saddle River, New Jersey, 1993.

[ASWB20] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S. Berger. Caching with Delayed Hits. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, pages 495–513, New York, NY, USA, 2020. Association for Computing Machinery.

[AZ14] Dulcardo Arteaga and Ming Zhao. Client-side Flash Caching for Cloud Systems. In *Proceedings of International Conference on Systems and Storage*, SYSTOR 2014, 2014.

[BBH17] Daniel S. Berger, Nathan Beckmann, and Mor Harchol-Balter. Practical Bounds on Optimal Caching with Variable Object Sizes. *CoRR*, abs/1711.03709, 2017.

[BD10] Simona Boboila and Peter Desnoyers. Write Endurance in Flash Drives: Measurements and Analysis. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 9–9, Berkeley, CA, USA, 2010. USENIX Association.

[Bel66] Laszlo A. Belady. A Study of Replacement Algorithms for Virtual-Storage Computer. *IBM Systems Journal*, 5:78–101, 1966.

[Ber18] Daniel S. Berger. Towards Lightweight and Robust Machine Learning for CDN Caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, pages 134–140, New York, NY, USA, 2018. ACM.

[BLM+12] 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 *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, April 2012.

[CDS+16] Yue Cheng, Fred Douglis, Philip Shilane, Grant Wallace, Peter Desnoyers, and Kai Li. Erasing Belady's Limitations: In Search of Flash Cache Offline Optimality. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 379–392, Denver, CO, June 2016. USENIX Association.

[CoT68] F.J. Corbató and Project MAC (Massachusetts Institute of Technology). *A Paging Experiment with the Multics System*. Project MAC. Defense Technical Information Center, 1968.

[CV65] Fernando J. Corbató and V. A. Vyssotsky. Introduction and Overview of the Multics System. *Proc. of the AFIPS Conference, Volume 27, Part 1, 1965 Fall Joint Computer Conference*, pages 185–196, 1965.

[dm-] Linux dm-cache kernel module. https://www.kernel.org/doc/Documentation/device-mapper/cache.txt.

[DT90] Asit Dan and Don Towsley. An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '90, pages 143–152, New York, NY, USA, 1990. ACM.

[EK72] Jack Edmonds and Richard M Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM (JACM)*, 19(2):248–264, 1972.

[FJF62] Lester Randolph Ford Jr and Delbert Ray Fulkerson. *Flows in Networks*. Princeton university press, 1962.

[FT87] Michael L Fredman and Robert Endre Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[GPG+11] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost Effective Storage using Extent-based Dynamic Tiering. In *Proc. of the USENIX Conference on File and Storage Technologies*, February 2011.

[HAWS13] David A. Holland, Elaine Angelino, Gideon Wald, and Margo I. Seltzer. Flash Caching on the Storage Client. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 127–138, San Jose, CA, 2013. USENIX.

[HQS16] Jian Huang, Moinuddin K. Qureshi, and Karsten Schwan. An Evolutionary Study of Linux Memory Management for Fun and Profit. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 465–478, Denver, CO, June 2016. USENIX Association.

[HWC+13] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng. Improving Flash-based Disk Cache with Lazy Adaptive Replacement. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, May 2013.

[iod10] I/O Deduplication (project webpage). http://sylab.cs.fiu.edu/projects/iodedup/, 2010.

[JVF13] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 404–415, New York, NY, USA, 2013. ACM.

[JZ02] Song Jiang and Xiaodong Zhang. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, pages 31–42, New York, NY, USA, 2002. ACM.

[KBC+15] D. Kang, S. Baek, J. Choi, D. Lee, S. H. Noh, and O. Mutlu. Amnesic Cache Management for Non-Volatile Memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–13, May 2015.

[KKI+14] Hyojun Kim, Ioannis Koltsidas, Nikolas Ioannou, Sangeetha Seshadri, Paul Muench, Clement L. Dickey, and Lawrence Chiu. How Could a Flash Cache Degrade Database Performance Rather Than Improve It? Lessons to be Learnt from Multi-Tiered Storage. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*, Broomfield, CO, October 2014. USENIX Association.

[KMR+13] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write Policies for Host-side Flash Caches. *To appear in the Proc. of the USENIX File and Storage Technologies*, February 2013.

