Measurement and Analysis Methods of Performance Problems in Distributed Systems

By

Lei Zhang
Doctor of Philosophy

Computer Science and Informatics

Measurement and Analysis Methods of Performance Problems in Distributed Systems

By

Lei Zhang
B.A., Tsinghua University, Beijing, 2015
M.Sc., Georgia Institute of Technology, GA, 2017

Advisor: Ymir Vigfusson, Ph.D.

An abstract of
A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science and Informatics
2021

Abstract

Measurement and Analysis Methods of Performance Problems in Distributed Systems
By Lei Zhang

Today's distributed systems invest significant computational and storage resources to accommodate their large scale of data, but more resources do not automatically improve performance. To deliver high performance, new types of large-scale solutions, such as the cloud computing and microservices paradigms, follow the design of deploying loosely coupled components that perform but, in the process, make it harder to maintain a global view of system performance. With the ensuing growing complexity of system architectures, diagnosing and understanding performance problems has become both critically important and highly challenging.

The aim of this thesis is to fill in some missing but significant parts towards monitoring and analyzing performance problems in distributed systems, by asking the question: What is the performance bottleneck of distributed systems performance, and how should we improve it? First, my thesis proposes a novel retroactive tracing abstraction where full telemetry information about a distributed request can be retrieved "back in time" soon after a problem is detected without unduly burdening any node in the system, with an always-on distributed tracing system. Second, my thesis frames the challenges of data placement in modern memory hierarchies in a generalized paging model outside of traditional assumptions, and provides an offline data placement algorithm towards optimal placement decisions. Last, my thesis derives a rule-of-thumb expression for cache warmup times, specifically how long caches in storage systems and CDNs need to be warmed up before their performance is deemed to be stable.

Measurement and Analysis Methods of Performance Problems in Distributed Systems

By

Lei Zhang
B.A., Tsinghua University, Beijing, 2015
M.Sc., Georgia Institute of Technology, GA, 2017

Advisor: Ymir Vigfusson, Ph.D.

A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science and Informatics
2021

Acknowledgments

First of all, I want to thank my advisor, Prof. Ymir Vigfusson. Ymir quickly turned me in the right direction in my second half of Ph.D. career. He not only taught me how to deliver high quality research works, but more importantly let me know how to recover from adversity and lead me back as an optimistic and positive student on system research. Ymir brought me to new research directions on caching, where he also showed me how to successfully leverage my mathematical background on system research. I feel grateful to pursue my Ph.D. with Ymir as my advisor.

I thank Prof. Avani Wildani and Prof. Nosayba El-Sayed, who not only serve on my committee but also provide me with many helps during my Ph.D. time. I also feel grateful for Prof. Robbert van Renesse, who offered me great help during my Ph.D. time and also joined my thesis committee.

I would like to thank my collaborator, Irfan Ahmad and Prof. Jonathan Mace, with who I closely worked on my major research projects and who offered me significant advice like my second advisors. I also thank my colleges from or was from Emory University, Juncheng Yang, Reza Karimi, and other members from my group, with whom I worked together and had a great time.

Before working with Ymir, I spent my first three years of Ph.D. at Georgia Tech, where I took graduate courses and enhanced my background in system research. I appreciate that period of time. I'm grateful to all professors and students I know and worked together with. Though that's not a perfect time, I'm glad to what I learned and experienced at Georgia Tech.

I'm especially grateful for, and in memory of, Prof. Karsten Schwan, my first Ph.D. advisor. Sadly I cannot work with you until the end, but I hope I've accomplished your requirements as your Ph.D. students.

I want to thank my parents. Growing in a family with two Ph.D. in computer science is somehow different for a kid. I appreciate this special education since I was very young.

Finally, I want to thank my fiancee, Dan Li. We spent the same period of time abroad to pursue our Ph.D., keep supporting each other. I cannot imagine how things would go without Dan, especially during the tough period of my career. Besides, special thanks to our lovely cat, Riceball, who serves as my research assistant and a loyal reader of my papers.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Performance is Key

Distributed systems involve more computational and storage resources today for larger data scales, but performance is not automatically improved. Data scale is growing rapidly on modern systems including social networks and web search systems. Facebook claims their data warehouse stores upwards of 300 PB of Hive data in 2014, with an incoming daily rate of about 600 TB [3]. Google is also estimated to maintain more than 10 EB data in their warehouse [28]. Together grown is the demand for service quality. Performance is often directly related to cost benefits, where small improvements in performances could lead to large savings on production systems. Akamai reported that each additional 100ms of latency experienced by end-users decreased the sales by 7% [1]. Similarly, Amazon also claimed that a 100ms reduction on latency leads to 1% lost on sales[74]. Providing guaranteed performance in distributed systems should be considered one of the highest priorities.

Unfortunately, traditional system solutions do not scale by themselves, even with faster processors and storage hardware. Distributed systems are driven by more machines, more storage nodes, and more complex architectures, which simultaneously make system development a harder issue. New types of large-scale solutions, including cloud computing and microservices, follow the design idea to deploy loosely coupled components. While system developers can build decoupled applications easier, a global view of system performance

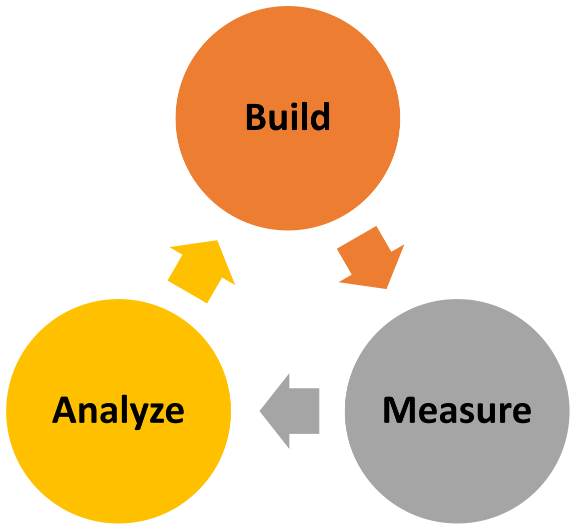


Figure 1.1: Developing distributed systems [18].

is missed. Applications always follow domain-specific design patterns on design logic and resource utilization, leaving all heavy lifts of global performance optimization to the system architecture.

Performance improvement is not a one-way process. On one hand, estimating performance ahead of execution, with monitoring, testing, and verifying methods, can provide a basic sense of how the system behaves. However, distributed system performance is usually undetermined and cannot be fully tested before execution, where **performance problems** may occur during execution. Thus solving performance problems is even more important towards better improving the system.

Improving system performance is hard if it's not well understood. This includes measuring, analyzing, and understanding performance patterns. Figure 1.1 [18] presents a circle of system building procedures. When building a large-scale system with desired performance, performance measurement can expose general, hidden, or indiscoverable problems, while performance analysis derives how to further improve performance, especially why is the current performance not as expected. This drives the importance of carefully quantifying system performances. Performance quantification includes several considerations:

- Performance should be carefully measured. Within the grown system scale, it also becomes more challenging to monitor and diagnose unexpected performance problems, including failures, lags, and bottlenecks. Troubleshooting those issues requires a global view across different machines, processes, and networks. Distributed tracing is the solution to those questions, by deploying end-to-end tools on machines for recording and analyzing root causes of performance symptoms.

- Performance should be well understood. Caches have been proved as the general solution to speed up systems, understanding how cache works in distributed systems helps maximize the power of such improvements. However, caches are still always used as a black box. We should understand where the bottleneck occurs, what metrics are the core of those bottlenecks, and how the system could be further optimized.

Though diagnosing distributed system performance problems is an extremely hard problem, this thesis argues that we lack a thorough understanding of distributed system performances, and aims to point out some significant but missing points on monitoring and analyzing performance problems in distributed systems. Specifically, this thesis aims to answer the question: **What is the performance bottleneck of distributed systems performance, and how should we solve it?** Since this is an expansive and deep question I expect significant work to happen in the coming years. For this thesis, I will concentrate on several key challenges towards addressing the proposed solutions.

1.1.1 Challenges

Diagnosing and troubleshooting distributed systems is onerous, because performance related issues and root causes scatter across different machines without indications or practically predictions ahead of time. Performance diagnosis relies on recording large amounts of logs, metrics, or traces and with a branch of tools to achieve such root cause analysis. This raises a practical issue when the amount of log data is huge. Current diagnosis solutions sidestep these costs by tracing off the data volume, either by only gathering the most pertinent information or by sampling on a small subset of requests or time periods with the hope that performance issues would occur on those samples. In all those cases no one knows prior to a problem occurring, so collecting details for diagnosing a rare problem is almost impossible. We need to think about how to solve the problem to provide performance diagnosis guarantee.

Even with the ability to know performances, we should carefully design system components for performance improvement. Caching systems are broadly used today but usually used in a simple form. There is a lack of understanding the usage, architecture, and optimal objective of caching systems. Unfortunately, most caching systems in production still follow the traditional cache idea that only focuses on minimizing cache misses. Those systems are treated as a simple black-box middleware to hold “hot” data for an uncertain amount of speed-up. A trending cache systems design is to firstly understand the optimal boundary and further design caches towards it. However, many recent theoretical works miss the consideration of realistic caching system usages.

1.1.2 Opportunities

As we addressed the many missing parts of system performance quantification, there exist more chances to solve the problems. Performance measurement is not a blank world. Generating performance tracing data is actually cheap, while the following steps are the expensive aspects. Operators can generate as much data as they like, and the real bottlenecks for performance measurement are in the storage to persist that data for the future, and network costs for centralizing the data. Besides, most performance data is superfluous. Normal operations are by definition uninteresting for tracking down a specific problem. Finally, there is a large class of problems where the symptoms of a performance problem can be detected quickly. For example, the moment a low-latency system observes an outlier request, *e.g.*, one that took seconds to be processed, something has evidently gone wrong and an investigation is warranted. In these cases, the relevant data for troubleshooting does not extend far back in time. Those observations indicate that we have the chance to properly deal with a small amount of data for widely and speedily diagnosing the “interesting” performance problems.

From the perspective of building cache systems, there are many realistic aspects pushing us to rethink the cache design. As nascent memory technologies muddy the traditional distinction between layers in terms of storage capacity, latency, power, and costs, the assumptions underlying data placement decisions need to be revisited. Since cache performance relies on hardware performances, this opens a great chance to design caching system architecture with those new memory hardware, with a consideration of high performance hierarchies. Byte-addressable non-volatile memory (NVM), as one example, is slated to deliver larger capacity than DRAM at competitive latency, with currently available NVM hardware (e.g., Intel Optane DC Persistent Memory [89, 69]) incurring $2 - 5\times$ the read latency and $4 - 10\times$ the write latency of DRAM, far closer than the 2–3 orders of magnitude performance differences between DRAM and SSD.

1.2 Contributions

The thesis proposes to fill in the missing parts of distributed system performance quantification by making the contributions as listed below.

- **Retroactive distributed tracing in runtime.** I propose a novel retroactive tracing abstraction where full telemetry information about a distributed request can be retrieved "back in time" soon after a problem is detected without unduly burdening any node in the system. I build an always-on distributed tracing system, named Hindsight, that enables retroactive tracing. The key idea is that all trace data about requests is written into a fixed-size, thread-local buffer. The buffer is flushed when full and a buffer with older data is recycled. When an application detects a problem and wishes to persist the relevant trace data, it invokes Hindsight's trace collection mechanism and collects the full request traces while aggressively curtailing overhead.
- **Offline optimal placement for memory hierarchy.** I frame the challenges of data placement in modern memory hierarchies in a generalized paging model outside of traditional assumptions. I design and build CHOPT, an offline data placement algorithm for providing optimal placement decisions as the upper bound of performance gain for any data placement algorithm. CHOPT utilized spatial sampling to provide practical and accurate approximations. Experimental results on a wide range of data show that CHOPT can expose the opportunities to improve latency performance ranging between 8.2% and 44.8% on average versus the standard BELADY algorithm. As an

additional contribution, this thesis also theoretically and experimentally evaluate the spatial sampling performances on real world workloads.

- **Cache warmup time analysis.** I derive a rule of thumb expression for cache warmup times, specifically how long caches in storage systems and CDNs need to be warmed up before their performance is deemed to be stable. Warmup time estimation allows CDN operators to compute the required redundancy and extra capacity to maintain a level of service in failure scenarios. This first provides a concrete definition of cache warmup time, that is, a cache server has warmed up when its cache hit rate over time is and stays comparable to that of an identical cache service that processed the same workload but suffered no downtime. This thesis then analyzes dozens of traces across workloads collected from diverse systems, ranging from block accesses of virtual machines in storage systems to cache accesses of large CDN providers. Our experiments show that simple parameters concentrate at specific values for each type of workload. Our simulation results indicate that the formula provides an accurate expression for operators to estimate their cache server warmup time.

1.3 Thesis Overview

This thesis follows the general process as shown in Figure 1.1. Chapter 2 provides the necessary backgrounds and related works, especially on distributed tracing and distributed caching. Chapter 3 introduces the proposed method to measure edge case performance problems. Chapter 4 introduces an offline analysis work to understand the optimal placement policy on cache and memory systems. Chapter 5 introduces a practical case study on quantifying cache warmup time.

Chapter 2

Background

In this chapter, we provide the necessary background of understanding distributed system performance. We specifically introduce how performance problems occur in modern distributed systems, and what are the state-of-the-art methods for performance improvement.

Performance Problems. In distributed systems, performance problems refer to unexpected behaviors of performance metrics. In this thesis, we consider all types of metrics but specifically take latency as a major example, given that latency is one of the most important metrics of distributed system performances. A latency issue means the execution time is longer than expected. This falls into two categories:

- Performance bug occurs, where the latency is larger than normal. A commonly used metric is P99 latency, which represents the 99th latency percentile so each request with latency higher than P99 latency is counted as the slowest 1% requests among all.

- Performance is not good enough as designed or expected. Note that there are many system components designed to improve performance. Cache is broadly used to store data for repeated requests in the near future and reduce request execution time. Cache itself is a system and it can also have performance problems when not providing expected performance speed-up.

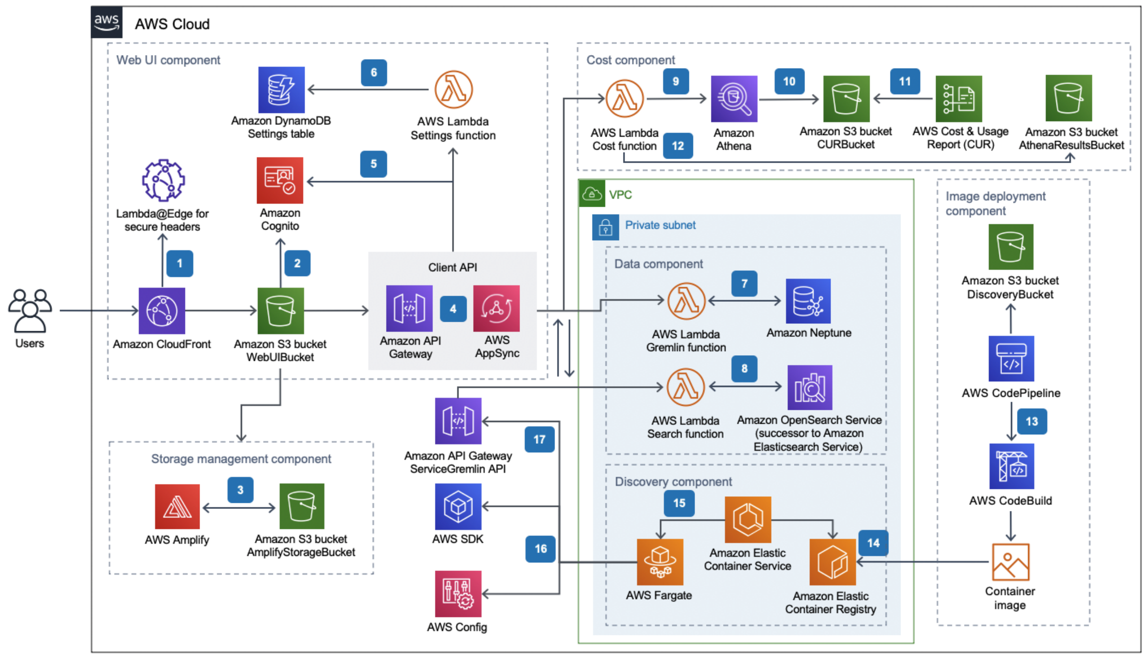

AWS Cloud
Web UI component
Amazon DynamoDB Settings table
AWS Lambda Settings function
Lambda@Edge for secure headers
Amazon Cognito
Client API
Users
Amazon CloudFront
Amazon S3 bucket WebUIBucket
Amazon API Gateway
AWS AppSync
Storage management component
AWS Amplify
Amazon S3 bucket AmplifyStorageBucket
Amazon API Gateway ServiceGremlin API
AWS SDK
AWS Config
Cost component
AWS Lambda Cost function
Amazon Athena
Amazon S3 bucket CURBucket
AWS Cost & Usage Report (CUR)
Amazon S3 bucket AthenaResultsBucket
VPC
Private subnet
Data component
AWS Lambda Gremlin function
Amazon Neptune
AWS Lambda Search function
Amazon OpenSearch Service (successor to Amazon Elasticsearch Service)
Discovery component
AWS Fargate
Amazon Elastic Container Service
Amazon Elastic Container Registry
Image deployment component
Amazon S3 bucket DiscoveryBucket
AWS CodePipeline
AWS CodeBuild
Container image
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17

Figure 2.1: AWS Microservice Architecture Example [2].

2.1 Modern Distributed Systems

As we discussed in chapter 1, distributed systems grow with higher demand on performance today. We first introduce some examples of large-scale modern distributed architectural patterns that we focus on in this thesis.

Cloud Computing. Distributed systems are developed today with huge demand for processing massive requests and data. This raises questions on both service quality and the complexity to design, build, and maintain systems. Such new demands push the development of cloud computing, where massive computational and storage resources are deployed for application developers to build their systems or services on top of. Cloud computing provides the basis of today's distributed systems where we have to build and deploy systems at scale and traditional system architectures are limited.

Microservice. A trending distributed system design is the microservice architecture, where system components are fully independent, loosely coupled, and usually single-purposed. Building large-scale systems is then transferred to putting those microservices together as a monolithic service as a functional system. Figure 2.1 shows an example of building a microservice based application with AWS components. Microservice architecture benefits distributed system design and building by simplifying the system development process, enabling heterogeneous systems where different subsystems can be developed with different programming languages, and simplifying the testing, debugging, and maintaining system performances [76].

Heterogeneous Memories. The memory hierarchy has been a guiding model over decades of system research, giving system designers a framework for reasoning about efficiency trade-offs in architectures that combine different memory hardware, ranging from multi-core processors to large-scale distributed web caches to multi-tier storage systems. The historical optimization and limitation of CPU caches, the quintessential memory hierarchy, of not allowing blocks in lower layer caches to be directly addressed have influenced the models and algorithms deployed in other settings, even where layers are directly addressable. Addressability becomes important as new memory hardware simultaneously diversifies the characteristics of memory components (for instance, data volatility and bandwidth) and decreases the performance differences (less pronounced latency or capacity difference between layers, for example).

Specifically, non-volatile memory is an existing concept but recently released products that significantly contribute to modern memory hierarchies. Intel recently released the Optane DC Persistent Memory [89] whose load and store latency are within the same order of magnitude as regular DRAM. Other NVM technologies are under development, including Spin-Transfer-Torque RAM (STT-RAM) and 3D-XPoint, and are expected to reach similar performance. Table 2.1 shows a performance comparison of DRAM, NVM, and NVMe memories. Intel Xeon scalable processors currently support DRAM and NVM together in their main memory systems [14, 37], with NVM poised to sit between DRAM and the hard drive as an additional layer in the memory architecture.

2.2 Performance Symptoms

We discuss more in detail where performance problems occur in distributed systems.

2.2.1 Control Plane

The control plane in distributed systems includes both processing and execution on each system component, and network topologies for data transfer. Control plane can be represented as control flow graph (CFG). With the growing distributed system scale, CFG becomes huge and complex for real world production systems. This is especially true for microservice architectures where system components are fully decoupled.

Microservice vs. monolithic. Building distributed systems with microservice architecture moves the heavy lifting to system operators that design and maintain the monolithic system. Even assuming performances are well tested on individual subsystems, distributed systems serve as a whole and global performance is the most intuitive metric of service quality. Optimal local performance on all components does not guarantee optimal global performance.

Such comparison is true not only for performance optimization, but also for performance problems. We discuss why we need a monolithic view of performance problems.

- A component could become a performance bottleneck. A component can meet congestion problems when it serves as a scheduler, load balancer, or when contains a job queue. Such a component can only observe that many requests are stalled with additional latency, but not how the requests are further affected. Such nodes can be buggy because of some improper design that's not compatible with the overall system. Such nodes can also be not buggy but only busy. However, understanding such performance problems is impossible only with the performance of that component, but requires looking at many components at the same time.

- The root cause of a performance problem may be far away from where it's observed. For example, a node could observe a slowly executed request by capturing a slow response. This is typically true for some RPC-based systems, where requests are usually handled with a root node. However, the performance problem(s) usually occur not on this node but along other nodes processing the node. It could result from a buggy component, or even no performance bugs but a combination of hardware and

	DRAM	**NVM**	**NVM Block Devices**	**TLC Flash**
Load Latency	70ns	180-340ns	10μs	100μs
Store Latency	70ns	300-1000ns	10μs	14μs
Max. Load Bandwidth	75GB/s	8.3GB/s	2.2GB/s	3.3GB/s
Max. Store Bandwidth	75GB/s	3.0GB/s	2.1GB/s	2.8GB/s

Table 2.1: Memory Hierarchy Characteristics of DDR4 DRAM, NVM (Intel Optane DC Persistent Memory), NVMe block device (Intel Optane SSD DC), and TLC flash device [89, 69, 35]

software. Thus the performance problem is hard to analyze without a global view.

Thus we still need a monolithic view of system performance, which is even more important for today's distributed systems. We discuss this in chapter 3.

2.2.2 Data Plane

Besides the control plane of distributed system architecture, data placement and movement also significantly affect system performance. Data placement refers to the system design, both of storage and computation, to assign data on nodes that can be later retrieved [128]. Data placement aims to optimize system performance in different metrics like locality, load balancing, data availability. In this thesis we still focus on latency, the same as discussions in other scenarios. Different data placement policy results in different data plane for data access, which may systematically affect performance. For example, a bad placement design might require a computational node to always access data from a slower storage node, which introduces very high latency.

Data placement and movement play even more important roles in modern memory hierarchies as we discussed above. As nascent memory technologies muddy the traditional distinction between layers in terms of storage capacity, latency, power, and costs, the assumptions underlying data placement decisions need to be revisited. Byte-addressable non-volatile memory (NVM), as one example, is slated to deliver larger capacity than DRAM at competitive latency, with currently available NVM hardware (*e.g.*, Intel Optane DC Persistent Memory [89, 69]) incurring $2-5\times$ the read latency and $4-10\times$ the write latency of DRAM (Table 2.1), far closer than the 2–3 orders of magnitude performance differences between DRAM and SSD. Similar data placement challenges exist in Non-Uniform Memory Access (NUMA) architectures, Non-Uniform Cache Access (NUCA) architectures, multi-tier storage systems, distributed caching systems, and across the CPU-GPU divide to name a few examples.

Caching techniques. Cache refers to more than traditional CPU caches today. In distributed systems, cache systems play one of the most important roles to improve service quality by storing recently accessed data in fast memory. With the trending of in-memory systems, it's proper to understand memory hierarchy as a cache and thus transfer the data placement and movement problem into designing proper caching strategies. Caching techniques are also widely applied in distributed systems, not only for performance improvement (with prefetching), but also other considerations like isolation for privacy (with multi-tenancy and shared cache design). The key metric of cache performance is the cache miss ratio, as the number of cache misses over all received requests. Cache miss always refers to higher data access latency because data has to be accessed from the slower storage. The golden design goal of cache strategy is to minimize the cache miss ratio.

Note that traditional cache discussion is always based on a two-layer cache model with one faster cache layer and the other slower storage layer. In modern memory hierarchies where we usually have more than two layers, cache design needs to be reconsidered. For example, even the cache miss ratio requires re-definition because it's not clear to define a "slower layer" then. We discuss this in chapter 4.

2.3 Quantifying Performance

Quantifying system performance, and furthermore performance problems, requires a two-fold solution: measure the performance, and analyze how it is. This thesis focuses on developing measurement and analysis methods for distributed system performances. With the above background on performance problems in distributed systems, we go through some state-of-the-art solutions on quantifying performance problems.

2.3.1 Measurement Methods

Measuring performance serves as the first step to understanding and quantifying how system works. People conclude many concepts which are internally overlapped but still different. We list some useful concepts for distributed system performances here.

Logging. Logging is the basic method to keep track of everything happened on a system. Log messages record events during the execution of a system with timestamps. Logging systems are designed to keep as much log data as possible, which is usually the only source of rich information to dig into the root cause of a failure. Log messages can also serve for analyzing performance behaviors. The pros and cons are very obvious for logging: it can keep most rich runtime information, but it's usually very expensive thus limited on the ability to keep a large amount of log data [170]. Besides, logging is mostly used for systems on a single node. Though some previous works focus on logging for multi-thread environments, it's not natural to obtain logging messages across nodes.

Profiling. Compared with logging, profiling aims to measure performance metrics, which are usually aggregated. Profiling aims to collect fine-grained runtime performances while it's executed. Performance profilers can be applied from low level metrics like CPU usage and bandwidth to high level metrics like application behaviors. There are many well used performance profilers like perf, gprof, Valgrind, and gperftools which are powerful to collect metrics in many use cases. Like logging, however, profiling can also be expensive to collect large amounts of data.

Tracing. Distributed tracing is designed to collect events across machine or system component boundaries. In other words, tracing aims to solve the missing parts of distributed system performance measurement. Typical tracing systems focus more on application level context propagation to retrieve how a request is executed across different nodes. Tracing outputs can be not only request traces but also service maps like CFG. Though many people understand tracing as for producing only control flow relations, general tracing systems should also include system information as well, partly overlapped with logging systems. Tracing systems serve as the major performance quantification tool for distributed systems, especially for bugs not resulting from programs on a single node. Almost all major production systems apply tracing systems for performance debugging, including Dapper (used by Google) and OpenTracing framework (Facebook, Twitter, Uber, etc.). Ebpf is also a popular kernel level tracing system for low level tracing, which is applied not only on Linux kernels but also on storage systems.

2.3.2 Analysis Methods

Understanding system performances is vitally important for system development. Improving performance usually contains several steps. Before execution, performance is evaluated through debugging and testing methods; During execution, performance should be carefully measured. However, it's also very important to analyze performance after execution, by digging into collected execution data and performance metrics to debug or find out hidden performance problems. Especially for distributed systems, performance analysis is sometimes the only way to reveal performance bugs because static debugging processes cannot present performance issues on runtime.

Offline Optimal Analysis. Offline analysis represents getting log data from execution and analyzing it afterward. Though this cannot provide online feedback on system performance, it enables optimal performance analysis. Optimal analysis can not only provide performance bounds of potential improvements, but also indicate performance patterns towards better system design. There are many practical works on offline optimal placement or caching analysis. Belady's MIN [45] is known as the standard offline optimal caching algorithm for basic cache assumptions. To the best of our knowledge, Berger et al. [49] and Li et al. [104] are two state-of-the-art offline optimal placement analysis results, with both papers focusing on variable object sizes caching problems. Berger et al. provide a method to calculate offline optimal bounds *FOO* as well as a practical approximation for such bounds *PFOO* for real world storage and CDN workloads through rounding. Li et al. [104] proposed an offline optimal caching algorithm *OSL* which statistically predicts object lifetime with histories and assigns leases for cached objects.

Sampling. As we discussed above, a fundamental bottleneck for performance measurement is data scale. This results in a trade-off between log data amount and measurement overhead. A practical solution is to use a subset of large-scale log data through sampling.

Sampling techniques have been proven empirically to be efficient for measuring cache utilities with low overhead. Many recent caching or placement works have deployed spatial sampling [159, 97, 42, 134, 135, 158, 85] or temporal sampling [51, 163] to improve the efficiency. Spatial sampling has been cited as a remarkably robust statistic for constructing miss ratio curves to better reflect the common cache metrics like reuse distances, compared with temporal sampling.

Except caching works, sampling is also well used in online performance measurements. Recall that most tracing systems today do not trace everything; instead, they *sample* traces. Head-based sampling, where the decision about whether to sample a request is made uniformly at random when it first enters, is a key feature of current tracing frameworks in industry [94, 19, 22, 24] with some sampling rates reportedly as low as 0.1% [149]. Tracing with head-based sampling or coordinated bursty sampling [38]— periodically recording detailed traces across multiple layers simultaneously over a brief time interval, otherwise recording nothing— can help diagnose issues that show up frequently enough in captured data. However, they miss information about anomalous behaviors and infrequently exercised code paths that are likely missing from the traces [101, 100, 129, 149, 7].

Delaying the sampling decision has been a hot topic among open-source tracing systems [23, 17, 39, 10, 15, 16, 20], which even saw a limited implementation in Facebook's Canopy [94]. These attempts have been unable to overcome incomplete or partial traces, traces with unexpected shapes, and incorrect results in aggregate analytics [112, 39, 15, 17, 11].

Tail-based sampling is also a claimed feature of LightStep [107], whose proprietary internals, eviction decisions, and performance impacts are unknown. For detailed telemetry at high rates, 100% sampling of traces is quickly infeasible both for the processing overheads and due to limitations on memory and storage. OpenTelemetry [21], the main open-source standard for distributed tracing, still has the scalability of its tail-based sampling listed as an open issue [108].

Chapter 3

Tracing Edge Cases With Hindsight

Problem. Distributed tracing frameworks record detailed, end-to-end traces of requests executing in distributed systems, and are helpful for a wide range of troubleshooting use cases [147, 130]. Diagnosing and troubleshooting performance problems in large scale is famously onerous, exacerbated by growing system complexity, limited engineering resources, and rising end-user expectations. Particularly vexing for application developers are *edge cases*: performance problems that have adverse effects but are too rare or complex to ascribe obvious root causes.

Consider, as running examples, the problems of diagnosing uncommon errors, mitigating tail-latency, and conducting temporal provenance. High tail-latency among end-to-end requests is a well-known bane of interactive performance [65]. Although high latency is easy to detect and immediately experienced by the end user [65], its root causes may be scattered across previously traversed machines and layers galore, with an exponential number of configuration parameters or hardware components that may have contributed to the delay.

Similarly, high-level API exceptions may culminate from prior interactions with other requests, system misconfiguration, hardware failures, transient faults [95], elusive programming mistakes, or even a significantly hard-to-reproduce combination of issues.

Third, a request exhibiting issues because of an overfull queue somewhere in the distributed system is merely the casualty of an earlier problem [167]; troubleshooting requires systematically investigating these other, unrelated requests.

In the ideal setting, when the developers can trace every request flowing through the system, tracing is a powerful tool to diagnose issues, including edge cases. In typical production environments, however, resources are not infinite. Tracing every single request—including generating, collecting, and storing comprehensive telemetry—requires enormous backend infrastructure and storage that is unacceptable to infrastructure operators. To manage overhead, distributed tracing frameworks collect only a small (0.001% [149, 94]), random sample of traces in hopes that problematic requests of interest to application developers will show up in the sampled traces [101, 149, 94, 15]. But edge cases are, by definition, rare, so when an edge-case request is encountered, it might not have been traced. This presents a bind: for most edge cases *there is no trace for the application developer to analyze*, thus application developers don't have enough *useful* traces.

A known shortcoming of today's distributed tracing approaches [130], the root of the predicament is that application developers want as many *useful* traces as possible, while infrastructure owners have to balance resource constraints. Yet crucially, developers are not asking for *more* traces—they are asking for the *right* traces.

Secret Sauce. A few observations lead us to believe this Gordian knot can be cut. First, we can separate out the performance concerns by noting that **data is cheap to generate but expensive thereafter** [170, 88]. Operators can generate as much data as they like — the bottlenecks are in the storage to persist that data for the future, and network costs for centralizing the data. Second, we note that **most data is superfluous**: normal operations are by definition uninteresting for tracking down a specific problem [129, 101]. While curbing needless data can alleviate the storage overhead concerns, determining *which* data to keep remains a challenge. Finally, we pose there is a large class of problems where the **symptoms of a problem can be detected quickly**. For example, the moment a low-latency system observes an outlier request, *e.g.*, one that took seconds to be processed, something has evidently gone wrong and an investigation is warranted. In these cases, the relevant data for troubleshooting does not extend far back in time. Notably, when problems can be detected shortly after they occur, say within 10 seconds, then even **buffering all data for a short interval is within the memory limits of each machine**. Then, at the moment of detection, all pertinent data for the problematic request would thus still exist in memory on the nodes through which the request traversed. This data could then be persisted on or collated from the relevant nodes, thus avoiding network and slow storage overheads for the vast majority of requests.

Solution.[1] We resolve the problem of tracing edge-case requests in production environments. We focus our attention on *symptomatic* edge cases, where the performance effects of the problem manifest shortly after its causes and where the impacts can be observed programmatically—a broad family of problems that includes the three classes of use cases above. We call these *symptomatic edge cases.*

We built Hindsight—an always-on, lightweight distributed tracing system for diagnosing symptomatic edge-cases. Hindsight is compatible with existing tracing APIs—as a practical tool for edge-case analysis,such as occasional requests experiencing high latency, without relying on good fortune. Hindsight offers ***retroactive sampling***, inspired by network provenance [182, 181, 95, 56], to collect telemetry data back in time from the present moment of detection, across all machines that handled the request. Under retroactive sampling, all trace data is recorded locally but only reported when a symptom is detected, allowing applications to generate copious trace data in case they are needed without encumbering the tracing system's backend collection infrastructure. Retroactive sampling ultimately reports the same volume of trace data as other sampling methods, but ensures that edge-case traces are not missed. To provide efficient and coherent retroactive sampling, Hindsight's design carefully separates its dataplane – *e.g.* generating trace data into fast local memory – from control logic – *e.g.* for indexing metadata, coordinating among machines, and triggering collection for symptomatic requests on demand.

As demonstration, we apply Hindsight on three use cases corresponding to our running examples. We run experiments on the DeathStar Microservices Benchmark [76], the

[1]This work revises the paper: *The Benefit of Hindsight: Tracing Edge-Cases in Distributed Systems* that is currently under review.

Hadoop Distributed File System [148], and on several micro- and macro-benchmarks. We have also integrated Hindsight as a replacement collection component for X-Trace [75] and OpenTracing [22]. Our experimental results show that Hindsight imposes nanosecond-level overhead when generating trace data, can scale to GB/s of data per node, rapidly reconstructs traces when triggered, and effectively captures problematic traces, as well as related lateral traces, within tens of milliseconds of identifying a symptom.

In summary, this chapter makes the following contributions.

- We describe the retroactive sampling abstraction for capturing traces of symptomatic edge-cases.
- We present the design of Hindsight, a distributed tracing system that implements retroactive sampling. Hindsight is compatible with existing tracing APIs and can be transparently integrated with existing applications.
- We apply Hindsight on real-world use cases and show that efficiently collecting edge-case requests is practical.
- We evaluate Hindsight on multiple benchmarks and real systems, showing that it can achieve nanosecond-level overhead on trace data generation and handle GB/s data per node while collecting coherent traces.
- We illustrate that Hindsight is compatible and performs better than state-of-the-art tracing systems (X-Trace and Jaeger) with more efficient trace-data generation and lower overhead, while providing edge-case tracing.

3.1 Motivation

Distributed tracing systems are in widespread use in both open-source [21, 24, 19] and major internet companies [149, 94, 129]. These tools record traces of *end-to-end requests*: a trace contains logs, metrics, and other event data, along with timing and ordering, generated from every machine visited by the request. End-to-end request traces have proved to be especially useful for troubleshooting distributed systems since they explicitly tie together the individual slices of work performed across different machines, enabling an operator to observe how the work done by one machine influences, and is influenced by, work done on others [75, 149, 129]. Prior research has demonstrated a range of troubleshooting tasks using request traces, including common-case analyses centered on aggregate system behavior, distributions over data, and relationships between system components [141, 130, 147]; and edge-case analyses such as error diagnosis [111, 172, 173] and tail-latency troubleshooting [65, 103, 178, 151, 122]. In this chapter, we consider general edge-case analysis use cases, so we begin with some examples.

The picture is different, however, for the other major use case of distributed tracing: *edge-case analysis* [10, 15, 16, 20, 39, 11, 108, 23, 17] , which relies on capturing rare, outlier, and anomalous end-to-end requests [167, 82, 57, 137]. Rather, with maturing open-source implementations [19, 24], standardization of key tracing concepts [22, 21], and widespread adoption of distributed tracing in practice [130, 147], the gap between existing tracing systems and the needs of edge-case analysis is becoming more apparent [39, 10, 15, 23, 17]. We consider the following cases as examples.

Error diagnosis (UC1). Hardware failures, component errors, exceptions, and programming mistakes abound in large distributed systems. Despite advances in testing [36] and verification [79, 102, 83], new or complex problems slip through the cracks and may wreak havoc if not promptly addressed. Application developers thus often play the role of detective, to identify root causes of errors. Yet without a trace of a problematic request, the developer will face a daunting task: each request traverses many different processes and machines and its outcome is influenced by every machine it visits. The symptoms of a problem often manifest far from the root causes [115, 72, 111], and the potential root causes are manifold, perhaps a combination of software or hardware problems on multiple nodes or network links [95].

Tail-latency troubleshooting (UC2). Distributed systems track a wide range of high-level health metrics, such as API distributions, latency percentiles, resource utilization, and many others [94, 93]. An operator may observe an unusual metric jump, say the 99.9^{th} percentile latency has spiked for some important API. However, knowing about the spike is not enough; the application developer wants to identify the specific service, code paths, or conditions that contribute to the peak, so that any underlying problems can be addressed [103, 65, 122].

Temporal provenance (UC3). Many modern distributed systems respond to requests through an architecture of loosely coupled microservices [147]. Application developers need tools for tracking queuing issues [6, 13, 8, 5, 9] when the number of components in a distributed system is large, since a request r exhibiting symptoms (*e.g.* prolonged queueing time) may not be the true culprit for the backlogged queue. Rather, the developer wants to follow the *temporal provenance* of r to determine *lateral traces* of other related requests with which r interacted through shared components and queues [167].

3.1.1 Trace Collection Infrastructure

Distributed tracing frameworks require a supporting backend *trace collection infrastructure* that is distinct from the traced applications. Applications continually generate trace data using a client library, which internally serializes and transmits the data over the network. The trace collection infrastructure, or *backend* for short, receives trace data, and thereafter processes it in various way to combine data from different machines, construct trace objects, apply user-defined functions, and ultimately store the trace in a database [94].

Production applications cannot trace *every* request, as doing so would generate far too much data and impose an enormous burden on the backend [108, 149, 94]. In production systems capturing a trace of *every* request to an application would impose enormous burden on the backend trace collection infrastructure. At Facebook, for example, several MBs of tracing data are generated per traced request [94]; at Google, traces are typically more detailed than debug-level logging [149]. Thus a key design concern for current tracing frameworks is to reduce the volume of data reported to the backend.

Head-based sampling. The de facto approach of all existing distributed tracing tools is to simply capture fewer traces. Instead of tracing every request, the application will only record traces for a small number of requests, chosen by *random sampling* in the client library [147]. Sampling decisions occur at the *beginning* of a request when it enters the system and before it starts executing. Trace data is only recorded if the sampling decision is successful. Head-based sampling satisfies an all-or-nothing property: if a request is sampled, then applications will record an *entire* trace of the request, including all data it produces across all machines it visits. Otherwise, the application records nothing about the request on *any* machine. Coherent traces are essential for distributed tracing. A partial or fragmented trace has limited value in diagnosis [75, 149] because it sacrifices the end-to-end visibility that makes the trace useful in the first place [129].

Overall, head-based sampling reduces the volume of trace data sent to the trace collection infrastructure, with the sampling probability determining how much trace data is collected. In production, the sampling probability is typically very low: Jaeger's default is 0.1% [90]; some systems sample as few as 0.001% [149, 94].

3.1.2 Sampling Derails Edge-Case Analysis

Head-based sampling is indiscriminate: the fate of a trace is determined a priori, upon arrival to the system. With low sampling probabilities, the vast majority of requests are not traced. For edge-case analysis especially, requests of interest are inherently rare and thus unlikely to be traced at all.

The outcome for the application developer investigating edge cases is thus disappointing: problems arise, but traces do not exist. For example, the developer may have reports that errors took place (UC1) yet the corresponding 'rare' requests were not sampled when those requests began. The developer therefore lacks the detailed cross-machine data necessary for finding the error's root cause. Likewise, the application's high-level metric monitoring may indicate a spike in end-to-end tail-latency (UC2); a developer is thus aware that these high-latency outliers exist, yet without a trace, the developer lacks the ability to localize the problem to a particular component or request class. The situation is even more problematic when investigating bottlenecked queues via temporal provenance (UC3): the tracing system will have only a vanishing probability that traces of *all* relevant requests in the queue were captured, since each request was sampled independently.

Practitioners have long pointed out a discord between what traces are interesting and what traces gets sampled [39, 10, 15, 23, 17]. ***Tail-based sampling*** describes how the backend trace collection infrastructure could identify 'interesting' traces and only persist those to storage [141, 101, 100], yet such approaches elide the basic scalability challenge faced by the trace collection infrastructure in the first place: we cannot collect all traces. To date, the desire for a low-overhead sampling system that would support edge-case analysis remains unfulfilled.