[KMR15] Ricardo Koller, Ali Jose Mashtizadeh, and Raju Rangaswami. Centaur: Host-side SSD Caching for Storage Performance Control. In *Proc. of ICAC*, July 2015.

[KR10] Ricardo Koller and Raju Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. In *Proc. of the USENIX Conference on File and Storage Technologies*, February 2010.

[KSGS19] Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, and Ramesh K. Sitaraman. RL-Cache: Learning-Based Cache Admission for Content Delivery. In *Proceedings of the 2019 Workshop on Network Meets AI & ML*, NetAI'19, pages 57–63, New York, NY, USA, 2019. ACM.

[KV12] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, fifth edition, 2012.

[KVR10] Ricardo Koller, Akshat Verma, and Raju Rangaswami. Generalized ERSS Tree Model: Revisiting Working Sets. In *Proc. of IFIP Performance*, November 2010.

[Lev08] Adam Leventhal. Flash Storage Memory. *Commun. ACM*, 51:47–51, July 2008.

[Li18] Cong Li. DLIRS: Improving Low Inter-Reference Recency Set Cache Replacement Policy with Dynamics. In *Proceedings of the 11th ACM International Systems and Storage Conference*, SYSTOR '18, pages 59–64, New York, NY, USA, 2018. ACM.

[LJBR+16] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. CacheDedup: In-line Deduplication for Flash Caching. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 301–314, Santa Clara, CA, February 2016. USENIX Association.

[LLC+14] Cheng Li, Joao Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 281–292, Philadelphia, PA, June 2014. USENIX Association.

[MGST70] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[MM03] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2Nd USENIX Confer-*

ence on File and Storage Technologies, FAST '03, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[MS14] O. Mutlu and L. Subramanian. Research Problems and Opportunities in Memory Systems. *Supercomputing Frontiers and Innovations: an International Journal*, 1:19–55, October 2014.

[NDR08] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *Usenix FAST*, 2008.

[NDT+08] Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony Rowstron. Everest: Scaling Down Peak Loads Through I/O Off-Loading. *Proc. of the USENIX OSDI*, December 2008.

[Net] NetApp. Flash cache. https://www.netapp.com/us/products/storage-systems/flash-cache/index.aspx.

[Orl93] James B Orlin. A Faster Strongly Polynomial Minimum Cost Flow Algorithm. *Operations research*, 41(2):338–350, 1993.

[QJP+07] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[Sch14] Greg Schulz. Tiered Storage: Enterprise SSHD and Flash SSD. https://storageio.com/Reports/SIO_IndustryPerspective_Seagate_SSHD_Mar17_2014.pdf, 2014.

[SLK+15] Ricardo Santana, Steven Lyons, Ricardo Koller, Raju Rangaswami, and Jason Liu. To ARC or not to ARC. In *Proc. of USENIX HotStorage*, 2015.

[SNI19] SNIA. MSR Cambridge Traces. http://iotta.snia.org/traces/388, October 2019.

[SSZ12] Mohit Saxena, Michael Swift, and Yiying Zhang. FlashTier: A Lightweight, Consistent and Durable Storage Cache. In *European Conference on Computer Systems, April 2012*, April 2012.

[Sto] Storage Networking Industry Association. The SNIA's I/O Traces, Tools, and Analysis (IOTTA) Repository. *http://iotta.snia.org/*.

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[THL+15] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, Santa Clara, CA, February 2015. USENIX Association.

[VKUR10] Akshat Verma, Ricardo Koller, Luis Useche, and Raju Rangaswami. SRCMap: Energy Proportional Storage Using Dynamic Consolidation. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, Berkeley, CA, USA, 2010. USENIX Association.

[Wil19] David P. Williamson. *Network Flow Algorithms*. Cambridge University Press, 2019.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[WOHL17] Kai Wu, Frank Ober, Shari Hamlin, and Dong Li. Early Evaluation of Intel Optane Non-Volatile Memory with HPC I/O Workloads. *arXiv e-prints*, page arXiv:1708.02199, Aug 2017.

[WPGA15] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC Construction with SHARDS. In *Proc. of USENIX FAST*, 2015.