Tail-based sampling is a proposed solution that falls short in practice; it entails tracing *all* requests, then discarding irrelevant traces within the backend trace collection infrastructure. Since collecting all traces in production environments with high rates of detailed telemetry is infeasible due to processing overheads on back-end servers [108], current tail-based sampling systems impose *additional* head-based sampling [141], thus inheriting its shortcomings.

3.2 Challenges

The crucible of edge-case analysis under prevailing sampling approaches is that developers miss out on *relevant* traces. Our goal is to remedy this situation: to capture all relevant request traces without relying on serendipity or imposing additional overhead on the tracing backend. We first address a series of motivating questions.

What traces are relevant? Typically, strong signals of an interesting request come towards the end of its execution when more information is known about its fate [15, 8, 141, 23, 17, 39]. Moreover, looking to our motivating use cases, we observe that many issues manifest a known set of symptoms that are cheaply detectable soon after they occur, like error codes (UC1), high end-to-end response time (UC2), or an overfull queue (UC3). In each scenario, the symptoms can be detected programmatically even though the underlying causes remain unknown. We call such issues *symptomatic edge-cases.*

Why not just collect traces after we see the symptoms? Paradoxically, since the request has already executed, enabling tracing at a late point of the request means we have already missed the events that led to the anomaly. The sure-fire way of obtaining coherent traces for any edge-case is to trace from the very beginning of the request.

Why don't tracing frameworks use ring-buffers? Machine-level logging and teleme-

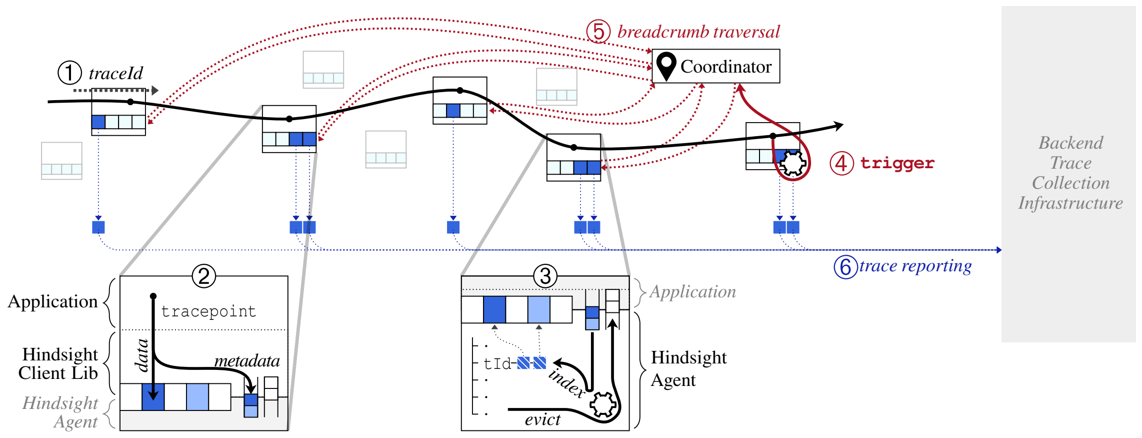


Figure 3.1: **The end-to-end lifecycle of a trace in Hindsight** (subsection 3.4.1). A request (①, solid black line) traverses the processes of the system, depositing trace data data using Hindsight's client library (②). A Hindsight agent on each machine locally indexes data (③). If the application detects outlier symptoms, it can trigger trace collection (④). Hindsight's coordinator traverses breadcrumbs left behind on each machine (⑤), instructing each agent to set aside and report this request's trace data. Subsequently, agents will report the triggered trace data to the existing backend trace collection infrastructure (⑥).

try systems often make use of in-memory ring-buffers for temporarily persisting telemetry in case it is needed in the near-future [179, 64, 56]. While conceptually appealing, the mismatch in data granularity makes them a poor fit for distributed tracing. Machines execute many requests concurrently and the generated trace data on any given machine is interleaved with that of other requests, making it difficult to extract any one request's specific slice of data. Conversely, each request executes across many machines, so trace data must be identified and extracted from all relevant machines.

3.3 Retroactive Sampling

Building on these answers, we now detail our approach to the problem, using an abstraction we call ***retroactive sampling***. Whereas traditional distributed tracing systems based on sampling obtain edge-case traces only by luck, retroactive sampling enables applications to obtain them explicitly. Inspired by work on network provenance [95, 182, 181, 55, 56], retroactive sampling endows the application with the benefit of hindsight: when the application detects something wrong or anomalous about a request, the tracing system can reach back into the recent past to retrieve its trace data as follows.

Nodes generate, but do not collect, all trace data. Our tracing system is not clairvoyant: it lacks information about whether a request will be impacted by some sort of problem (and thereby be interesting) until after the symptoms manifest themselves [23, 17, 39, 15]. As argued, the only conclusion is to record *all* trace data. Recording the data, however, need not burden the backend infrastructure until the traces are reported, giving applications leeway to generate fine-detail trace data locally in case it will be beneficial after a problem occurs.

A trigger signals that a trace is relevant. Under retroactive sampling, a tracing system programmatically signals after-the-fact that a request was indeed relevant. We call this a ***trigger*** that may be fired at any point during or even after a request is served, since symptoms (*e.g.* high request latency) may only be measurable after the fact.

Traces are collected across machines. So far, traces are only recorded locally into memory without any forwarding or coordination among the nodes—we deliberately refrain from doing more complex processing for every trace. Thus for each individual request, its trace data will be dispersed across memory of the multiple machines that it visited. Yet when a machine fires a trigger to collect a trace, it is effectively requesting *all* corresponding trace data from *all* machines visited by the request. In response to the trigger mechanism, systems implementing retroactive sampling must therefore notify pertinent machines that they must *report* the relevant trace data to the backend.

Trace data expires. The tracing system deposits trace data in fast local memory on the machine where the data is generated. Lazy collection via triggers implies that, eventually, we will exhaust the memory available to record traces; naturally, we should overwrite old trace data to make way for traces of new requests. We call the implicit time duration between generating data and overwriting it the ***event horizon***. After the event horizon, the trace data has been overwritten and is gone forever. With large and detailed traces, or highly constrained memory, the event horizon will be short—tens of seconds.

Triggers are automatic. To guarantee trace collection of any traces of interest within the event horizon, the developer needs *automatic* triggers. Symptomatic edge cases, by definition, can be detected programmatically within a short interval of their root causes. An automatic trigger for tail-latency (UC2), for example, could simply check for a tardy response.

3.4 Design

This section introduces the design details of Hindsight.

3.4.1 Overview

Hindsight is a distributed tracing framework that implements retroactive sampling. We begin with a high-level overview of Hindsight's main components before describing our key design choices. Figure 3.1 depicts the end-to-end flow of a request executing in a distributed system, interacting with Hindsight's tracing API along the way.

① Upon request arrival, Hindsight generates a random unique **traceId** for this request and thereafter propagates it alongside the request to any machine visited [115].

② While the request executes, it generates trace data (*e.g.* logs, tracepoints, spans) that the application reports using Hindsight's **tracepoint** client API. When the request completes, its trace data is left scattered across the machines visited.

③ Internally, a Hindsight ***agent*** running on each machine receives and manages trace data. Hindsight agents do not eagerly report trace data to the backend collection infrastructure – instead, agents index and organize metadata in-memory and await an explicit trigger. For most traces, there is no trigger, the trace is not reported, and the data residing on the local machine is overwritten by new data.

④ The application may detect an outlier symptom (*e.g.* an erroneous response value, high latency, or a bottlenecked queue) during the request and thus want to persist its trace. The application invokes Hindsight's trigger API on the machine where the symptom is detected, passing the **traceId**.

⑤ The local agent immediately initiates trace collection by sending **traceId** to Hindsight's logically centralized *coordinator* service. The coordinator's role is to recursively contact all machines that the request visited by following *breadcrumbs* deposited by the request at each machine it visited; a breadcrumb is a pointer to another machine involved in the request (*e.g.* to the RPC caller or callee).

⑥ When instructed by the coordinator, an agent will set aside relevant data for this **traceId** and report it to the backend.

3.4.2 API Compatibility

Hindsight centers on redesigning the *internals* of a distributed tracing system to support retroactive sampling while remaining compatible with other use cases, such as tracing common-case requests that establish aggregate system behavior. Hindsight's design is encompassing: rare triggers for edge-case requests can intermingle with frequent triggers for baseline common-case requests. Moreover, Hindsight does not redesign or replace the backend trace collection components.

Hindsight is compatible with existing distributed tracing APIs and can be transparently integrated into an application's existing tracing instrumentation (*e.g.* OpenTracing [22]). In our evaluation (cf. section 3.6), we integrated Hindsight as the collection component for both X-Trace [75] and OpenTracing [22]. Existing tracing instrumentation API calls relay directly to Hindsight. For example, API calls that create and annotate spans are proxied as **tracepoint** calls. Hindsight seamlessly supports any existing head-based sampling policy: trigger is invoked immediately for a positive sampling decision or if the `sampled` flag is set within a received OpenTracing context. Lastly, Hindsight piggybacks its breadcrumbs within OpenTracing's context propagation; transparently enabling Hindsight's trace coordination procedure.

For a developer wanting to benefit from retroactive sampling, the only additional concern is when to invoke trigger. For example, this may entail adding a trigger call within a service's exception handler, or after checking for outlier latency upon a request's completion. A developer can explicitly decide the conditions for triggering a request, but Hindsight also provides a library of automatic triggers based on metric percentiles, categorical features, and exceptions. All our use cases (UC1–UC3) can be implemented using Hindsight's autotriggers (cf. subsection 3.5.2).

3.4.3 Data Coherence

A trace is only useful if it has coherent data: the trace must contain *all* information about the request across *all* machines visited. If a trace is patchy (*i.e.* missing data from one or more machines) then the trace's value as a whole is compromised.

Hindsight encounters potential trace incoherence in several places: at ③ when an agent chooses which old data to evict; at ⑤ if the coordinator is too slow to contact all machines that handled a request; and at ⑥ if more traces have been triggered than can be reported. The threat is exacerbated because it only takes one agent dropping its slice of a trace to render the remaining data on other agents effectively worthless.

We ensure coherence as follows. Within a single agent (③), a trace is an atomic entity; when an agent evicts a trace, the trace's data are evicted in its entirety. Agents carefully organize and index metadata to account for a trace being non-contiguous and fragmented in memory, and enable efficient insertion and eviction. After a trace is triggered (⑤), the coordinator service rapidly disseminates the trigger, recursively following breadcrumbs, to ensure no agent inadvertently evicts its slice of the trace. Breadcrumbs are scalable, as the coordinator only needs to contact services explicitly known to be part of a request's execution, and trivially scales by sharding on the triggered **traceId**. Triggers are eagerly disseminated; subsequently agents report the actual trace data to the backend collection infrastructure asynchronously in the background. Hindsight enforces a user-configured rate-limit on background trace reporting.

Unlike head-based sampling that traces a fixed percentage of requests, retroactive sampling may fire triggers arbitrarily often. A trigger-happy application may cause a backlog of unreported traces within the agent (⑥). If the agent passes a threshold of triggered data, it will begin to drop data for triggered traces. A backlog of data on one agent strongly implies a backlog on others, thus agents making different choices about which data to drop risk trace incoherence. Hindsight uses consistent hashing of **TraceIds** in several places to decide priority both for reporting and evicting trace data.

Finally, a developer might invoke trigger at several different places in their application, for a diverse range of symptoms; developers can distinguish different triggers by providing a **triggerId**. Hindsight can prevent a profuse trigger from impacting trace collection of other, low-frequency triggers: agents implement weighted fair sharing over triggers for reporting and evicting trace data, and users can configure weights and rate-limits for each **triggerId**.

3.4.4 Efficient Data Management

From the perspective of the backend trace collection infrastructure, an application using Hindsight will report a similar volume of traces and have similar demands to conventional head-based sampling. For an individual machine (③), however, retroactive sampling entails the application generate a substantially higher volume of trace data into local memory. For perspective, traces in production systems can be verbose, containing tens of thousands of events [94], and often incorporating detailed debug information; *e.g.* our Hadoop experiments, (cf. subsection 3.6.1), sustained 30 MB/s per process. Even when **tracepoint**s involve string formatting and argument packing, prior work on efficient logging has attained lower bounds of a few nanoseconds per logging point [170, 171]. More generally, Hindsight's **tracepoint** API can be a sink for arbitrary telemetry sources such as function tracing or new technology like Intel® Processor Trace (PT) [88, Ch. 36], which can reach 100–200 MB/s of data per core at 5–15% runtime overhead. Handling this data at large volumes requires a design centered on performance.

The most sensitive performance bottleneck for Hindsight occurs between the client application generating data via **tracepoint**, and the local Hindsight agent that manages trace data. Agents must efficiently receive new trace data for *every* request, evict old data to be overwritten, and set aside triggered data for reporting, while managing trace data coherently with respect to other agents. Simple approaches fail to meet these needs; ring-buffers, *e.g.* are effective across many systems designs but deteriorate in our setting by incoherently overwriting trace data and posing significant costs to extract triggered data via expensive scanning. Instead, our design establishes a clear split between *control* and *data* activities, which congregates general-purpose data and efficiency in the data plane, and embeds all logic in the control plane.

Hindsight's data plane is concerned with efficiently writing trace data from client applications. Hindsight writes trace data to a large shared memory pool, divided up into *buffers*. A client invoking **tracepoint** writes to a buffer, and each buffer only belongs to one trace at a time. Full buffers are continually circulated to the agent via shared memory queues; the agent likewise continually frees up old buffers for reuse.

Hindsight's agent process encapsulates control activities: it receives buffer metadata, indexes them according to **traceId**, and coherently evicts old buffers. Agents receive triggers and communicate with Hindsight's coordinator, manage breadcrumbs linking the trace data that is strewn across many agents, extract triggered trace data, and report data asynchronously to the backend trace collection infrastructure. Hindsight's control and data distinction yields an efficient agent implementation, and confers additional desirable properties, such as making it easy to change and update control logic like eviction and fair sharing policies.

In our description thus far, we assume that an application will trace *all* requests into local memory. Such completeness is not a strict requirement for Hindsight; an application may opt to trace a smaller fraction of requests (*e.g.* 10%), for instance, if generating data is expensive or unoptimized, or if the application is highly performance-sensitive. Hindsight remains effective in such a scenario since we still capture significantly more traces to local memory than can be reported to tracing backends (*e.g.* 0.1% and lower). The scale-back is supported via a *trace percentage* option (default 100%). Hindsight enforces scale-back coherently across agents through consistent **traceId** hashing (cf. subsection 3.4.3), and will maintain breadcrumbs and support triggers for all requests, traced or not.

3.4.5 Divorcing Triggers from Traces

Applications initiate retroactive sampling via Hindsight's trigger API (⑤). Triggers are orthogonal to traces in Hindsight for several reasons.

Efficiency. In principle, triggers could be calculated directly atop trace data. For example, end-to-end latency can be calculated directly from span start and end times recorded in a trace. Yet applying user-defined functions in-dataplane poses an infeasible performance challenge. Moreover, traces are brittle data structures: in practice, as engineers continually modify and update instrumentation, the traces may sometimes 'break' [27, 26, 29, 31, 30, 32, 25, 112], leading to incorrect or degenerate derivative features and metrics [94, 113, 149].

Integration. In the common case, symptoms are easy to detect and localize: top-level error codes; high latency; increased queue time. Such symptoms can be readily recognized and cheaply computed without the trigger mechanism needing the trace data itself. It further leads to a straightforward integration of triggers into existing metric monitoring systems regardless of their architecture.

Lateral traces. Outlier behavior may not map directly to a single request; instead there may be several other related *lateral* requests. For example, to diagnose a bottlenecked queue (UC3), a trigger needs to capture traces for the previous N requests to understand what led to queue buildup [167]; to diagnose a write-ahead log, we desire all requests blocking on a log sync [6, 13]; to diagnose resource contention we require all requests contending for a slow disk or network [8, 5, 9]. Hindsight enables an application to atomically trigger a group of related lateral **traceIds**; internally Hindsight will ensure that the group as a whole is coherently collected.

3.5 Implementation

We have implemented Hindsight in ≈4KLOC in C for the client library, ≈3KLOC in Go for the agent and coordinator, and ≈300LOC for a JNI-based Java client library for integrating with Hadoop in our experiments (cf. section 3.6). We chose C for dataplane efficiency and Go for its ease of use for the more complex control plane logic. In this section, we elaborate how we meet the Hindsight design goals (cf. section 3.4).

3.5.1 Agent Data Management

Hindsight pre-allocates a fixed-size *buffer pool* in shared memory for storing trace data. We choose a fixed-size pool to bound memory overheads, a desirable property for telemetry

begin(traceId)	Record that request with id **traceId** is processing in the current thread.
tracepoint({payload})	Record data for the current trace; **payload** is arbitrary size.
breadcrumb(address)	Adds a breadcrumb to the current trace on the local node pointing to some other node **address**.
serialize()	Obtain the current **traceId** and a **breadcrumb** to the current node.
end()	Request finished processing in current thread; flush and remove buffers.
trigger(traceId,triggerId, lateralTraceIds...)	Tell Hindsight to trigger **traceId** for collection; **triggerId** is an arbitrary identifier used for rate-limiting.

Table 3.1: **The Hindsight client API** can be called by applications directly, or by other distributed tracing tools (*e.g.* X-Trace [75], OpenTracing [22]) as the trace collection component (cf. section 3.6).

systems [157]. Hindsight subdivides the buffer pool into fixed-size buffers, by default 32 kB. Client processes write trace data to buffers via Hindsight's client API. The agent process does not touch data in the buffer pool except when reporting triggered traces. At each point in time, a buffer can only contain trace data of a single request; no two different requests will write trace data to the same buffer at the same time. A single trace will typically comprise (1) multiple non-contiguous buffers and (2) many buffers scattered across numerous machines. Buffers are the granularity of data management within Hindsight. Within clients and agents, a buffer is addressable by its **bufferId**—its offset into the buffer pool.

3.5.2 Client Library

Writing trace data. Table 3.1 outlines Hindsight's client API. When a request begins executing in a thread, it must call **begin**; subsequently it may call **tracepoint** an arbitrary number of times; and finally when it completes executing in a thread, it must call **end**. As noted in subsection 3.4.2, this aligns with existing tracing instrumentation calls. Hindsight internally maintains thread-local state including the current **traceId** and a pointer to a buffer. Calls to **tracepoint** write directly to the thread-local buffer without needing any synchronization. Synchronization is only required when acquiring a new buffer or returning a buffer to the agent; these operations touch shared-memory queues but are infrequent. A buffer must be acquired during **begin**, returned during **end**, and is otherwise only acquired or flushed when it becomes full (*i.e.* from successive **tracepoint** calls).

Communicating with agents. To acquire a buffer, the client library polls a shared-memory *available queue*. The queue returns an integer **bufferId** that points into the buffer pool. Clients do not block if the available queue is empty—clients immediately return, and instead of writing to the buffer pool, write to a 'null buffer', which is discarded afterwards. When the client fills a buffer, it writes its **traceId** and the **bufferId** to a shared-memory *complete queue*. The agent continually drains the complete queue, and likewise continually returns fresh buffers to the available queue. Shared memory queues are lock-free and support batch operations; using batch operations, agents are robust to queue contention from multiple client writer threads.

This paired channel design forms a natural separator between control and data with two desirable properties: (1) queues only communicate metadata—they avoid data copying and use a single integer **bufferId** to represent, by default, a 32 kB buffer; (2) communication is infrequent, occurring only when buffers are filled or a thread switches over to execute a different request, thereby minimizing synchronization. From the client library's perspective, it cheaply and blindly writes trace data into shared memory and forwards only the control metadata to agents; conversely agents are agnostic to buffer contents—they do not inspect data in the shared memory pool and use only the metadata communicated via the complete queue.

Depositing breadcrumbs. Hindsight clients deposit *breadcrumbs* on every machine visited by a request. Each breadcrumb addresses another Hindsight agent that handled the request. When a request arrives at a node, it is carrying the breadcrumb of the previous node. Hindsight's **breadcrumb** API is called during trace context deserialiation, which establishes within the local agent a pointer to the previous node visited for this **traceId**. Similarly, during trace context serialization prior to communication with another node or an RPC response, Hindsight will insert the current node's breadcrumb transparently as a key-value baggage field of the trace context [22, 113]. For synchronous RPCs, breadcrumbs are sufficient for reconstructing full traces triggered by any node, including requests with

PercentileTrigger(p)	Clients call **addSample(traceID, measurement)**. Trigger fires for measurements above percentile p. (*e.g.* high latency or resource consumption)
CategoryTrigger(f)	Clients call **addSample(traceID, label)**. Trigger on rare categorical data that is less frequent than threshold f (*e.g.* rare API calls or odd attributes)
ExceptionTrigger	Trigger on an exception or error code
TriggerSet(T,N)	Trigger all lateral traces in a sliding window of length N when trigger T fires (*e.g.* gather the last N traceIDs in a bottlenecked queue)

Table 3.2: **The Hindsight AutoTrigger API.**

arbitrary concurrency and fan-out. To handle asynchronous settings, clients can also use the **breadcrumb** API to eagerly establish *forward-breadcrumbs* to a named destination node prior to communication. Internally, the **breadcrumb** API call reports the **traceId** and breadcrumb to a shared memory *breadcrumb queue*. Agents poll this queue and index breadcrumbs alongside the buffer metadata. However, agents do not forward or act upon breadcrumbs until a trace is explicitly collected with a trigger.

Triggering trace collection. Applications initiate trace collection by invoking trigger, passing a **traceId** and a **triggerId**; internally this writes to a shared-memory *trigger queue*. The **trigger** API allows any condition that can be programmatically detected to initiate trace collection in Hindsight—*e.g.* a trigger detecting latency outliers; a trigger for erroneous return values; or a trigger for high queue latency. A trigger can specify **traceIds** for a group of related lateral traces (cf. subsection 3.4.5). A developer can implement custom outlier detection and invoke trigger directly, or they can make use of Hindsight's autotrigger library, a separate collection of triggers that track simple conditions over time and automatically invoke trigger when a condition is met in Table 3.2. **TriggerSet** is noteworthy as a building block for lateral tracing; it captures recent traces related to a request that exhibited symptoms.

3.5.3 Agent

Agents maintain metadata for each trace in a map keyed by **traceId**. The metadata for each **traceId** includes a list of **bufferIds** and a list of breadcrumbs. Agents also maintain metadata for each trigger that has fired, including the **traceId**, **triggerId**, and zero or more lateral **traceIds**.

Indexing and reusing buffers. Agents poll the complete queue, each time reading the **traceId** and **bufferId** of a full buffer, then adding this **bufferId** to the trace's metadata. The agent then updates an LRU of **traceIds**. The agent performs eviction when 80% or more of the buffer pool is indexed, by removing the least recently used **traceId** and returning all its **bufferIds** to the available queue. Agents never touch data within buffers; all management is done using **bufferIds**.

Local triggers. Agents poll the trigger queue, each time draining the trigger metadata. The agent immediately forwards the trigger metadata and breadcrumbs to the coordinator, which begins disseminating the trigger to other related agents by recursively following breadcrumbs. Meanwhile the agent schedules the trigger for collection. In the case of a spammy trigger, an agent may decide to immediately discard the trigger instead of forwarding and scheduling it—this is implemented using a per-**triggerId** token bucket. Doing so prevents flooding other agents with triggers.

Remote triggers. Agents receive remote triggers fired by other machines via the coordinator. To facilitate rapid trigger dissemination, the agent immediately responds with any breadcrumbs it has accumulated for the specified **traceId**. Agents do not rate-limit remote triggers (unlike local triggers) since it risks incoherent traces. Instead, all remote triggers are scheduled for collection.

Reporting traces. When a trigger is scheduled for collection, its **traceId** and lateral **traceIds** are removed from the agent's LRU and can no longer be evicted by the regular buffer eviction cycle. The trigger metadata is then inserted into a per-**triggerId** priority queue. In the normal case when an agent is not overloaded, the queue will be empty. The agent asynchronously pulls trigger metadata from the queues; fetches all buffers for all **traceIds** from the buffer pool; reports the buffer *contents* to the backend collection infrastructure; and finally frees the **bufferIds** by returning them to the available queue. If configured, the agent will apply a rate limit to pace its data reporting to the collection infrastructure. New data for any triggered **traceIds** can continue to arrive via the available queue, possibly rescheduling the trigger if existing data is already reported.

Ignoring triggers during overload. If trace collection becomes overloaded, the priority queues will begin to fill up. The agent tracks the proportion of buffer pool pending for collection, and when this exceeds 40%, the agent will pick a **triggerId** and abandon the lowest-priority trigger from its queue. Abandoning a trigger entails removing it from the priority queue and returning all buffers of all **traceIds** to the available queue. If a **traceId** happens to be included in *multiple* triggers, its buffers are only returned after all triggers are abandoned. The agent provides a fair allocation of buffer pool to each **triggerId** and selects whichever **triggerId** most exceeds its weighted max-min fair share. In addition to memory pressure, a trigger will be automatically abandoned if it has not been reported after a configurable delay (5 minutes by default).

Reporting traces during overload. During overload, the agent continues to report traces as described above for the normal case. The agent implements weighted fair queueing to select the next **triggerId** queue, and will adhere to any configured per-**triggerId** rate limits. From the selected queue, the agent dequeues the highest-priority trigger, and reports its data as described above for the normal case.

Coherence during overload. The per-**triggerId** priority queues uses consistent hashing of the **traceId** to determine priority. This priority is consistent across all agents that may have relevant data and results in a given trace enjoying the same priority across all agents. Consequently, even during overload, agents will consistently report the highest-priority traces and evict the lowest-priority traces. The agent's fair allocation of buffer pool to each **triggerId** and fair sharing of reporting bandwidth means a low-throughput **triggerId** is not impacted by a spammy trigger.

3.6 Results

We now evaluate how effectively Hindsight overcomes the fundamental problem of head-based tracing methods in examples (UC1)–(UC3) and meets the goals of retroactive sampling to provide lightweight and effective request tracing.

Tracing integration. We compare Hindsight to two existing distributed tracing systems, X-Trace [75] and Jaeger [19]. We also integrate Hindsight with OpenTracing [22] and X-Trace [75], and, in our experiments, we evaluate Hindsight as a replacement backend to X-Trace [75].

Systems. We have integrated Hindsight with two distributed systems: the Hadoop Distributed File System (HDFS) [148], the DeathStar Microservices Benchmark (DSB) [76]. We also benchmark Hindsight's single-node characteristics and cross-node trace retrieval performance.

Summary. Our experiments demonstrate the following.

- Hindsight effectively provides retroactive sampling and collects relevant edge-case traces across real use cases.
- Hindsight is lightweight and not a bottleneck for client applications. Hindsight can achieve $< 5\,\text{ns}$ **tracepoint** latency and tolerate write throughput up to 55 GB/s. Hindsight's control/data split lets Hindsight's agent match client data generation.
- With large trace data volumes, Hindsight's 'event horizon' extends tens of seconds into the past.
- Hindsight has substantially lower overheads than X-Trace and Jaeger when generating trace data.

3.6.1 Case Studies

We first evaluate Hindsight's tracing libraries and trigger mechanisms on collecting edge-case requests on our motivating use cases.

Error diagnosis (UC1). We deploy Hindsight on DSB, a microservice system with 12 microservices and 17 backends [76]. We add an **ExceptionTrigger** from Hindsight's autotrigger library to the ComposePostService module in DSB's Social Network Benchmark. We run DSB's default workload with 300 req/s. We randomly inject exceptions in the ComposePostService module with error rates ranging from 1% to 10%, varied after each 30 s;

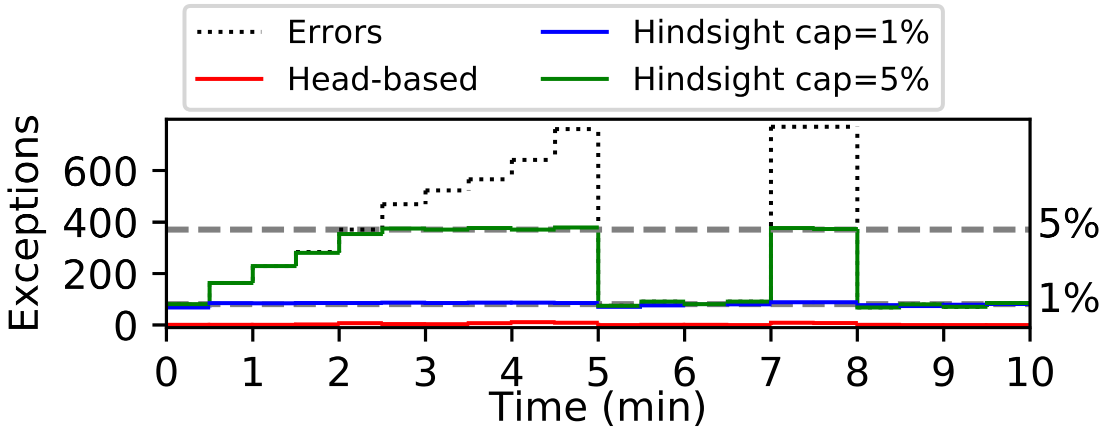


Figure 3.2: **Case Study: Error diagnosis (UC1).** Exceptions captured by different sampling strategies as the error rate varies.

Figure 3.2 shows the collected exceptions for each 30 s time window. We limit Hindsight's agent to trigger on only 1% and 5% exceptions among all requests. Figure 3.2 shows Hindsight is able to collect nearly all exceptions except when they surpass its collection limits. We add a burst of 10% error rates at 7 seconds and Hindsight rapidly meets its collection limit. Unlike Hindsight, a head-based sampling methods (at 1% sampling rate) can only collect 1% of the exceptions at random (see Figure 3.2).

Tail-latency troubleshooting (UC2). We add a **PercentileTrigger** from Hindsight's autotrigger library to the ComposePostService module in the same setting as above, invoking **addSample** at the end of each ComposePost RPC call and providing the measured RPC duration. We set p to 0.99, 0.95, and 0.9, as different thresholds for tail latency. We inject 10% requests at random with 20–30 ms latency. We also compare with head-based sampling methods with 1% sampling rate. Figure 3.3 shows that Hindsight predominantly collects requests near the threshold, whereas head-based sampling methods cannot. The vertical dotted line marks the tail-latency percentile threshold. The figure also shows that Hindsight collected 98.77%–100% of requests above the latency threshold; head-based sampling could gather only $\approx$ 1%.

Temporal provenance (UC3). We add a **QueueTrigger** from Hindsight's autotrigger library to the HDFS NameNode queue — the **QueueTrigger** combines a **TriggerSet** with a **PercentileTrigger**, parameterized to capture $N = 10$ most recently dequeued lateral

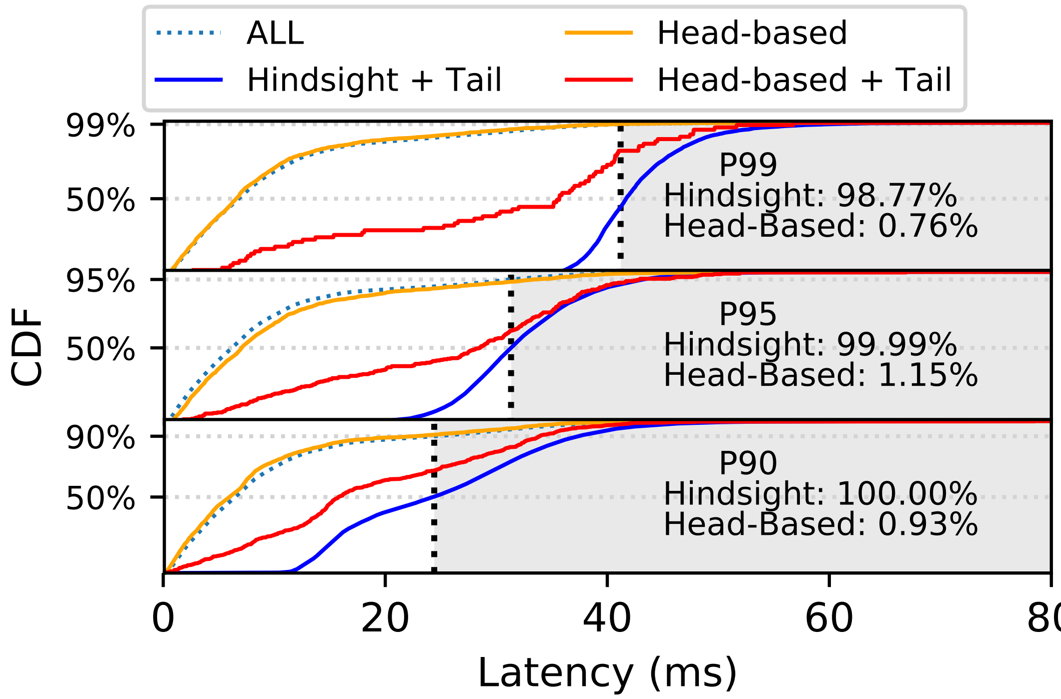


Figure 3.3: **Case Study: Tail-latency troubleshooting (UC2).** Latency of requests captured through different sampling strategies with different tail-latency triggers (top to bottom).

requests when 99.99th percentile queueing latency is observed. We deploy HDFS on 10 machines (8 DataNodes, 1 NameNode, and 1 client) and run a Hindsight agent on each machine. We run a closed-loop workload of random 8 kB reads with 10 concurrent requests.

Figure 3.4 (left) shows NameNode queue latency over time. We inject a burst of 10 expensive **createfile** requests 21 second into the trace that briefly saturate the queue—Figure 3.4 (right) zooms in on this time window. The figure shows high-latency requests (●), requests that fire the autotrigger (**X**), and the additional lateral requests that were triggered to Hindsight (**X**). The first expensive request occurred at 22 seconds, followed by a pause while it was executed. Upon dequeuing the subsequent **read8k** request, **QueueTrigger** fired due to high queue latency, and Hindsight retroactively sampled the 10 prior traces leading up to the trigger. The sample included the culprit expensive request. Overall, all 10 expensive requests were sampled, 8 unrelated requests prior to the first expensive request, and 9 additional **read8k** requests. Moreover, several intermittent latency spikes occurred unrelated to the experiment (Figure 3.4, left), which Hindsight also captured; upon

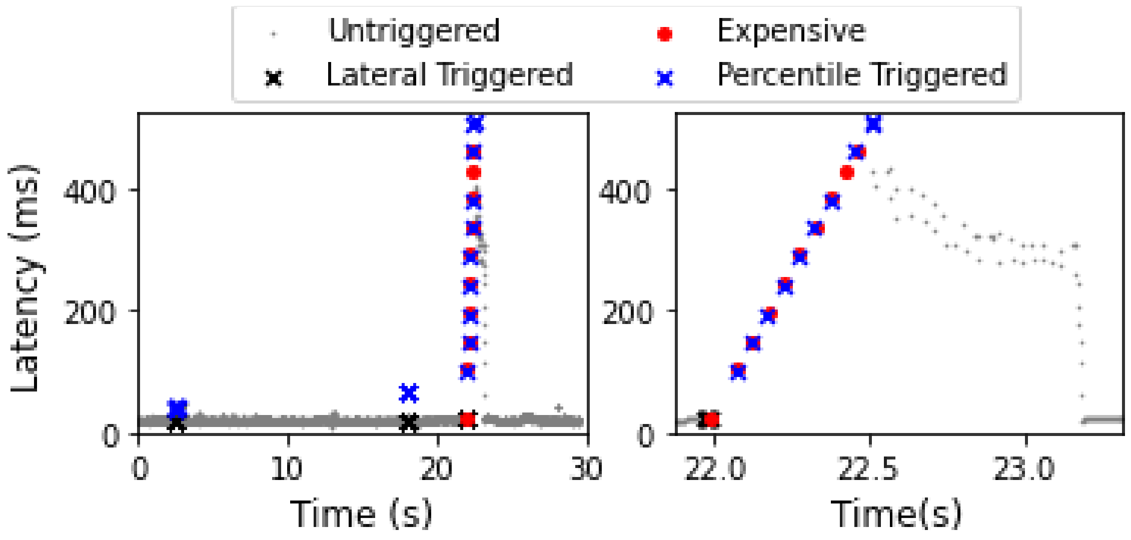


Figure 3.4: **Case Study: Temporal provenance (UC3).** Lateral requests gathered (blue) after triggering on a high-latency request (red) due to an overfull queue in Hadoop's HDFS.

investigation, these were due to garbage collection.

3.6.2 Hindsight Tracing Performance

Hindsight's design emphasizes low-overhead data ingestion. To evaluate this, we consider (a) the latency of client API operations; (b) achievable client data throughput; and (c) agent buffer indexing throughput. We run these experiments on an Intel Xeon W-2245 3.9 GHz workstation with 8 physical cores (16 hyperthreads) and 128 GB RAM. We run a benchmarking program that uses the Hindsight client library along with Hindsight's agent sidecar process.

Client API. In this experiment, we run a client application that executes a loop comprising **begin**, 100 **tracepoint**, and **end** calls. We configure Hindsight with a 1 GB buffer pool divided into 32 kB buffers. Each **tracepoint** writes only a 4-byte payload—this use case is common in statically-preprocessed call site logging (*e.g.* wherever a program logs only static strings [170]). We vary the number of threads from 1 to 32. Figure 3.5 plots latency of **begin**, **end**, and **tracepoint** calls, as we vary the number of threads. For up to 8 threads, **tracepoint** latency ranges from 4.3–4.8 ns/call, benefiting from each thread writing to its own thread-local buffer. Beyond 8 cores, **tracepoint** latency increases lin-

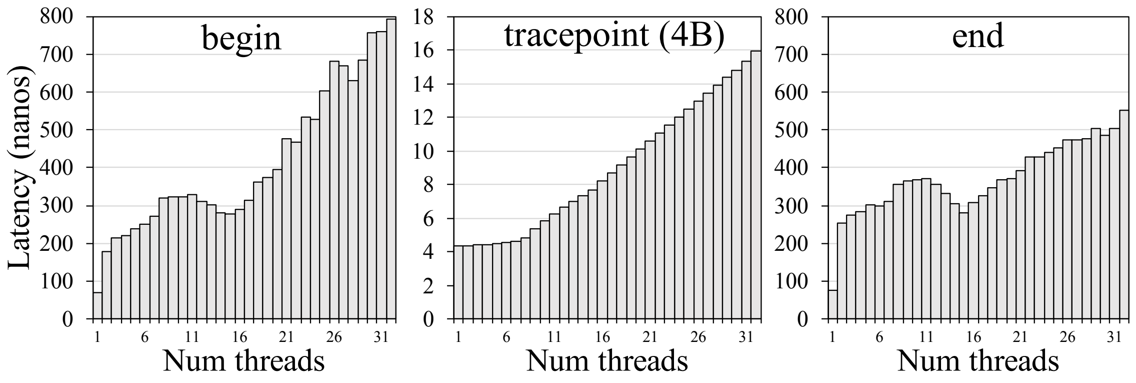


Figure 3.5: **Client API latency.**

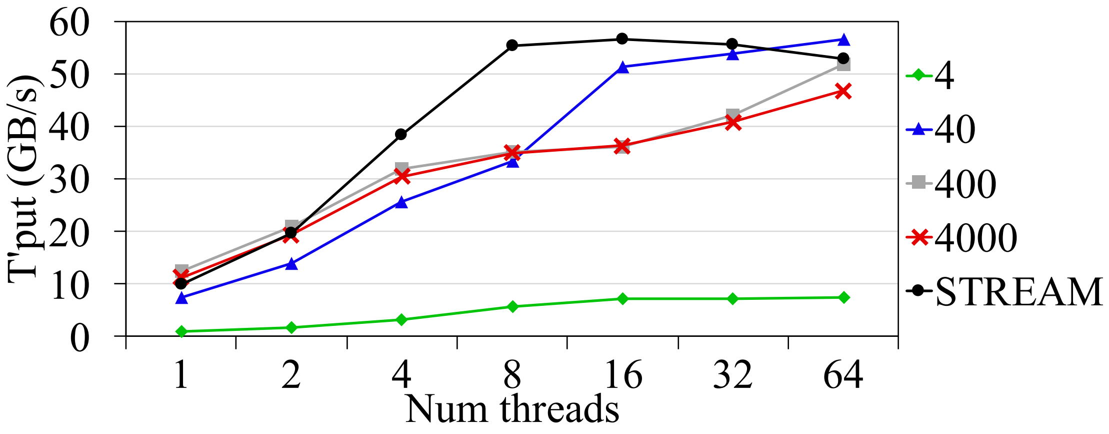


Figure 3.6: **Client Throughput.**

early as we begin to multiplex threads on cores. By comparison, **begin** and **end** are more expensive and variable since they contend for shared-memory queues to acquire and return buffers; fortunately, these concurrent operations are uncommon by design. Latency is between 200–400 ns with at most 8 threads, and much lower for a single thread with no concurrent interference.