[WSAP17] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache Modeling and Optimization using Miniature Simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498, Santa Clara, CA, July 2017. USENIX Association.

[YKH+20] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, 2020.

[ZHZ+18] P. Zuo, Y. Hua, M. Zhao, W. Zhou, and Y. Guo. Improving the Performance and Endurance of Encrypted Non-Volatile Main Memory through Dedupli-

cating Writes. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 442–454, Oct 2018.

[ZKAV20] Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. Optimal Data Placement for Heterogeneous Cache, Memory, and Storage Systems. *Proc. ACM Meas. Anal. Comput. Syst.*, 0(ja), 2020.

[ZLL+18] J. Zhang, P. Li, B. Liu, T. G. Marbach, X. Liu, and G. Wang. Performance Analysis of 3D XPoint SSDs in Virtualized and Non-Virtualized Environments. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1–10, 2018.

[ZPL01] Yuanyuan Zhou, James Philbin, and Kai Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, pages 91–104, Berkeley, CA, USA, 2001. USENIX Association.

[ZS15] Yiying Zhang and S. Swanson. A Study of Application Performance with Non-Volatile Main Memory. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, May 2015.

APPENDIX A

mOPT PROOF

Statement of the problem

Consider the following *offline* cache management problem: Suppose a sequence of requests $\mathscr{R} = \langle \sigma_1, \ldots, \sigma_T \rangle$ from a set Σ is known in advance and we are given an empty cache of size n. On each request, if it is already in the cache, our gain is 1; otherwise the item must be fetched from the backing store. When this happens, we may either choose to cache the item and pay a write cost of α or choose not to cache it. The problem is to design a cache replacement algorithm A which maximizes our gain; namely, a cache replacement policy which maximizes

$$\text{cost}(A, \alpha, n, \mathscr{R}) = (\text{number of hits}) - \alpha \cdot (\text{number of writes})$$

The problem is how to find an optimal replacement algorithm that attains

$$_{\text{m}}\text{OPT}(\alpha, n, \mathscr{R}) := \max_A \text{cost}(A, \alpha, n, \mathscr{R}).$$

In the traditional setting, when an item is fetched from a slower memory, it is necessarily written into the cache; this causes some item that is already in the cache has to be evicted if the cache is already full — hence the name cache "replacement" problem. It is well-known [MGST70] that the so-called **MIN** algorithm, which evicts an item in the cache whose next request is furthest in the future, is optimal in this setting.

Denote by $\text{MIN}(n, \mathscr{R})$ the number of cache hits **MIN** algorithm achieves for a request $\mathscr{R}$ with a cache of size n. By setting the write cost α to zero, we may compare the performance of the same size cache under these two different settings.

Theorem A.0.1 *For any cache size n and request sequence $\mathscr{R}$, we have*

$$\text{MIN}(n, \mathscr{R}) \leq {}_{\text{m}}\text{OPT}(0, n, \mathscr{R}) \leq \text{MIN}(n+1, \mathscr{R}).$$

Proof. The first inequality is trivial since choosing not to write to the cache is a privilege: the new cache can definitely simulate the behavior of **MIN** algorithm and achieve the same number of hits. For the second inequality, one can imagine one slot of the size-$(n+1)$ traditional cache is a special "temporary storage", and the rest n slots are normal storage. It can simulate the cache behavior of a modified size-n cache as follows. If the fetched item is stored in the modified cache, so does the traditional one, and the item is stored in the same place in the normal storage space. Otherwise, fetched item will be stored in the temporary storage. It follows that **MIN** algorithm with $(n+1)$-sized cache has at least the same number of hits as any cache algorithm of the modified cache with n-sized cache. □

Theorem A.0.2 *Given a request sequence $\mathscr{R} = \langle \sigma_1, \ldots, \sigma_T \rangle$ from a set Σ, with $q = |\Sigma|$, there is an algorithm that computes an optimal caching policy for a modified cache of size n, and runs in time $O(n \cdot (qT + T \log T))$.*

An algorithm for optimal cache replacement policy

We reduce the offline cache management problem to the MINIMUM COST FLOW PROBLEM. By constructing an instance of the MINIMUM COST FLOW PROBLEM, we first show that the minimum cost flow of the network must be integral-valued. Since all the edge weights in the constructed network are integers, this implies that the optimal flow can be decomposed into a set of edge-disjoint paths between the source and sink. Then we prove a one-to-one correspondence between the

The minimum cost flow problem

In the following, we use $\mathbb{R}^+$ to denote the set of non-negative real numbers.

A *network* is a quadruple $\{G,u,\mathbf{s},\mathbf{t}\}$, where $G=(V,E)$ is a directed graph, $u: E(G)\to\mathbb{R}^+$ is an edge capacity function, and $\mathbf{s}$ (the source) and $\mathbf{t}$ (the sink) are two specified vertices in $V(G)$. A *flow* in a network $\{G,u,\mathbf{s},\mathbf{t}\}$ is a real-valued function $f: E(V)\times E(V)\to\mathbb{R}^+$ satisfying the following two properties. (i) Capacity constraint: $f(i,j)\le u(i,j)$ for all $(i,j)\in E(G)$ and (ii) Flow conservation: for all $i\in V(G)\setminus\{s,t\}$, $\sum_{j\in V(G)} f(i,j)=\sum_{j\in V(G)} f(j,i)$. If (i,j) is an edge in G, then the non-negative quantity $f(i,j)$ is called the flow from vertex i to vertex j. The value of a flow f is defined as $|f|=\sum_{i\in V(G)} f(\mathbf{s},i)-\sum_{i\in V(G)} f(i,\mathbf{s})$.

A useful fact about flows in a network is the following theorem.

Theorem A.0.3 (Flow Decomposition Theorem [FJF62]) *Let $(G,u,\mathbf{s},\mathbf{t})$ be a network and let f be an* **s**-**t**-*flow in G. Then there exists a family $\mathscr{P}$ of* **s**-**t**-*paths and a family $\mathscr{C}$ of cycles in G along with a weight function $w:\mathscr{P}\cup\mathscr{C}\to\mathbb{R}^+$ such that for every edge $(i,j)\in E(G)$, $f(i,j)=\sum_{P\in\mathscr{P}\cup\mathscr{C}:(i,j)\in E(P)} w(P)$ and $\sum_{P\in\mathscr{P}} w(P)=|f|$. Moreover, if f is integral then all weights w can be chosen to be integral.*

Our approach for the offline modified cache policy problem is to reduce it to the MINIMUM COST FLOW PROBLEM, and then apply a well-known Edmond-Karp algorithm to solve the problem.

Definition A.0.4 (MINIMUM COST FLOW PROBLEM) *Given a directed graph $G=(V,E)$, a capacity function $u: E(G)\to\mathbb{R}^+$, a function $b: V(G)\to\mathbb{R}$, and a weight function $c: E(G)\to\mathbb{R}$, a* b-flow *in (G,u,c) is a real-valued function $f: E(V)\times E(V)\to\mathbb{R}^+$ satisfying the following two properties. (i) Capacity constraint: $f(i,j)\le u(i,j)$ for all $(i,j)\in E(G)$ and (ii) Flow constraint: for all $i\in V(G)$, $\sum_{j\in V(G)} f(i,j)-\sum_{j\in V(G)} f(j,i)=b(i)$ (that is, for every vertex i in the network, the net flow out of the vertex is equal to the "injected" flow $b(i)$ into that vertex). The* MINIMUM COST FLOW PROBLEM *is to find a b-flow f which minimizes the cost function $c(f):=\sum_{(i,j)\in E(G)} c(i,j)f(i,j)$.*

MINIMUM COST FLOW PROBLEM is one of the central problems in network flow research and has been extensively studied during the past half century; see e.g. [AMO93, KV12, Wil19] and references therein. To date, the fastest strongly polynomial[1] algorithm for the MINIMUM COST FLOW PROBLEM is due to Orlin [Orl93], with a running time $O(m\log n(m+n\log n))$ for networks with n vertices and m edges.