Client throughput. We repeat the experiment, keeping a buffer size of 32 kB. We now vary the size of the payload used in **tracepoint** calls, using 4, 40, 400, and 4,000 bytes per call. Figure 3.6 plots the throughput achieved in GB/s. As expected, small payloads of 4 bytes fail to fully saturate memory bandwidth, achieving only 887 MB/s with one thread and peaking at 7.55 GB/s with 64 threads. By contrast, even a modest increase in payload size to 40 bytes is enough to nearly saturate memory bandwidth; with 400 byte payloads, we achieve throughput of 12.5 GB/s on a single core. We include in Figure 3.6 measurements

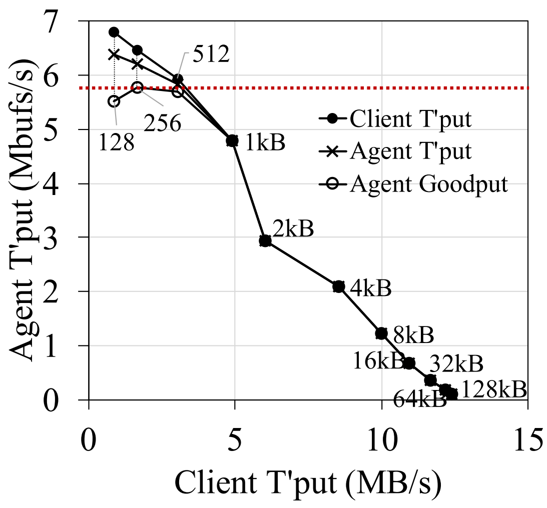


Figure 3.7: **Agent Throughput.**

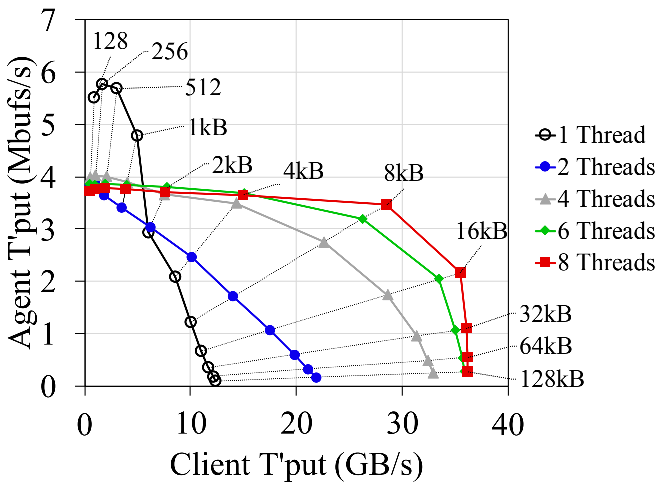


Figure 3.8: **Agent Throughput.**

of peak memory bandwidth from the STREAM benchmark [119]. These throughputs occur because in the common case, **tracepoint** is a memory copy to a thread-local buffer, and thus applications will never be bottlenecked by Hindsight's client library.

Agent throughput. Client data throughput is moot if the agent cannot keep up; recall the agent must continually index buffers and cycle them back to the available queue for clients—if no buffer is available, the client will write to a 'null buffer' and the new data will be lost. This is a last-resort mechanism in Hindsight that fails to respect coherence, and thus must be avoided at all costs.

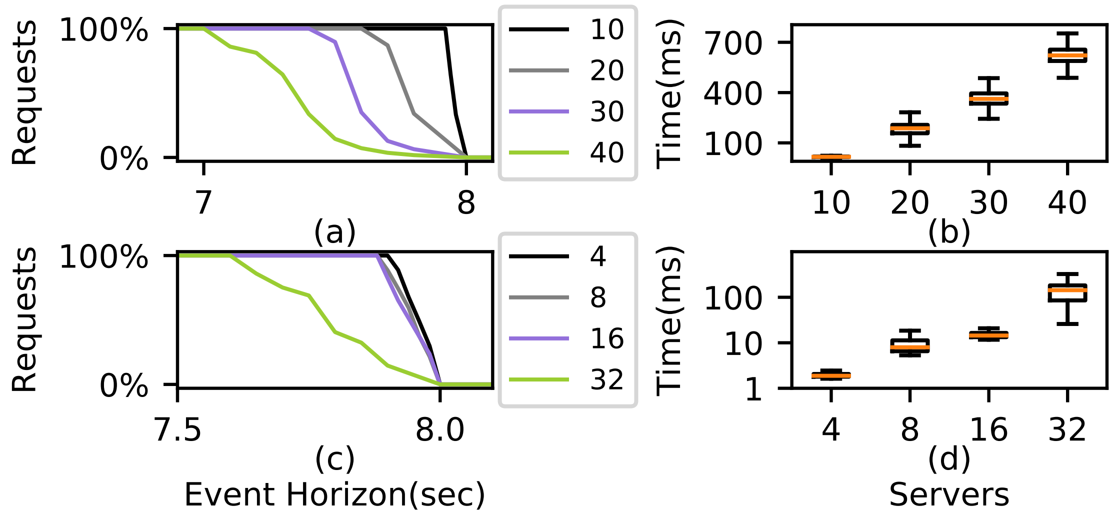


Figure 3.9: Hindsight's **event horizon** and **trace reconstruction** on gRPC-Chain and gRPC-Branch as we vary the number of servers ((a)–(d)).

In this experiment we run one thread, fix the **tracepoint** payload size to 1 kB, and vary the buffer size from 128 B to 128 kB (Hindsight will fragment payloads across multiple buffers when necessary). Figure 3.7 shows the client-side data throughput along agent-side buffer throughput achieved by one thread. The data points are annotated with corresponding buffer size. We see, for example, that large buffer sizes (128 kB) can achieve peak client data throughput (12.1 GB/s) while requiring little of the agent. Conversely, tiny buffer sizes (128 B) stress the agent buffer throughput since we more frequently cycle buffers through the queues. Figure 3.7 plots three lines and indicates two important phenomena. The client throughput line plots the rate at which the client writes buffers, whereas the agent throughput line plots the rate at which the agent cycles buffers; the delta in-between are 'null buffers', written by the client because the available queue is empty, *i.e.* the agent cannot keep up. Writing to null buffers means lost trace data; the third line, agent goodput, counts only the agent-side throughput of buffers that are useful, *i.e.* buffers of traces that did not lose data. We observe that the goodput with 128 B buffers is lower than with 256 B buffers due to greater loss. In general, with $\geq$ 1 kB buffers, the agent is able to consistently keep up without losing data.

Figure 3.8 repeats this experiment with varying numbers of threads, and plots client-

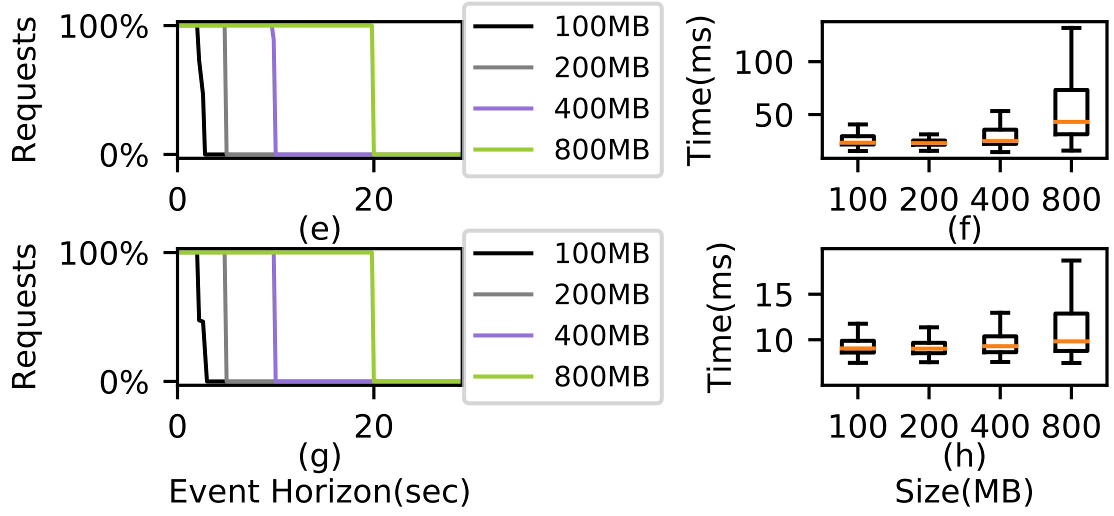


Figure 3.10: Hindsight's **event horizon** and **trace reconstruction** on gRPC-Chain and gRPC-Branch as we vary the buffer pool size ((e)–(h)).

side data throughput and agent-side buffer goodput. Surprisingly, peak buffer throughput is only achievable with one thread; beyond that, client-side contention on the shared queues reduces the achievable client-side throughput. The same effect appears in Figure 3.5 in increased **begin** and **end** latency. We also observe that buffer sizes of 16 kB and higher are sufficient for reaching peak write throughput while remaining comfortably within agent throughput limits; by default, we select 32 kB for Hindsight.

3.6.3 Retroactive Sampling

Hindsight is only useful if trace data is still in-memory on all agents when a trigger fires. Hindsight's *event horizon* describes the time window between a request completing and its data being overwritten: triggers must fire within this window, or else it will be too late and the data will be overwritten.

We evaluate Hindsight's event horizon on two distributed benchmarks: gRPC-Branch is a binary tree of N gRPC servers; each server concurrently invokes its 2 child servers and calls **tracepoint** with a configurable payload. gRPC-Chain is a configurable chain of N gRPC servers that invoke each other in sequence and calls **tracepoint**. We deployed all benchmarks on 12 CloudLab c8220 nodes [66], each with two Intel E5-2660 v2 10-core

CPUs@2.2 GHz, and ran a Hindsight agent per process.

Event horizon. In this set of experiments, we introduce a delay between requests completing and subsequently firing a trigger. We randomly trigger 1% of requests, adding variable delay to the trigger, and measure how many coherent traces are ultimately received by the collection backend.

We vary the number of nodes between 4 and 40 (Figure 3.9 (a)-(d)). We configure a 40 MB buffer pool with 4 kB buffers. The request rate is 1,000 r/s and each request writes 4 kB trace payload per server. We intentionally select a small buffer pool to more readily illustrate the event horizon. Figure 3.9 (a,c) shows the percentage of coherent traces collected for each configured trigger delay. They illustrate how, 8 seconds after a request is complete, the trace data is no longer available for collection. Nevertheless, Hindsight successfully reports 100% of triggered traces after 7 seconds with 40 servers in the gRPC-Chain benchmark, where breadcrumb coordination spans all 40 nodes. Figure 3.9(b,d) shows the trace reporting time in milliseconds—the time between issuing a trigger and data received by the tracing backends. Trace reporting typically takes 100–400 ms, and even with 40 nodes to traverse, completes in less than one second.

Extending the event horizon. Hindsight's event horizon is proportional to the size of the buffer pool configured for agents. To illustrate, we vary the buffer pool size (Figure 3.10 (e)-(h)) from 100 MB to 800 MB. We configure the number of servers as 10 and 8, and use 4 kB buffers. The request rate is 2,000 r/s and each request writes 16 kB trace payload per server. Figure 3.10 (e,g) illustrates how increasing the buffer pool size effectively extends the event horizon possible. Figure 3.10 (f,h) shows that the delay between triggering a trace and receiving its data increases marginally with larger buffer pool sizes.

3.6.4 Comparison with the State-of-the-Art

We finally evaluate how Hindsight compares with existing state-of-the-art tracing systems.

Comparison with X-Trace. To provide more context, we integrate Hindsight with X-Trace [75]; all data generation is done within X-Trace, but instead of reporting data in-band, we reroute it to Hindsight. To a user, this integration is transparent. We run a benchmark

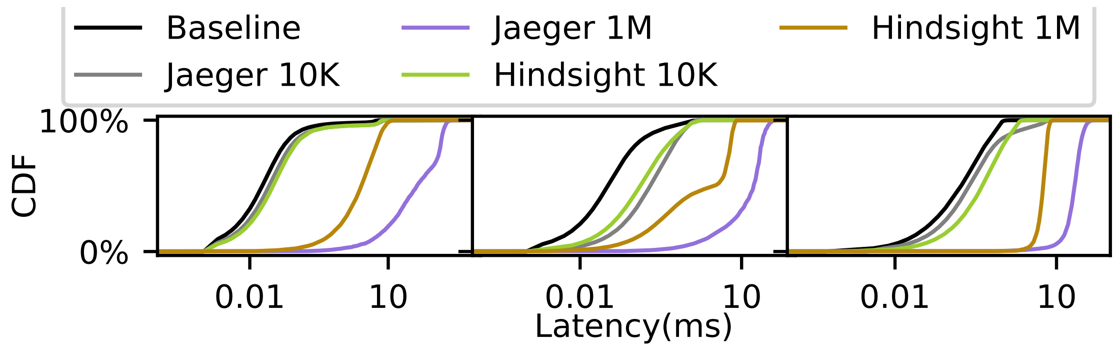


Figure 3.11: **Comparison with Jaeger.** End-to-end latency with three workloads (compose-post, read-home-timeline, read-user-timeline).

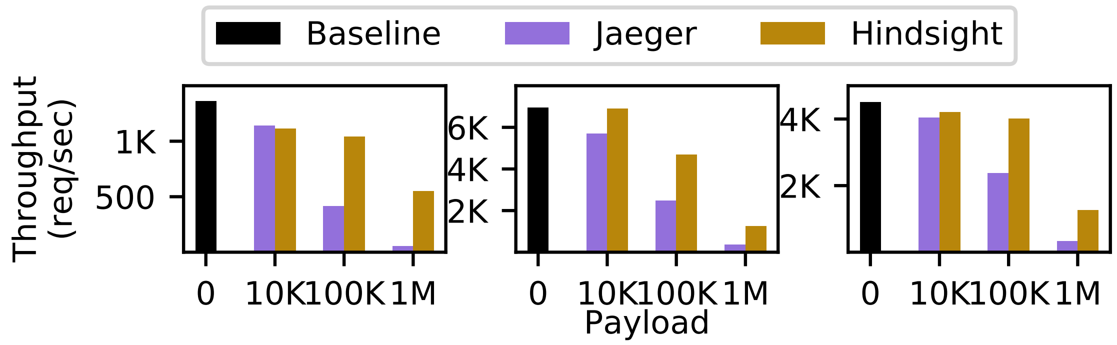


Figure 3.12: **Comparison with Jaeger.** Throughput with three workloads (compose-post, read-home-timeline, read-user-timeline).

that uses X-Trace to create traces and log 1,000 X-Trace events. With 8 threads and 32 kB buffers, the client generates 1.33 million events/s, totaling 259.6 MB/s and consuming 66,458 buffers/s—well below Hindsight's peak ingestion throughput. By comparison, X-Trace's client library experiences local performance bottlenecks and drops data from overfull queues, only reporting data at 5.9 MB/s. This underscores how a dedicated dataplane design can yield substantial performance improvements.

We also offer numbers from HDFS [148] (cf. subsection 3.6.1). We route Hadoop's debug-level logging into the Hindsight-modified X-Trace. A sustained request workload generates ≈30 MB/s from Hadoop's NameNode and 5 MB/s from each of Hadoop's DataNodes. In our 10-node experimental setup (subsection 3.6.1), this results in a total of 70 MB/s.

Comparison with Jaeger. In the final experiment, we deploy Hindsight in DSB on the cluster using **docker swarm**, interposing a Hindsight agent within each of the 12 DSB microservices. We augment the existing Jaeger instrumentation of DSB with breadcrumbs and masquerade Jaeger spans as **tracepoint** payloads. We use the built-in **wrk2** workload generator with compose-post, read-home-timeline, and read-user-timeline.

We compare Hindsight's performance to Jaeger by measuring the tracing overhead on DSB, as well as a baseline of DSB stripped of all tracing instrumentation. To saturate the systems, we use a 100% trigger rate for Hindsight and a 100% sampling rate for Jaeger. We repeat the experiment with two different per-tracepoint payload sizes of 10 kB and 1 MB, to compensate for DSB's short request length and minimal instrumentation. For apples-to-apples comparison, we add the payload to both Hindsight and Jaeger. Figure 3.11 and Figure 3.12 shows the end-to-end latency and throughput comparison. Hindsight displays significantly better performance than Jaeger at larger payloads. At 1 MB payloads, Hindsight boasts average latency of 3.2 ms and throughput of 549 req/sec compared to Jaeger's average latency of 66.7 ms and 56 req/sec.

3.7 Discussion

This section provides discussions on Hindsight design and usage, including some current limitations and some observations we have with our experience.

Event horizon. For Hindsight to coherently capture a trace, its trigger must both fire, and be fully disseminated, within the event horizon – otherwise an agent may have already evicted relevant data. We note that once an agent learns of a trigger, the trace data is safe from eviction and can be reported asynchronously in the background. In tandem with buffer pool size, Hindsight's event horizon is influenced by generated trace data volume. Lastly, for use cases extending even further into the past, Hindsight's optional trace percentage (cf. subsection 3.4.4) offers flexibility orthogonal to buffer pool size.

The event horizon is a function of data lifetime in the buffer and the trace reconstruction time. At fixed request rates, trace data is expected to remain in buffer as $\textit{data lifetime} = \textit{trace index threshold} \cdot \frac{\textit{buffer pool size}}{\textit{request rate} \cdot \textit{payload}}$, allowing the event horizon to be estimated by $\textit{data lifetime} - \textit{trace reconstruction time}$. As shown in Figure 3.10(e,f), 2,000 req/s, and payload is 16 kB/req, the expected data lifetime is $80\% \cdot \frac{800\,\mathrm{MB} \cdot 1{,}024\,\mathrm{kB/MB}}{2{,}000\,\mathrm{req/s} \cdot 16\,\mathrm{kB/req}} = 20\,\mathrm{s}$, yielding an event horizon that is three orders of magnitude longer than the trace reconstruction time of ≈10–20 ms, the event horizon time is then about 19.98 s. In practice scenarios where tracing is desired, event horizon can be configured to up to minutes or more.

Adaptive tracing. Collecting cross boundary information is challenging, especially considering nodes and servers are different from loads. Such difference directly affect trace data payload on each node. As we discussed in our design, data coherence is important to keep full traces of each request. Although Hindsight focuses a lot on maintaining coherence on local node, Hindsight doesn't maintain cross node coherence in the current implementation. We observe that payload linearly affect the event horizon. For example, in the DSB workload, one service (ComposePostService) has 6 times loads than other services so the event horizon is dominated by the trace data liveness on that node.

Though Hindsight does not solve the problem, we believe there are many practical solutions for further optimization. The general solution is to make tracing on each node adaptive. Buffer size and per tracepoint payload could be intelligently adjusted on each node. If there's strong intention to carefully trace a busy node, more memory size should be allocated. On the other hand, if a node does not have busy payload, then it should not allocate much memory space. Another dimension of adaptive operations is to change the sampling rate within the current rate limiting mechanism, which is designed to limit the tracing overhead but can also be helpful on reducing data generation rate, thus keeping more useful traces for less requests rather than keeping all pieces of many requests.

Trace collection. Hindsight implements a relative simple but effective trace collection mechanism with Hindsight coordinator. When operating a tracing system with large scale, such mechanism may delay trace collection and require further optimization. Propagating more middle-node pointers along the control flow would reduce the depth to retrieve trace data, which is effectively paralleling the trace collection. Our breadcrumb mechanism also reduce the collection complexity to half efforts.

A practical question is breadcrumbs may fail. In many cloud or microservices settings, a service has the knowledge of which server or node is it calling. This is especially true for many RPC-based frameworks. However, when the called function is not determined, a caller may fail to know the callee's tracing pointer, or address. For example, if there's a job scheduler or a pool sitting between two services, the scheduling mechanism will determine the next hop for execution, thus the initial hop cannot know where the request is going to. In those cases breadcrumb may fail. However, such mechanism is not fully deployed across services and in many cases, breadcrumb mechanism still works well as an optimization to cut down the trace retrieval time. In our practice, breadcrumbs can effectively speed up the trace collection.

Aggregated metrics. As Hindsight buffers trace data on local nodes, it can also provide aggregated metrics collection and calculation, which is also a major usage of common case analysis through other existing tracing systems. Because Hindsight agent has already maintained trace data indexing and managed data coherence, it's easy to integrate other aggregations for statistics. Taking latency as an example, a direct follow-up work is to calculate request latency distribution on local agents. Such task would add few engineering work and computational overhead. In the current implementation we limit our contribution to mainly building the tracing framework, however Hindsight can enable many future analysis tasks towards monitoring, debugging, and analzing with low overhead.

3.8 Related Work

Distributed tracing. Numerous prior works identify end-to-end requests as a useful granularity for slicing telemetry data and troubleshooting distributed systems. Example use cases include detecting anomalous request structures [167, 149, 101], diagnosing changes in the steady-state [57, 140, 127], modeling workloads [156, 117], and identifying resource and queue contention [167, 114, 77]. Distributed tracing systems have been presented in industry [149, 94], as open-source tools [19, 24, 22, 21], and in academia [75, 115]. We described how head-based sampling is widespread in practice (cf. section 3.1), and despite a desire for the ability to discriminate towards edge-cases, no techniques to do so have emerged to date. Tail-based sampling shares similar aspirations [101, 100]; its goal is to identify, from a collection of traces, which are most 'interesting' [39, 10, 15, 23, 17]. Yet tail-based sampling—perhaps a misnomer—does *not* capture fewer traces or reduce overheads on tracing backends; instead it is useful in tracing backends after collection and processing, for deciding whether traces can be discarded.