Edmond-Karp algorithm

Early network flow algorithms rely crucially on the concept of *residual network*, introduced by Ford and Fulkerson [FJF62]. Given a flow network G and a flow f , the residual network G_f consists of the same set of vertices as G, but with additional edges and modified edge capacity. Specifically[2], $u_f(i,j) = u(i,j) - f(i,j)$ if $(i,j) \in E(G)$, and $u_f(i,j) = f(j,i)$ if $(i,j) \notin E(G)$. Moreover, the edge weights of the original edges in G are unchanged, and the newly added edges (j,i) has weights $c_f(j,i) = -c(i,j)$.

The Edmond-Karp algorithm [EK72] for the MINIMUM COST FLOW PROBLEM is built on the following basic result (see e.g. [FJF62]), p. 121.

Theorem A.0.5 *Let (G,u,c,b) be an instance of the* MINIMUM COST FLOW PROBLEM *and let f be a minimum cost b-flow. Let P be a shortest (with respect to edge weight c_f)* **s**-**t** *path in G_f Let f' be a flow obtained by augmenting f along path P and denote the resulting flow function as b'. Then f' is a minimum cost b'-flow.*

---

[1]Roughly speaking, an algorithm is said to be *strongly polynomial* if its running time is a polynomial only of the problem instance size (in our case, the number of vertices and edges of the network) but independent of the parameters of the problem instance (in our case, the quantities such as capacity u, cost c and flows b), as long as we can assume that all basic arithmetic operations involving these parameters can be computed in $O(1)$ time.

[2]Without loss of generality, for convenience, we assume that in the network $(i,j) \in E(G)$ implies $(j,i) \notin E(G)$. If this condition is not met, we may modify G by adding auxiliary vertices. Such a condition is certainly satisfied for the network constructed in this paper.

The Edmond-Karp algorithm starts from an empty flow $f = 0$. Then repeat the following process until no such vertices $\mathbf{s}$ and $\mathbf{t}$ can be found: Choose a vertex $\mathbf{s}$ with $b(\mathbf{s}) > 0$, choose a vertex $\mathbf{t}$ with $b(\mathbf{t}) < 0$, such that $\mathbf{t}$ is reachable from $\mathbf{s}$ in G_f; Augment the flow f along a shortest path between $\mathbf{s}$ and $\mathbf{t}$ with flow equaling the minimum edge capacity along the path; update b.

Therefore, computing a minimum cost flow is reduced to finding a shortest $\mathbf{s}$-$\mathbf{t}$ path in the residual network defined by the previous flow, and augmenting the flow along this shortest path. As computing a shortest $\mathbf{s}$-$\mathbf{t}$ path in a general graph with negative edge weights is expensive (the best known Bellman-Ford algorithm takes $O(nm)$ time), the key insight of Edmond-Karp is that we can introduce a "potential function" $\pi_f(i)$for each vertex i in the residual graph, defined as the shortest path distance $\delta(\mathbf{s}, i)$ between $\mathbf{s}$ and i in G_f, and use $\bar{c}_f(i,j) = c_f(i,j) + \pi_f(i) - \pi_f(j)$ — which are guaranteed to be non-negative — as the edge weights to perform shortest path calculation. By employing Fibonacci heap [FT87], Edmond-Karp algorithm runs in $O(B(m + n\log n))$ time, where $B = \frac{1}{2}\sum_{i \in V(G)} |b(i)|$ is the size of the flow, n is the number of vertices and m is the number of edges in the network. See e.g. [KV12] for a detailed description and analysis[3].

Network construction

Let $\Sigma = \{\sigma_1, \ldots, \sigma_q\}$ denote the set of possible requested items with $q = |\Sigma|$. Without loss of generality, assume that every symbol $\sigma \in \Sigma$ appears in the request (otherwise, we can remove it from Σ). Let $\mathscr{R} = \langle \sigma_1, \ldots, \sigma_T \rangle$ be a sequence of T requests. Therefore, each request can be represented by a tuple (σ, t), where $\sigma \in \Sigma$ and $t \in \{1, \ldots, T\}$. Abusing notation, we will use $\mathscr{R}$ to also denote the set of such request tuples. For every $\sigma \in \Sigma$,

[3]The running time stated here applies only to the network constructed in this paper; since our original network does not have any negative weighted edges, so the initial run of the Bellman-Ford algorithm to find a shortest path from $\mathbf{s}$ to every other vertex in G is avoided in our case.

let $\mathrm{fi}(\sigma)$ be the time stamp at which σ first appears in the request, and $\mathrm{la}(\sigma)$ be its last appearance in the request. Finally, for any $(\sigma,t) \in \mathscr{R}$, let $\mathrm{ne}(\sigma,t) = \min\{t' > t \mid (\sigma,t') \in \mathscr{R}\}$ denote the next time stamp at which symbol σ appears in the request after time t, and set it to $T+1$ if (σ,t) is the last occurrence of σ.