Network provenance. Hindsight is similar in spirit to network packet provenance systems that chronicle the history of network state, enabling use cases such as tracking the origin or path traversed by a packet across the network. Earlier systems, like ExSPAN [181] and SNP [182], adopt this abstraction; more recent works like SyNDB [95] and SPP [56] apply network provenance for packet-level root-cause analysis on Internet scale. Packet provenance systems primarily trace only packet metadata, which is well-structured and can be summarized in-band; these systems tackle additional trust challenges outside of Hindsight's purview. By contrast, handling metadata to reconstruct the path of a trace is but one concern for Hindsight; Hindsight is focused on handling arbitrary payloads (*i.e.* trace data), and the resulting performance, coherence, and fairness challenges. Hindsight also draws inspiration from works focused on temporal provenance [183] and packet reputation [55] in distributed systems, although Hindsight's tracing abstractions operate entirely at the application level.

3.9 Takeaway

Hindsight circumvents the false dilemma between performance and usefulness for diagnosing symptomatic edge cases by providing developers detailed traces from the recent past when they encounter symptoms of failures, performance regressions, SLO violations, or other issues among the requests that pass through their systems, while making use of standard instrumentation for distributed tracing and without burdening the backend. We believe the retroactive sampling abstraction, and our Hindsight implementation of it, can shift the conversation around tracing away from mechanism (how to collect traces) to a question of policy (what traces should be collected), and allow distributed tracing systems to support edge-cases analysis: a key use case for which they were originally conceived.

By this chapter, we have answered the question on how to monitor distributed system performance at scale, especially solving the open question about how to track edge case performance problems in real time.

Chapter 4

Optimal Data Placement on Memory Hierarchy

Problem. The goal of the memory hierarchy model for data placement is to carefully trade off properties of heterogeneous resources to optimize overall system utilization and performance. Historically, adjacent layers in the memory hierarchy (such as SRAM to DRAM, DRAM to SSD, SSD to disk) have differed in cost, capacity and performance by several orders of magnitude, readily supporting design decisions such as the inclusive caching policy whereby the blocks in a higher layer memory are also present in a lower layer cache.

As nascent memory technologies muddy the traditional distinction between layers in terms of storage capacity, latency, power, and costs, the assumptions underlying data placement decisions need to be revisited. Byte-addressable non-volatile memory (NVM), as one example, is slated to deliver larger capacity than DRAM at competitive latency, with currently available NVM hardware (*e.g.*, Intel Optane DC Persistent Memory [89, 69]) incurring $2-5\times$ the read latency and $4-10\times$ the write latency of DRAM (Table 2.1), far closer than the 2–3 orders of magnitude performance differences between DRAM and SSD. Similar data placement challenges exist in Non-Uniform Memory Access (NUMA) architectures, Non-Uniform Cache Access (NUCA) architectures, multi-tier storage systems, distributed caching systems, and across the CPU-GPU divide to name a few examples.

Secret Sauce. We posit that the entrenched cache replacement model for data placement in adjacent layers of a memory hierarchy fails to express and capitalize on opportunities brought on by these new technological realities. Using the DRAM-NVM interface as a running example, this work revisits the following two assumptions.

A1 (**Cache-Bypass**) First, the notion that each requested block needs to be brought into and served from faster memory is unnecessary when the slower memory is directly addressable, allowing for *cache bypassing* [123] or *cache admission* [48] techniques (Figure 4.1), discussed further below.

A2 (**Performance-Asymmetry**) Next, read (load) and write (store) operations can have asymmetric performance characteristics (Table 2.1), implying that the latency impact of a cache miss differs between request types. Consequently, the conventional approach of minimizing cache miss ratio as a proxy for optimizing latency fails to capture the nuance of how higher latency operations can be balanced against lower latency ones. For example, it may optimize overall latency to place a write-heavy block in DRAM instead of NVM, at the expense of a read-heavy block that would otherwise have seen more cache hits.

We will refer to cache policies that support Cache-Bypass and Performance-Asymmetry as *data placement* algorithms.

Solution.[1] We present an offline data placement algorithm across cache, memory, and storage hierarchies that optimizes latency while supporting **A1** and **A2**. Many recent works have considered these assumptions in isolation [71, 67, 123, 98, 69, 105, 96, 33]. When assumptions change and models are revised, the yardstick for what constitutes "good" performance within the model need to be adjusted as well, which underscores the need for offline optimal algorithms. Our approach follows the template of recent and ongoing work that revisits canonical memory model assumptions, such as by supporting variable sized items [49], accounting for cache write-back policies [44], and enabling caches to dynamically adjust their capacity [104].

[1]This work revises the previously published paper: *Optimal Data Placement for Heterogeneous Cache, Memory, and Storage Systems* at SIGMETRICS'20 [175]. The published paper is awarded with Kenneth C. Sevcik Outstanding Student Paper Award.

Under the hood, our algorithm, Chopt (for **CH**oice-aware **OPT**), casts the demand-based memory data placement problem within a network flow framework, then uses a Minimum-Cost Maximum-Flow (MCMF) algorithm to determine whether each requested memory block should be accepted into faster memory. To help accelerate trace simulation for larger workloads, we exploit sampling and show both empirically and theoretically that the scaled-up latency performance of Chopt on a spatial sample of a trace gives a faithful approximation of the performance on the original workload. Our analysis of spatial sampling of cache streams provides a rigorous footing for recent empirical results in the area [159, 158, 42].

Simulation results of Chopt on dozens of traces from diverse systems, including program memory accesses in the PARSEC benchmark suite (for the DRAM-NVM interface), block accesses from virtual machines (VMs) on hypervisors used in production (for multi-tier storage systems), and web cache accesses from a major content distribution network (CDN) suggest that average latency reduction of 8.2%, 44.8%, and 25.4%, respectively, are possible over Belady's MIN cache replacement policy (Table 4.4). By providing the best possible performance as a yardstick, offline trace simulation of Chopt can afford algorithm designers and operators greater visibility into defining and evaluating online data placement policies for their workloads.

4.1 Modern Memory Hierarchies

The memory hierarchy has been a guiding model since the beginning of computing, providing system designers a framework for managing complexity and reasoning about trade-offs inherent in combining very different memory hardware into functioning systems. Consequently, most systems until recently have required data in lower tiers to be addressed only indirectly via higher tiers (typically operating as inclusive or exclusive caches). Such abstraction was not only convenient because subsequent tiers of memory were orders of magnitude different in latencies and capacities, but it also made the CPU design process simpler and more optimized.

We suspect that the success of the strict memory hierarchy in CPU caches may also

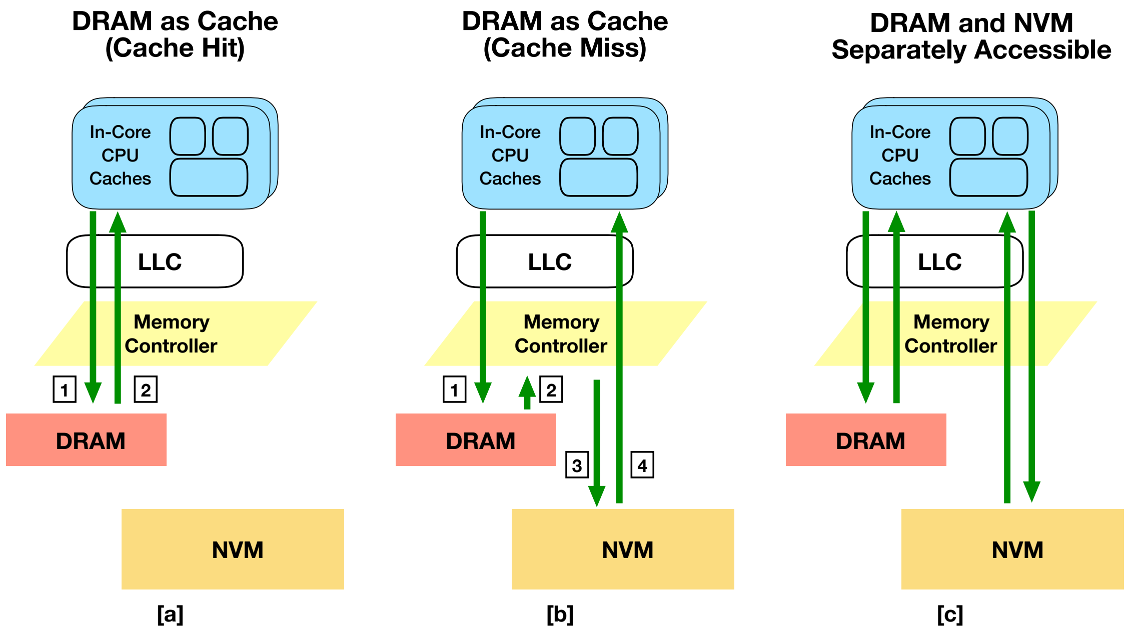


Figure 4.1: **Supported memory modes for DRAM and NVM.** If memory access is virtualized by the memory controller, DRAM acts as an architecturally invisible "cache" (a,b). However, unlike a traditional cache, data may also be loaded directly into the last-level cache (LLC) from NVM. The memory controller employs algorithms to dynamically manage data placement and therefore optimize hit latencies. On the other hand, if DRAM and NVM are separately accessible by the software layers (c), the OS or the application must decide data placement. *(a)* A requested block is found and returned from DRAM. *(b)* The block is not found in DRAM so the memory controller redirects the request to NVM. *(c)* Both DRAM and NVM are visible to the OS and the application that together dynamically control which blocks reside in DRAM versus NVM. **(a)** and **(b)** represent "Memory Mode" whereas **(c)** represents the "App-Direct Mode" supported by Intel's Optane DC processor [12].

have decelerated algorithm innovation and analysis into the more general data placement problem. For example, flash-based solid-state disks (SSDs) and hard disk drives (HDDs) have been equally addressable from an operating system (OS) point of view since at least the mid-2000s, yet most optimization research has treated the flash layer as a cache tier.

Addressability becomes even more important as new memory devices simultaneously diversify the characteristics of memory components (*e.g.*, data volatility and bandwidth) and decrease the performance differences (*e.g.*, less pronounced latency or capacity difference) between layers. We begin by surveying three domains where such developments are playing out.

Non-Volatile Memory. Intel recently released the Optane DC Persistent Memory [89] whose load and store latency are within the same order of magnitude as regular DRAM. Other NVM technologies are under development, including Spin-Transfer-Torque RAM (STT-RAM) and 3D-XPoint, and are expected to reach similar performance. Table 2.1 shows a performance comparison of DRAM, NVM, and NVMe memories. Intel Xeon™ scalable processors currently support DRAM and NVM together in their main memory systems [14, 37], with NVM poised to sit between DRAM and SSDs as an additional layer in the memory architecture. These NVM memories support direct access to data and optionally direct addressability from software. We illustrate the availables modes in Figure 4.1, including the case where NVM is accessed without involving the DRAM cache. As another layer, Intel Optane NVMe block devices have performance closer to SSDs and can be interposed between main memory and slower storage, with commercial deployment already underway [69].

Multi-Core Processors. The arrangement of the memory hierarchy among caches and DRAM on modern multi-core processors is also changing. Non-uniform memory access (NUMA) technologies allow direct addressability to data in DRAM on remote CPU sockets through an interconnect, forcing the OS to consider data placement among local and remote memories for maximizing overall space and bandwidth utilization at a lower latency. Similarly, last-level caches (LLCs) in a single CPU socket may be arranged in a non-uniform cache access (NUCA) architecture with different latency to different CPU cores.

Distributed Caches. Cache servers have been pivotal in accelerating the web and making services responsive, both within data centers through large-scale look-aside caches like `memcached` [4], and on the wider Internet through large-scale content delivery networks (CDN) operating on geographically distributed cache servers [86]. Internally, CDNs must solve challenges of data placement among nodes, cache pollution from "one-hit wonders", routing, and replication, all to minimize the latency experienced by end-users [142]. Each server in a distributed cache is internally addressable, allowing for CACHE-BYPASS, and *gets* and *puts* of objects may have asymmetric performance [48].

4.2 CHOPT: An Optimal Data Placement Algorithm

Having seen that data placement decisions across layers arise in multiple domains, we next consider how assumptions **A1** and **A2** can be incorporated into a generalized data placement model. We then define an offline optimal algorithm for the model, allowing algorithm designers to contextualize the level of effort that should be invested in developing online heuristics for the problem, and to evaluate the fruits of such labor.

4.2.1 Generalized Model and Objective

Let us consider two layers of directly addressable memory (L1 and L2). We assume L1 has a capacity of N blocks (or items), and that all data can be served from L2. We discuss extensions to more layers in Section 4.2.5.

We define a workload as a sequence $\vec{x} = (x_t)_{t=1}^{T}$ 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. A data placement algorithm processes the sequence $\vec{x}$ in order, and for each request x_t that is not already in fast memory (L1) makes an online decision whether to:

1. bring x_t into L1; note this may potentially evict another block on demand if L1 is full, or

2. serve x_t directly from L2 without loading into L1.

Note that (2) represents a CACHE-BYPASS (**A1**) decision.

Whereas cache performance is traditionally measured simply through miss ratio (or hit ratio) as a proxy for the average access latency, we wish to incorporate the performance asymmetry between read and write operations (**A2**) directly into the performance metric. These measurements can differ, *e.g.*, multiple low penalty load misses may be desirable over a costly write miss. Accordingly, we will directly measure and optimize average access latency throughout this work. Specifically, we let $\vec{w} = (w_t)_{t=1}^{T}$ with $w_t \in \mathcal{W}$ denote the latency penalty for each request, where $\mathcal{W} = \{W_\ell, W_s\}$ accounts for *marginal* latency penalty of load (read) and store (write) operations being served by L2 rather than L1. We assume without loss of generality that the latency of load and store operations in L1 are identically 1. Therefore, the access latency for request x_t is $W_\ell + 1$ (load operation) or $W_s + 1$ (store operation) if it was served from L2, and 1 otherwise.

In keeping with the two-layer DRAM-NVM memory hierarchy as a running example, let us refer to DRAM as L1 and NVM as L2. We set the latency of DRAM load and store operations as 1 and NVM load and store as 2 and 5 based on the hardware characteristics displayed in Table 2.1.

4.2.2 Why Investigate Offline Performance?

Identifying the optimal cache replacement strategies, Belady's MIN and Mattson's OPT (evict-farthest-in-the-future) [45, 118], was a critical junction in the creation of memory paging systems for two chief reasons. First, it allowed researchers to study the optimal decision making and incorporate ideas into the online heuristics. Second, and more importantly, it provided both a benchmark to beat and a gauge for success. For example, if evicting least-recently-used (LRU) pages were to yield a seemingly low 45% hit ratio on a trace, the result becomes relevant only when we learn that the clairvoyant OPT algorithm would obtain a hit ratio of, say, 48%. As such, evaluating offline optimal performance is the crucial yardstick in the recent wave of adopting machine learning algorithms to make caching decisions because these algorithms are specifically trained to imitate the optimal policy. Berger [47], for example, applied supervised learning to practically map object features to optimal decisions learned from offline analysis. Shi et al. [146] proposed to help design online hardware predictors with deep learning by training offline models of OPT decisions.

In 2020, we find that while OPT remains the touchstone for cache replacement algorithm performance, cracks are also forming. Recent papers have pointed out that as assumptions shift, such as when objects have different sizes [48, 49], when the cache size is dynamic [104], or when write endurance of the memory needs to be considered [58], the canonical cache model is inadequate, and with it, OPT. We thus ask: *Under the presented generalized data placement model, what latency performance can we hope for, even under offline conditions?*

4.2.3 Algorithm Design

We now introduce Chopt (**CH**oice-aware **OPT**imal), an optimal offline algorithm for the data placement under the generalized model defined above. The key idea behind Chopt is to represent memory hierarchy placement decisions as network flows, translating the optimal placement decisions into an instance of a Minimum-Cost Maximum-Flow (MCMF) problem. A similar approach was recently deployed by Berger et al. [49] in work concurrent with ours to evaluate the limits of optimal replacement under variable sized cache items. A chief difference is that the assumption of unit-size pages in our model sidesteps the knapsack/bin-packing style complexity and hardness that arises from arbitrary item sizes. Our preliminary results suggest our approaches can be combined but defer a full study to future work.

In Chopt, every access in the trace $\vec{x}$ is associated with an explicit node. Arcs connect both adjacent requests to simulate time (a *timeline link*), and requests for the same block. Positive flow on the *timeline links* implies requests should be served from L2 (NVM), whereas flow on the latter arcs implies that the block should be retained in L1 (DRAM) during the corresponding time interval. Chopt thus views each flow as the representation of a single memory slot in L1, tracking its occupancy sequence along with the workload, including swapping blocks in and out. Costs and capacities are associated to arcs to represent latency savings of serving data from L1 rather than L2, to ensure that each item is cached by no more than one L1 slot, and that the maximum number of concurrent flows is N. Chopt assumes that all requests are served by the NVM layer by default, and then calculates the "savings" from that baseline, where the maximum savings represents the minimum overall cost for handling the trace. Generally, replacing a block between the layers incurs a positive cost that we will seek to minimize, whereas accessing a block in the DRAM layer leads to a negative cost—a reward. The optimal algorithm is now reduced to a search for the MCMF flow solution.

4.2.4 The Anatomy of Chopt

We next provide a formal definition of the Chopt algorithm for an L1 cache of size N. We define a *cache schedule* of size N as a sequence of sets $C_0, \ldots, C_T$ where $C_i \subseteq I$ for all $0 \leq i \leq T$ with $C_0 = \emptyset$ and such that $|C_t| \leq N$, and where at most one block gets cached or evicted in each round, that is $|C_t \Delta C_{t+1}| \leq 1$ for all t. Here, $A \Delta B$ denotes the symmetric set difference $(A - B) \cup (B - A)$. We highlight that the schedule is *not* forced to bring the block currently being accessed into cache memory.

Graph Construction. We define a directed network G with $2T + 2$ nodes and up to $4T + 2$ edges. For each time point t between 1 and T, we add two nodes: one *main lane* node $\hat{x}_t$ for the time point, and another *high lane* $\hat{h}_t$ for the requested item $i = x_t$.

The directed edges are drawn as follows. First, we add arcs between simultaneous *main lane* and *high lane* nodes, specifically $\hat{x}_t$ and $\hat{h}_t$ for any t, which denote that the item could be swapped in or out. The capacity for both arcs is 1, and the cost is Z. These are *caching links* (pointing up to the *high lane*) and *eviction links* (pointing down to the *main lane*) for the item $i = x_t$ in question.

Second, for adjacent time points in the *main lane*, we add a forward arc $(\hat{x}_t, \hat{x}_{t+1})$ with infinite capacity and zero cost. The zero cost here indicates that storing data in L2, effectively a miss, offered no savings over L1. We call these *timeline links.*

Third, we add arcs for neighboring requests for the same item in the *high lane*, specifically $(\hat{h}_t, \hat{h}_{t'})$ where t' is the next time after t when item $i = x_t = x_{t'}$ is requested. These arcs each have a capacity of 1, and cost of $-w_t$ for $w_t \in \mathcal{W}$. Positive flow across this edge means a cache hit in L1 and so the latency of a miss was saved. We call this edge a *retention link* for item i.

At last, we add a final arc from a source node $s \in V$ to $\hat{x}_1$ with capacity of cache size N and cost of 0, and a zero-cost and infinite capacity arc from x_T to sink node $t \in V$. The

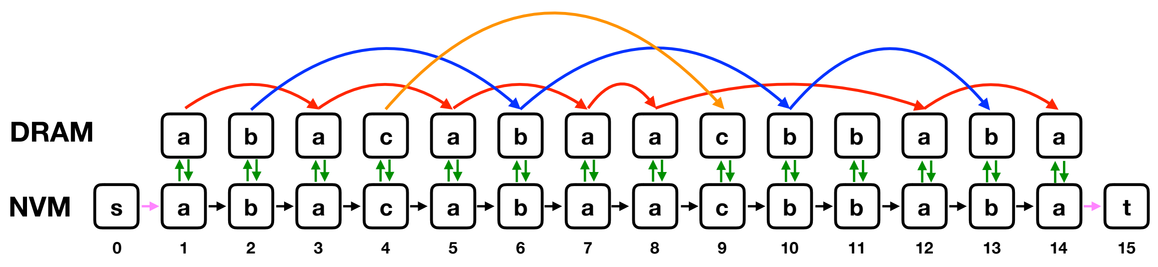


Figure 4.2: Example of a network flow constructed by Chopt, on a trace covering 14 requests for three unique items. Nodes and colored edges are described in Section 4.2.4.

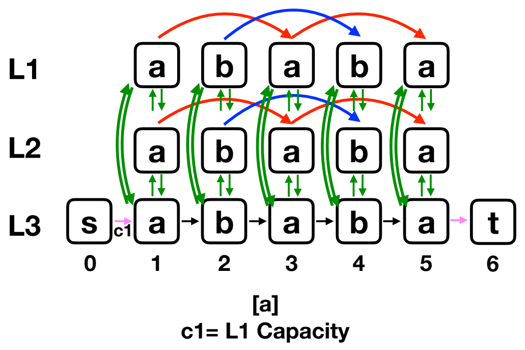
source node arc acts as a *choke link* to limit the cache size.

Optimization. We now run a MCMF algorithm on G [81]. Because each arc has an integer capacity, the ensuing optimal flow is integral. The resulting flow is interpreted with respect to a cache schedule as follows. Positive flow across a cache link at time t for item i indicates whether or not item i should be brought into L1 (flow of 1) or stay in L2 (flow of 0). Similarly, positive flow on an *eviction link* states that item i is no longer needed in cache at that time t. If there is any positive flow on a *retention link* $(\hat{h}_t, \hat{h}_{t'})$ for item i, which must equal 1, then item $i = x_t = x_{t'}$ remains in cache between t and t'. In this way, given a fixed cache size N, the minimum cost maximum flow implies a schedule for the cache: which items are swapped in and out at what time.

Illustrative Example. Figure 4.2 shows an example of the graph constructed by Chopt for a trace of 14 requests to blocks a, b and c. Each request is represented by two nodes: an upper one corresponding to the DRAM (*high lane*) and a lower one corresponding to NVM (*main lane*). We also show the source node s and sink node t. The nodes are connected by multiple types of edges which we differentiate by color. We explain each type of colored edges as below and reintroduce their cost and capacity in the example.

- **Green** edges represent the *caching links* and *eviction links*, denoting replacement operations. The cost of replacement in each layer should be equal to the latency of a store on that layer, so a green edge going upper or lower in the figure has cost of 1 and 5. Since at most one block can be replaced on a request, the green edge capacity is 1.

- **Black** edges represent the *timeline links*, and act as a baseline – all blocks are assumed



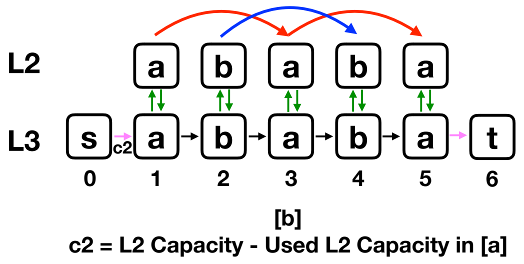


Figure 4.3: **Extending Chopt to more layers**. First, set *choke links* with capacity c_1, apply the Chopt algorithm on **(a)** to solve for data placement, reconstruct the graph as per **(b)**, modify the *choke link* capacity, apply Chopt algorithm again, and finally combine the placement solutions from **(a)** and **(b)**.

to be in NVM unless specifically moved to DRAM. Because the objective function calculates the maximum latency improvement over exclusively using NVM, the capacity of the black edges is $+\infty$ and their cost is 0.

- **Red**, **Blue**, and **Orange** edges represent *retention links*, which imply DRAM accesses on the request. The different colors represent accesses for different blocks. From each block's view, flow across the edge means that the block is cached in DRAM at the time of the request. Accessing the block in DRAM can save cost compared with NVM, so the cost for those edges is $W_\ell - 1 = 2 - 1 = 1$ if the request is a read, and $W_s - 1 = 5 - 1 = 4$ if the request is a write. Only one cell and thus flow should hold an item, and thus the capacity of those colored edges is set to 1.

- Pink edges represent the *choke links*. The capacity of a pink edge from the source node s is the DRAM size N, which in turn controls the maximum number of flows. The cost of a pink edge is 0.

Correctness. By construction, the MCMF over the graph implicitly considers all possible data placement options for the given trace. The proof of correctness for Chopt is mostly routine and established by two lemmas. As a special case, they establish the optimality of the cache-bypass policy OPT_b that was investigated without proof by Michaud [121].

We provide proofs of key assertions here.

Lemma 1. Each feasible minimum-cost flow in G is a cache schedule.

Proof of Lemma 1. For a given time point $0 \leq t \leq T$, let S_t denote $\{s, \hat{x}_1, \ldots, \hat{x}_t, \hat{h}_1, \ldots, \hat{h}_t\}$. The total flow across edges in the cut $(S_t, V - S_t)$ cannot exceed the maximum flow of the graph, which is upper bounded at N by the choke point $(s, \hat{x}_1)$. The *retention links* $e_1, \ldots, e_k$ with positive flow across the cut has capacity of 1, and thus a flow of 1. By construction, the retention edges all have different source nodes $\hat{h}_{i_1}, \ldots, \hat{h}_{i_k}$ where $i_j \in \{1, 2, \ldots, t\}$ for $1 \leq j \leq k$. The set of items in cache after the request at time t is correspondingly $C_t = \{x_{i_1}, \ldots, x_{i_k}\}$.

At most a single *retention link* terminates at $\hat{h}_t$, whose flow (if any) can either continue through an *eviction link* $(\hat{h}_t, \hat{x}_t)$ or another *retention link* for item x_t. In the former case, there was an eviction of item x_t at time t, or $x_t \in C_t - C_{t+1}$. At most a single *caching link* can be traversed from $\hat{x}_{t+1}$ to $\hat{h}_{t+1}$, and positive flow across this arc means that $x_{t+1} \in C_{t+1} - C_t$, or that item x_{t+1} was brought into cache at time $t+1$. Because the *caching link* and *eviction link* at a given time t both have capacity of 1 and positive costs, a minimum-cost flow would not simultaneously carry positive flow over both arcs: simply removing the flow over both links retains feasibility (per-node flow is conserved) and yet reduces cost, thus producing a cheaper feasible flow than the minimum-cost one – a contradiction. Thus no other flow across the cuts impact the sets C_t or C_{t+1}, implying they differ by at most one element (either one due to eviction or one from an item being brought into cache) or —$C_t \Delta C_{t+1}| \leq 1$ for all t. The sequence $C_0, \ldots, C_T$ is thus a valid cache schedule. $\square$

Lemma 2. Each cache replacement policy can be expressed in terms of feasible flow in network G.

Proof of Lemma 2. Let $\mathcal{P}$ denote a cache policy, and let $C_0, \ldots, C_T$ denote the cache schedule for $\mathcal{P}$ for the given workload $(x_t)_{t=1}^{T}$. Let $y_t \in C_t$ denote the (at most one) item x_t that $\mathcal{P}$ brought into cache at time t, specifically $y_t \in C_t - C_{t-1}$, and let $y_t = \perp$ if item x_t was already in the cache at time t (cache hit). Set $C_0 = \emptyset$. Similarly, let $z_t \in C_{t-1}$ denote the (at most one) item evicted at time step t, setting $z_t = \perp$ if no item is evicted.

Define the flow f_e over G as follows. For *high lane* links $e = (\hat{h}_t, \hat{h}'_t)$ with any pair $t < t'$, let $f_e = 1$ if $x_t \in C_t$ and $x_t \neq z_t$, otherwise $f_e = 0$. For *caching links* $e = (\hat{x}_t, \hat{h}_t)$, set $f_e = 1$

if $x_t = y_t$, and 0 otherwise. For *eviction links* $e = (\hat{h}_t, \hat{x}_t)$, set $f_e = 1$ if $x_t = z_t$, and 0 otherwise. For *timeline links* $e = (\hat{x}_t, \hat{x}_{t+1})$, set $f_e = N - |C_t|$. Also set $f_{(s,x_1)} = N$. All flows are therefore integral and no capacity constraints are exceeded.

We next show flow is conserved at every node. On one hand, the inbound flow to each *high lane* node $\hat{h}_t$ is at most 1 (from either a *caching link* or an incoming retention edge) and flows out (from either another retention edge or an *eviction link*, respectively). On the other hand, the flow into node $\hat{x}_t$ equals $f_{(\hat{x}_{t-1},\hat{x}_t)} + f_{(\hat{h}_t,\hat{x}_t)} = N - |C_{t-1}| + \mathbf{1}[x_t = z_t]$. This value in turn equals the outgoing flow

$$f_{(\hat{x}_t,\hat{x}_{t+1})} + f_{(\hat{x}_t,\hat{h}_t)} = N - |C_t| + \mathbf{1}[x_t = y_t]$$

because $C_t \Delta C_{t-1} \subset \{y_t, z_t\}$ and either $y_t = \perp$ or $z_t = \perp$, where we assume $\perp \notin A$ for any set A. $\square$

Lemma 3. (Chernoff bound) Let $X_1, \dots X_n$ be independent random indicator variables with $\mathbb{P}[X_i = 1] = p$ for $i \in [n]$. Set $X = \sum_{i \in [n]} X_i$ and $\mu = np$. Then,

$$\begin{aligned} \mathbb{P}[X \geq (1+\delta)\mu] &\leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu} \leq e^{-\frac{\min\{\delta^2,\delta\}\mu}{4}}, \text{ and} \\ \mathbb{P}[X \leq (1-\delta)\mu] &\leq \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\mu} \leq e^{-\frac{\min\{\delta^2,\delta\}\mu}{4}} \end{aligned}$$

with the former for $\delta \geq 0$ and the latter restricted to $\delta \in (0, 1]$.

Proof of Theorem 4.3.1. First assume without loss of generality that $w_t = w$ for all $t \in [T]$.

Then

$$\begin{aligned}
&\left|\mathbb{E}\left[\frac{1}{\alpha}\hat{m}(\alpha s) - m(s)\right]\right| \\
=&\left|\sum_{t\in[T]} \mathbb{E}\left[\frac{w_t}{\alpha}\mathbf{1}[\hat{r}_t \geq \alpha s]\mathbf{1}[Y_t = 1] - w_t\mathbf{1}[r_t \geq s]\right]\right| \\
=&\left|\sum_{t\in[T]} \frac{w}{\alpha}\mathbb{P}\left[\hat{r}_t \geq \alpha s\right]\mathbb{P}[Y_t = 1] - w\mathbf{1}[r_t \geq s]\right| \\
=&\left|w\sum_{t\in[T]} \mathbb{P}\left[\hat{r}_t \geq \alpha s\right] - \mathbf{1}[r_t \geq s]\right| \\
=&\left|w\sum_{t\in[T]} \mathbb{P}\left[\hat{r}_t \geq \alpha s\right]\mathbf{1}[r_t < s] - \mathbb{P}\left[\hat{r}_t < \alpha s\right]\mathbf{1}[r_t \geq s]\right|
\end{aligned}$$

$$\begin{aligned}
\leq& w\sum_{t\in[T]} \left|\mathbb{P}\left[\hat{r}_t \geq \alpha s\right]\mathbf{1}[r_t < s]\right| + \left|\mathbb{P}\left[\hat{r}_t < \alpha s\right]\mathbf{1}[r_t \geq s]\right| \\
\leq& w\sum_{t\in[T]} \exp\left(-\frac{\min\{\delta^2,\delta\}\alpha r_t}{4}\right) \\
=& w\sum_{t:r_t<\frac{s}{2}} \exp\left(-\frac{\alpha(s-r_t)}{4}\right) + w\sum_{t:r_t\geq\frac{s}{2}} \exp\left(-\frac{\alpha(s-r_t)^2}{4r_t}\right) \\
\leq& w\sum_{t:r_t<\frac{s}{2}} \exp\left(-\frac{\alpha s}{8}\right) + w\sum_{t:r_t\geq\frac{s}{2}} \exp\left(-\frac{\alpha s}{8}\right) = Tw\exp\left(-\frac{\alpha s}{8}\right)
\end{aligned}$$

for $\delta = \left|\frac{s}{r_t} - 1\right|$, where the first equality is justified by the linearity of expectation, the second by the independence of $\hat{r}_t$ and Y_t, the fourth by $\mathbb{P}[A] = 1 - \mathbb{P}[\overline{A}]$ for any event A, the first inequality by the triangle inequality, the second inequality combines the upper and lower-tail Chernoff bounds (3) using $\hat{r}_t$, recognizing that only either one of the two terms in the sum is non-zero for each $t \in [T]$, and the third one from that $\delta^2 \leq |\delta|$ when $0 \leq s \leq 2r_t$.

When the weights $a \neq b$ differ, we apply the bound separately for the subsequences $T_a = \{t \in [T] : w_t = a\}$ and $T_b = T - T_a$, obtaining an upper bound of

$$|T_a| a\exp\left(-\frac{\alpha s}{8}\right) + |T_b| b\exp\left(-\frac{\alpha s}{8}\right) = T\left(\xi a + (1-\xi)b\right)\exp\left(-\frac{\alpha s}{8}\right)$$

where $\xi = |T_a|/T$. □

4.2.5 Extending Chopt to Multiple Layers

Chopt is a general model that can serve as a building block when considering multiple memory or storage layers. We sketch how Chopt can support more layers in deeper hierarchies, keeping *average access latency* as our main metric. We assume that lower capacity layers are implicitly more valuable in terms of performance.

Suppose the memory hierarchy consists of three addressable layers L1, L2, L3, as shown in Figure 4.3(a). We expand the definition of nodes so that at each time point, we still have one *main lane* node but with two *high lane* nodes – the core modification of the graph structure. Assume without loss of generality that $c_1 < c_2 < c_3$ where c_i represents the capacity of L_i. We also extend the definition of links: *caching links* and *eviction links* should connect each pair of *main lane* and *high lane* nodes at any time point. *Timeline links* remain unmodified since we only have one baseline. *Retention links*, arise only in the *high lane* in a two-layer memory hierarchy, but since we have one more *high lane*, *retention links* should be constructed within each *high lane.* Note that we only added links for the additional *high lane*, and that node and arc roles have otherwise not been changed. The capacity and cost are unchanged, except *caching links* and *eviction links.* Cost w_t on those links represents the writing cost on the lane of the link sink, so between every two lanes, either between a *high lane* and a *main lane* or between two *high lanes*, the cost of those links can be reset correspondingly.

The main challenge now is to consider the *choke links*, which represents the *high lane* capacity in a two-layer hierarchy. Considering that in memory hierarchies, the first priority is to maximize the utilization of the most valuable layer — L1 in our example — and the next is L2. This does not mean that the capacity of these layers is filled. For the multiple layer hierarchy with multiple *high lanes*, we respect the priority and maximize utilization for them layer by layer. Figure 4.3 shows the process **(a)** and **(b)** with an example. First, Chopt sets the *choke links* with capacity as c_1, and then runs Chopt to determine a placement solution. Chopt terminates when the capacity of *choke links* is reached, or no more negative cost cycles can be found in the residual graph (in the Figure, there is surplus capacity on the *choke links*). In the latter case, the Chopt placement solution is already optimal since the utilization of each lane is maximized. In the former, however, the utilization of L1 is maximized but that of L2 may not be. In this case, we should remove all *high lane* nodes in L1 as well as all related *caching, eviction*, and *retention links* from the graph. For all remaining links, we continue searching and the cost and capacity are unaffected. Since the target changes to maximize the residual utilization of L2, the capacity for *choke links* should be reset as the difference between c_2 and the capacity already used by step **(a)**. We refer to this new graph shown in Figure 4.3(b) as the *degraded graph.* We can now run Chopt again on the degraded graph to identify more placement solutions. Note that the new solutions are compatible with the L1 placement decisions because the *high lane* nodes in L1 were already removed. In this manner, Chopt determines placement solutions for three-layer memory hierarchies, and can be expanded to support more layers.

4.3 Analyzing Long Traces by Sampling

Chopt provides an optimal offline latency estimate for a trace of length T in $O\left(T^2 \log^2 T\right)$ time, with different worst-case bounds depending on what MCMF algorithm is used for optimization [81]. Yet analyzing offline optimal placement is primarily of interest for large-scale real-world workloads, often comprising at least $10^7 - 10^9$ requests [49, 104]. The running time of Chopt for such long traces can be prohibitive.

To make Chopt practical to use, we apply *spatial sampling* on the traces — a sampling over blocks rather than requests — to reduce the scale of the simulation. Spatial sampling has been used successfully in recent cache replacement work [159, 158, 42] and was shown empirically to accurately calculate miss ratios. In addition to using spatial sampling on Chopt, we investigate the analytic limits of the approximation provided via sampling, highlighted in Corollary 4.3.1.1.

Stack algorithms. Expanding on our earlier notation, we let $C_0^n, \ldots, C_T^n$ denote the cache schedule of size n for a cache policy $\mathcal{P}$. Following the fertile line of work started by Mattson *et al.* [118], we define stack algorithms as follows.

Definition 1. Cache policy $\mathcal{P}$ is a *stack algorithm* if its cache schedule adheres to the *inclusion property*, specifically that $C_t^n \subseteq C_t^{n+1}$ for all $t \in [T]$ and $n \in \mathbb{N}$.

Here, we used the bracket shorthand for integer ranges, $[N] := \{1, 2, \ldots, N\}$. Common examples of stack algorithms include LRU, LFU, and OPT [118].

Stack algorithms induce an ordering over the elements in cache. Specifically, we say that $i \prec_t i'$ for $(i, i') \in \mathcal{I}^2$ at time t if for some $n \in \mathbb{N}$ we have $i \in C_t^n$ and $i' \notin C_t^n$. The relation $\prec_t$ defines a partial order over $\mathcal{I}$.

Definition 2. $\mathcal{P}$ is *stable* if the sequence $(\prec_t)_{t\in[T]}$ has the property that for every $t \in [T-1]$ and item pairs $i, i' \in \mathcal{I} - \{x_t\}$ with $i \prec_t i'$ we also have $i \prec_{t+1} i'$.

The omission of x_t implies that the item requested at time t is the only item whose relative order may change at time t. In the case of LRU, for instance, the requested item x_t is moved to the front (most recently used) location of the stack while leaving all others unperturbed.

Definition 3. Define r_t as the *stack distance* of element x_t at time t, s.t. $r_t = |\{i \in \mathcal{I} : i \prec_t x_t\}|$.

The following observations are immediate.

Remark 1. The cache policy $\mathcal{P}$ of size s on trace $\vec{x}$ has a cache miss at time t for item x_t if and only if $r_t \geq s$.

Remark 2. LRU, LFU, OPT and Chopt are stable stack algorithms.

Generalized Miss-Ratio Curves. We measure the impact of sampling by studying how latency changes with cache size through a slight generalization of the well-known *miss-ratio curves* [159]. Let $\mathbf{1}[A] \in \{0, 1\}$ denote the indicator function for predicate A, such that $\mathbf{1}[A] = 1$ iff A is true, with the standard generalization to random variable A in a probability space.

Definition 4. A *weighted miss-ratio curve* (WMRC) $m : \mathbb{N} \to \mathbb{N}$ over $\vec{x}$ on policy $\mathcal{P}$ is a function that aggregates the weighted impact (latency) of cache misses for a cache of size $s \in \mathbb{N}$ under policy $\mathcal{P}$ and workload $\vec{x}$, formally

$$m(s) = \sum_{t\in[T]} w_t \mathbf{1}[r_t \geq s],$$

where $w_t \in \mathcal{W}$ is the marginal increase in latency for missing request x_t.

The request weights $\vec{w}$ will serve to differentiate the latency impact W_ℓ of reads (loads) and writes W_s (stores) into lower level cache, in which case $\mathcal{W} = \{W_\ell - 1, W_s - 1\}$.

Sampling WMRCs. Sampling has been shown empirically to provide fast and accurate approximations for miss ratio curves [158]; our analysis provides a theoretical footing for these results. We will focus on *spatial sampling* of the trace $\vec{x}$, where each item — not request — is independently included in the trace with probability $\alpha \in [0, 1]$, where α can be referred to as the sampling ratio.

Let $Y_t \in \{0, 1\}$ be an indicator random variable denoting the event that cache item x_t was sampled, in which case $Y_t = 1$. By assumption, $\mathbb{P}[Y_t = 1] = \alpha$. Note that the Y_t variables are themselves *not* pairwise independent since they could refer to the same cache elements. Let $\mathcal{I}_S \subseteq \mathcal{I}$ denote the set of spatially sampled items.

We now adapt our definitions to the sampled trace.

Definition 5. The *sampled stack distance* $\hat{r}_t$ is a random variable, defined as $\hat{r}_t = |\{i \in \mathcal{I}_S : i \prec_t x_t\}|$.

We have the following lemma:

Lemma 4. Either $r_t = \hat{r}_t = \infty$, or the sampled stack distance $\hat{r}_t \approx \text{Binom}(r_t, \alpha)$. Also, $\hat{r}_t$ is independent of Y_t.

Proof. For the first part, assume $r_t < \infty$ and consider the subsequence $x_{j_1} \prec_t x_{j_2} \prec_t \cdots \prec_t x_{j_{r_t}} = x_t$. Then

$$\hat{r}_t = \sum_{s \in [r_t - 1]} \mathbf{1}[Y_{j_s} = 1]$$

which is the sum of r_t independent identically distributed Bernoulli variables with probability α, thus $\hat{r}_t \approx \text{Binom}(r_t, \alpha)$. For the second part, note that the sum for $\hat{r}_t$ specifically excludes Y_t. □

Definition 6. The *sampled miss ratio curve* $\hat{m} : [N] \to \mathbb{N}$ over $\vec{x}$ and weights $\vec{w}$ is defined as the aggregate of the weighted impact from cache misses for cache policy $\mathcal{P}$ of size $s \in \mathbb{N}$

that observes only those requests x_t in $\vec{x}$ with $Y_t = 1$. Formally,

$$\hat{m}(s) = \sum_{t \in [T]} w_t \mathbf{1}[\hat{r}_t \geq s] \mathbf{1}[Y_t = 1].$$

Our main result is the following.