Given a modified cache of size n, write cost α, and a sequence of T requests $\mathscr{R} = \langle \sigma_1, \ldots, \sigma_T \rangle$, an instance (G,u,c,b) of the MINIMUM COST FLOW PROBLEM is constructed as follows.

Vertices

Let $V' \subset \Sigma \times \{1,2,\ldots T\}$ be the set of all request tuples; that is $V' = \{(\sigma,t) \mid$ the request at time t is $\sigma\}$. Then $V(G) = \{\mathbf{s},\mathbf{t}\} \cup V'$, where $\mathbf{s}$ is the source and $\mathbf{t}$ is the sink.

Edges

Firstly, for every $\sigma \in \Sigma$, add an edge from $\mathbf{s}$ to $(\sigma, \mathrm{fi}(\sigma))$ of weight $1+\alpha$. We will refer to these edges as *source edges*. Also add an edge from $(\sigma, \mathrm{la}(\sigma))$ to $\mathbf{t}$ of weight 1, call them *sink edges*.

Secondly, for every vertex (σ,t) such that $t < \mathrm{la}(\sigma)$ (i.e. t is the not the last time stamp at which σ appears in the request), add an edge from (σ,t) to $(\sigma, \mathrm{ne}(\sigma,t))$ of weight $\mathrm{ne}(\sigma,t) - t - 1$. Note that the weight is a non-negative integer. We refer to such an edges as a *straight edge*. If $\mathrm{ne}(\sigma,t) - t > 1$ (i.e., there are some other symbols between the current appearance of σ and the next appearance of σ), then for every $t < t' < \mathrm{ne}(\sigma,t)$, add an edge from (σ,t) to (σ',t') of weight $t' - t + \alpha$. Since σ' is necessarily different from σ, such an edges will be called a *slanting edge*.

Finally, add an edge from $\mathbf{s}$ directly to $\mathbf{t}$ of weight $T+1$. All edges in our network have unit capacity, except for this last edge from $\mathbf{s}$ to $\mathbf{t}$, which has capacity n.

b-flow

We set the b-flow as follows: $b(\mathbf{s}) = n$, $b(\mathbf{t}) = -n$, and $b(i) = 0$ for every $i \in V'$.

Fact A.0.6 *The constructed network G satisfies that $|V(G)| = T+2$ and $|E(G)| \leq q(T+2)+1$.*

Proof. The size of $V(G)$ is straightforward. To calculate $|E(G)|$, first note that there are exactly q source edges, q sink edges, and another edge from source directly to sink. For any vertex $(\sigma,t) \in V'$ its total number of outgoing edges (counting both straight edge and slanting edges) is exactly $\mathrm{ne}(t) - t$. Summing over all such vertices with a fixed σ gives

$$\sum_{t:(\sigma,t)\in V'} \text{out-deg}((\sigma,t)) = \sum_{t:(\sigma,t)\in V'} (\mathrm{ne}(t) - t) = T - \mathrm{fi}(\sigma) < T.$$

Since there are q such symbols, the total number of such edges is at most qT. It follows that $|E(G)| \leq q(T+2)+1$. □

Completing the proof

The key observation of the proof is that there is a one-to-one correspondence between the paths from $\mathbf{s}$ to $\mathbf{t}$ in the network and the states of content of storage units in the cache. Specifically, a straight edge corresponds to a cache hit and a slanting edge corresponds to writing a new symbol into the cache unit.

We note that the constructed network (G,u,c,b) has the following nice properties. Firstly, the existence of a capacity n edge from $\mathbf{s}$ to $\mathbf{t}$ makes the instance of MINIMUM COST FLOW PROBLEM *feasible*. It is easy to see that a flow $f_0 \in \mathbb{N}$ along the $(\mathbf{s},\mathbf{t})$ edge corresponds to that f_0 slots of the cache have never been used, making the effective cache size $n - f_0$. For this reason, in the following, we assume that the flow along this edge is zero. Equivalently, we may assume that this edge is removed from G and we try flow value

b from 1 to a maximum value n' at which an n'-flow between **s** and **t** is still feasible in the modified network. Then n' is the maximum cache size can be fully utilized by the request sequence. Consequently, since now all edge capacity is 1, by the Flow Decomposition Theorem, the minimum cost flow found by Edmond-Karp theorem can be decomposed into n edge-disjoint **s**-**t** paths in G.

If we assign a time stamp 0 to the source and a time stamp $T+1$ to the sink, then every edge in the network is of the form $((\sigma,t),(\sigma',t'))$ with $t < t'$. Moreover, the weight of such an edge is either $t'-t+\alpha$ if a "write" occurs (that is, symbol of the vertex changes[4]), or $t'-t-1$ if a cache hit occurs, or just $t-t'$ if neither occurs (this applies only to the edge from **s** to **t**, which corresponds to an unused cache unit). Consequently, the total weight (hence the cost of a unit flow) of every path P from **s** to **t** is

$$c(P) = T+1+\alpha\cdot(\text{number of writes}) - \text{number of hits}.$$

Since the total flow is n, summing the total cost of all n unit flows together gives that the minimum cost of the flow is

$$c(f) = \sum_{\text{disjoint } \mathbf{s}\text{-}\mathbf{t} \text{ path } P} c(P) = n(T+1) - {}_{\mathrm{m}}\mathrm{OPT}(\alpha,n,\mathscr{R}).$$

Or equivalently,

$${}_{\mathrm{m}}\mathrm{OPT}(\alpha,n,\mathscr{R}) = n(T+1) - c(f).$$

Proposition A.0.7 *For n and T be positive integers, $0 \le \alpha < 1$, Σ be a finite set of size q, and $\mathscr{R} = \langle \sigma_1,\ldots,\sigma_T\rangle$ be a sequence of request from Σ. Let (G,u,c,b) be the instance of the* MINIMUM COST FLOW PROBLEM *constructed in Section A from an instance of the online modified cache problme $(n,\alpha,\Sigma,T,\mathscr{R})$. Then there is one-to-one correspondence between the cache states of the n units and the n edge-disjoint path of a feasible flow from* **s** *and* **t**.

[4]We view the empty state as a special symbol; therefore an edge from **s** to a normal vertex involves symbol change.

VITA

STEVEN LYONS
slyon001@fiu.edu

July 2020	M.S. Computer Science Florida International University Miami, FL
December 2014	B.S. Computer Engineering Florida International University Miami, FL
January 2016 - Present	Graduate Research Assistant Florida International University Miami, FL
May 2018 - August 2018	Intern Google, Inc. Sunnyvale, CA
January 2015 - December 2015	Graduate Teaching Assistant Florida International University Miami, FL
November 2013 - December 2014	Undergraduate Research Assistant Florida Internaional University Miami, FL

PUBLICATIONS AND PRESENTATIONS

- mOPT: Optimizing Hit-Rate and Write-Rate for Non-Datapath Caches (In Submission)
  ACM Transactions on Storage
  Authors: Steven Lyons, Liana Valdes, Raju Rangaswami
- Finding Optimal Non-Datapath Caching Strategies via Network Flow (In Submission)
  Information Processing Letters
  Authors: Steven Lyons, Ning Xie, Raju Rangaswami
- To Cache or Not to Cache (Submitted)
  ACM Transactions on Storage
  Authors: Steven Lyons, Liana Valdes, Raju Rangaswami

- Learning Cache Replacement with CACHEUS
  USENIX Conference on File and Storage Technologies (FAST 21)
  Authors: Liana Valdes, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan

- Driving Cache Replacement with ML-based LeCaR
  USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)
  Authors: Giuseppe Vietri, Liana Valdes, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan

- To ARC or Not To ARC
  USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)
  Authors: Ricardo Santana, Steven Lyons, Ricardo Koller, Raju Rangaswami, and Jason Liu