Theorem 4.3.1. (Spatial sampling theorem). For weights $\mathcal{W} = \{a, b\}$ with $0 \leq a \leq b$ and weight skew $\xi = |\{t \in [T] : w_t = a\}|/T$, we have

$$|\mathbb{E}\left[\hat{m}(\alpha s)\right] - \alpha m(s)| \leq T\alpha\left(\xi a + (1 - \xi) b\right) \exp\left(-\frac{\alpha s}{8}\right).$$

Corollary 4.3.1.1. When all weights are identically 1, the spatial sampling theorem states that $\hat{m}(\alpha s) \approx \alpha m(s)$ on average for any s with an error of at most $Te^{-\frac{\alpha s}{8}}$.

Summary. To avoid running CHOPT on a long trace, which would take prohibitively long, Theorem 4.3.1 establishes that CHOPT can be approximated for a cache of size s through the following procedure.

1. Spatially sample α-fraction of the blocks from the full trace T, producing shorter sub-trace S.

2. Run CHOPT on S with a cache of size αs to obtain average access latency of ℓ.

3. Estimate the average access latency of the original trace T for cache size s as $\frac{\ell}{\alpha}$.

According to the corollary, the absolute error for this approximation is no more than $\exp(-\alpha s/8)$ in expectation, meaning that the approximation is exponentially more accurate in larger cache sizes and higher sampling ratios. The corollary assumes read and write latency to be identical, which establishes the theorem for the special case of the conventional miss ratio curves from the literature. When the weights differ, there is an additional $\xi a + (1 - \xi)b$ factor on the error bound. We evaluate the empirical tightness of the bound below.

4.4 Results

We evaluate CHOPT through experiments on multiple types of real-world traces that focus on the following questions.

- Can CHOPT draw the optimal placement boundary for different types of workloads? How much improvement can CHOPT demonstrate compared to other state-of-the-art caching algorithms or placement policies?
- How do the two revisited assumptions affect data placement algorithms in memory hierarchy scenarios?
- Does spatial sampling provide useful approximations of real workloads? How accurate is it?

4.4.1 Traces and Workloads

We evaluate CHOPT on a variety of real-world traces on different workload categories, including memory traces, storage block traces, and content delivery network (*CDN*) traces. Throughout this section, we use original *CDN* traces as describe in the appendix. However, given their sheer size, we reduce the *Memory* and *Storage* traces in two different ways. In Section 4.4.3, we sample from the original traces to reduce runtime as described in Table 4.1. In Section 4.4.4, however, our experimental runtime would be astronomically high even with sampling so we choose to trim the traces before sampling.

	Memory	***Storage***	***CDN***
Number of Traces	15	106	7
Length ($\times 10^6$)	40–2120	3.2–2115	10
Sampled Length ($\times 10^6$)	≈ 10	≈ 1	≈ 10
Sampling Ratio (%)	0.5–25	0.05–20	100
Running Time (hours)	≈ 36	≈ 24	≈ 48
Unique Items ($\times 10^3$)	≈ 9.4	≈ 200	≈ 245

Table 4.1: Basic characteristics for *Memory*, *Storage*, and *CDN* workload categories. *Length* represents the number of requests; *Sampled* and *Sampling ratio* correspond to spatial sampling; *Running time* represents the execution time for running CHOPT to calculate offline placement policies on sampled (*Memory* and *Storage*) and original (*CDN*) workloads); *Unique items* represents the number of unique requests in the workloads.

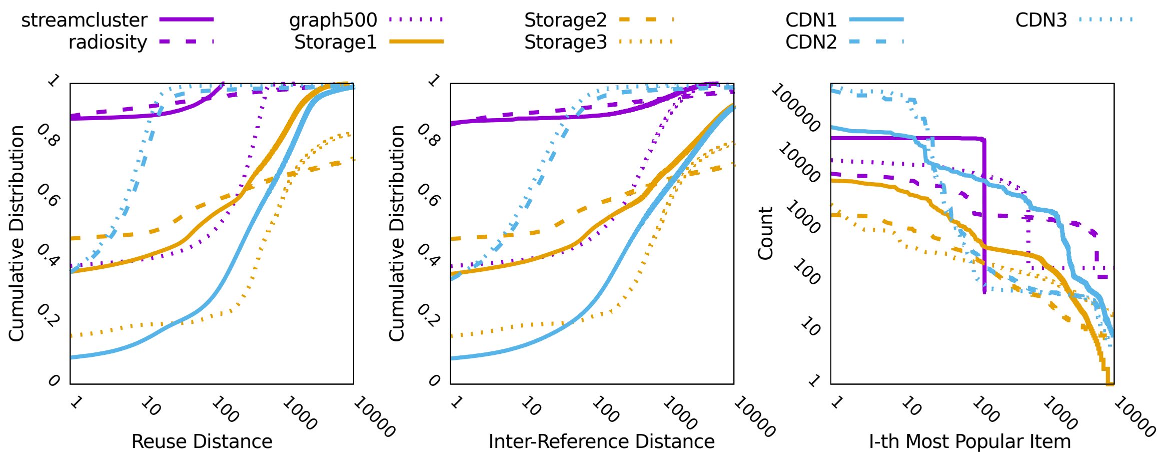


Figure 4.4: Trace characteristics, including reuse distance—the number of unique requests between two neighboring access of the same object, inter-reference distance—the number of requests between two neighboring access of the same object, and object popularity throughout the trace.

For *Memory* workloads, we use PARSEC [50] suite that include a variety of benchmarks with different execution characteristics and memory access patterns. We collected all the memory traces of 14 PARSEC benchmarks and a parallel breadth-first search algorithm for multicore single-node systems on the Graph500 [124] benchmark at page level using a Pin [110]-based profiler we developed. Our profiler leverages binary instrumentation to capture all the memory operations a specified program makes. The profiler emulates the CPU-level cache internally to filter out the cache-lines which could be cached on the CPU caches so the resulted trace would represent the operations that only end up accessing the main memory. For *Storage* workloads, we use 106 week-long disk access traces in production storage systems [159]. For *CDN* workloads, we use a cache trace from a major content distribution network that consists of week-long end-user requests for video-on-demand, streaming video, downloads, and e-commerce contents. Table 4.1 shows basic characteristics of our traces, and Figure 4.4 provides some characteristics of the workloads we used in our evaluation. The characteristics include the cumulative distribution of reuse distance of accesses throughout each trace, the inter-reference distance between accesses, and the popularity of i^{th} most popular item as a function of i. The trace names match those we showed in Figure 4.5.

For *Memory* and *Storage* traces, we use variable sampling ratios to unify sampled trace

	CHOPT	BELADY	BELADY-AD	LRU	W-TINYLFU
A1	$\surd$	$\times$	$\surd$	$\times$	$\surd$
A2	$\surd$	$\times$	$\times$	$\times$	$\times$
Online	$\times$	$\times$	$\times$	$\surd$	$\surd$

Table 4.2: Modeling assumptions for different algorithms.

lengths, at around $10M$ and $1M$ correspondingly for the two types. For *CDN* traces, since the original trace length is only around $70M$, we split the trace into 7 sub-traces, each containing about $10M$ requests. Traces in each workload type contain different number of unique requests. Typically, *Memory* traces have far fewer unique requests than the other two workload types, since *Storage* and *CDN* workloads usually contain behaviors like scans, when a sequence of many low frequency requests occur, and bursts, when a small group of high frequency requests occur.

Despite many other related works [49, 104] focusing on variable object size caching, we only focus on memory hierarchy and assume a unit object size for each workload type. For *Memory* traces, we assume each object is 4KB as the page size. For *Storage* traces, we assume 64KB as the block cache size. For *CDN* traces, we assume 64MB since the contents are mostly video based which indicates relatively larger object size than normal requests. Since we apply sampling on original traces, the cache sizes shown in our results are inversely amplified as numbers on non-sampled traces.

4.4.2 Experimental Setup

Implementation. We implemented an offline simulator for CHOPT in C++, where we apply the Bellman-Ford algorithm for solving the MCMF problem [162]. We use an Intel Xeon CPU E5-2670 v3 2.30GHz system for simulating our experiments. The running times for calculating optimal data placements by CHOPT on our workload traces are shown in Table 4.1.

Caching Policies. We implemented other prominent caching algorithms for comparing data placement results with CHOPT. Belady's MIN (named as BELADY in the results) policy is the authoritative offline algorithm for optimal cache placement, evicting the item that will be used farthest in the future—if at all. Since the original BELADY does not assume **A1**, we modify it to allow admission control, called BELADY-AD. Specifically, BELADY-AD considers the next access for the currently requested object, but bypasses any object whose next access is farther in the future than all other objects currently resident in the cache. We also implemented the Least Recently Used (LRU) algorithm as the most commonly used caching algorithm, and W-TinyLFU [67] as the state-of-the-art for a cache admission control policy. W-TinyLFU relies on request histories for making cache replacement decisions. The key idea is to maintain a freshness mechanism through lightweight counters. Table 4.2 shows how each of these algorithms meet our assumptions **A1** and **A2**. Since none of those original policies consider performance asymmetry, we evaluate **A2** by varying simulation configurations as detailed below.

Memory Model and Configuration. We apply a two-tier DRAM-NVM memory hierarchy in our experiments for evaluating Chopt's performance. We use normalized performance configurations based on the real measurements in Table 2.1, where DRAM load/store latency is normalized to 1, and NVM load/store latency as either 2 and 5. Since no comparison algorithms or policies assume **A2**, we evaluate Chopt on another configuration where the normalized DRAM latency remains the same while both NVM load/store latencies are set to 5. For each trace, we vary the cache size from tiny to large enough to cover all unique requests in the trace instead of configuring arbitrary cache sizes for each workload category. This allows all performance patterns to be shown in our results while avoiding occluding unexpected results from arbitrary cache configurations.

Metrics. The key metric for our evaluation is based on access latency, since data placement performance is more expressive than the proxy of hit ratio. We use Normalized Average Access Latency ($NAAL$) to capture the average latency on placement for each access. For apples-to-apples comparison of Chopt placement relative to other algorithms using workloads from different categories, we use the Relative Latency Improvement (RLI) to measure the percentage of Chopt latency performance over any other algorithm. The $NAAL$ of Chopt, denoted as $NAAL_0$, and that of another algorithm, say $NAAL_1$, is expressed as $RLI = 1 - \frac{NAAL_0}{NAAL_1}$. When presenting the RLI results, we vary the cache sizes as a ratio of unique requests to consider the diversity of different workload categories. For the *Memory* category, we choose 0.1%, 0.5%, and 1% of unique requests as the cache size for the results. In contrast, for *Storage* and *CDN* categories, we choose 1%, 2%, and 5%

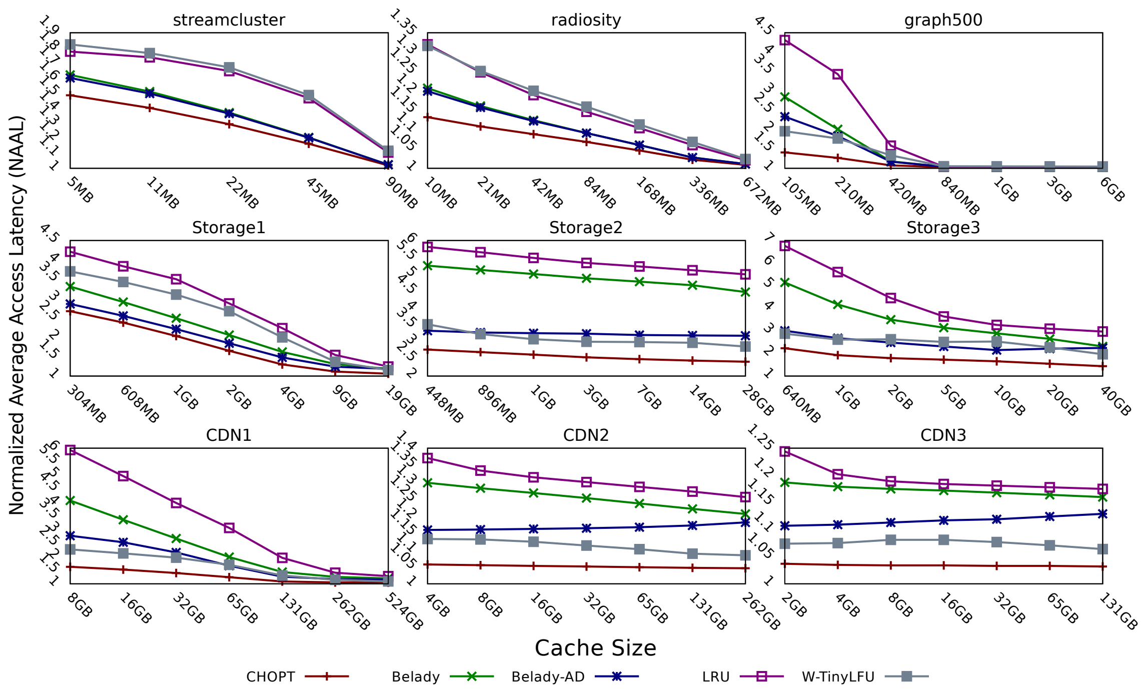


Figure 4.5: Normalized Average Access Latency ($NAAL$) results on three randomly chosen workloads from each workload category. Cache size is varied from very small to large enough to fit all unique items in the trace.

respectively. Finally, we also calculate hit ratios as a legacy comparison, as it also helps measure wear-out of memory layers.

4.4.3 Chopt Simulation Results

	A2	Chopt	Belady	Belady -AD	LRU	W-Tiny LFU
Memory	√	77.78	87.58	87.63	81.58	82.67
	×	85.45				
Storage	√	32.55	35.39	35.42	27.11	27.15
	×	31.42				
CDN	√	65.78	75.24	75.34	62.13	62.56
	×	71.76				

Table 4.3: Average hit ratio (%) under different **A2** asymmetry assumptions, specifically that NVM stores have respectively twice and 5× the latency of reads. Cache sizes are 1% of unique items in each workload.

Chopt Performance. Among all workloads in the experiments, Chopt provides better $NAAL$ than any other algorithm at any cache size. Figure 4.5 presents several results of

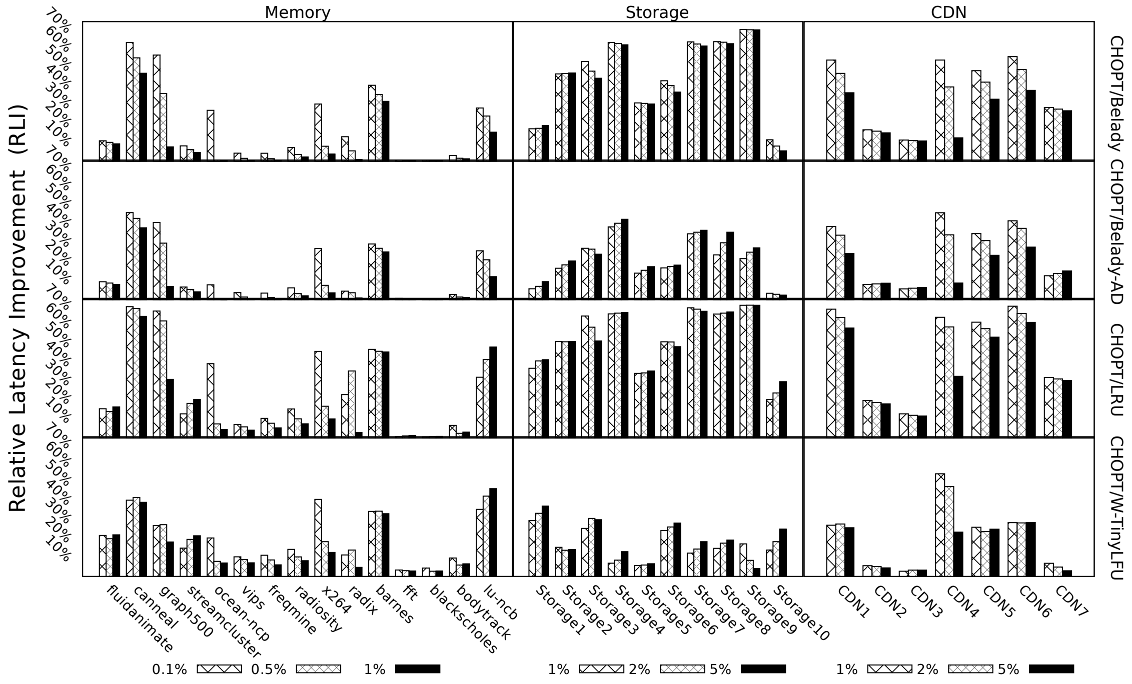


Figure 4.6: Relative Latency Improvement (*RLI*) results on single workloads over all workload categories (each in a column) algorithms (each in a row). Showing all *Memory* and *CDN* workloads, and 10 of 106 randomly chosen *Storage* workloads. Cache size is varied as $0.1\%, 0.5\%$, and 1% of unique requests for *Memory* workloads, while $1\%, 2\%$, and 5% for *Storage* and *CDN* workloads.

	N_s	Belady Avg %	Belady Max %	Belady-AD Avg %	Belady-AD Max %	LRU Avg %	LRU Max %	W-TinyLFU Avg %	W-TinyLFU Max %
Memory	2	8.2	63.8	6.6	47.1	14.5	70.9	12.3	47.5
	5	7.2	61.5	5.5	43.7	13.7	69.0	11.3	42.9
Storage	2	44.8	74.0	22.3	71.0	53.1	80.9	17.7	49.6
	5	45.0	73.9	19.2	71.2	53.0	80.9	17.9	49.1
CDN	2	25.4	66.1	17.4	47.0	33.5	75.5	13.3	55.4
	5	23.1	64.6	16.2	43.7	39.9	74.4	11.1	52.6

Table 4.4: Relative Latency Improvement (RLI) results aggregated on all workloads and over all algorithms. Average result among all varied cache sizes. NVM load latency NVM_ℓ is 5. Considering with and without **A2** assumption through configuring NVM store latency N_s (stands for NVM_s) as 2 and 5 respectively.

$NAAL$ on randomly chosen workloads, three from each workload category, under our normal configuration with **A2**. In those examples, Chopt always provides $NAAL$ less than 2 except at small cache sizes, which indicates good performance with respect of latency. Figure 4.6 presents results of RLI with workload specific chosen cache sizes as discussed above, and with the same configuration. This includes all workloads from *Memory* and *CDN* categories, and 10 randomly chosen workloads from *Storage* category. Trace names shown in Figure 4.5 also represent the same trace as in Figure 4.6. Table 4.4 presents aggregated results of RLI on all workload categories and over all other algorithms, with all varied cache sizes and under configurations with or without **A2**. From the results, Chopt provides average RLI at 8.2% over Belady on *Memory* workloads, and 44.8% and 25.4%, respectively, on *Storage* and *CDN* workloads. Among all algorithms, Chopt provides RLI of 6.6% – 53.1% on average, and even up to 74.0% over Belady on storage workloads. This exposes significant room for further improvements on data placement policies over the memory hierarchy.

Cache Bypass. As shown in Table 4.2, Chopt, Belady-AD, and W-TinyLFU considers **A1** while Belady and LRU do not. From Figure 4.5, algorithms considering **A1** often perform better. In *Storage2*, for example, algorithms with **A1** yield $NAAL$ between 2.5 – 3.5, while others have 5 – 6. Even if we define RLI for comparing improvements for Chopt over other algorithms, RLI can also contrast these algorithms on the same workload category, where a lower RLI by Chopt means superior latency performance. Figure 4.6 also suggests that algorithms considering **A1** provide lower latency in several workloads, especially in the *Storage* and *CDN* categories. In Table 4.4, the RLI over Belady-AD and

W-TinyLFU are 22.3% and 17.7% for the *Storage* workloads, whereas the improvements are 44.8% and 53.1% over Belady and LRU. A simpler explanation for the importance of **A1** lies in comparing the controlled results between Belady and Belady-AD, where **A1** is the only variable. From Figure 4.5 and Figure 4.6, all workloads show superior *NAAL* and *RLI* on Belady-AD over Belady. From Table 4.4, Belady-AD yields better *RLI* over Belady, with minimal improvements on the *Memory* traces but significant savings in the *Storage* and *CDN* categories.

Performance Asymmetry. Table 4.4 presents experimental results for different **A2** assumptions, where $NVM_s = 5$ means same NVM load/store latency as when disregarding **A2**. The results show that Chopt still provides *RLI* over other algorithms even without considering **A2**. This indicates that Chopt provides optimal data placement even without **A2**, while considering performance asymmetry is necessary for accurately making data placement decisions.

Hit Ratio and Endurance. Table 4.3 presents hit ratio results for all algorithms in configurations that either include or exclude **A2**. Note that different configurations only affect Chopt placement results. Since Chopt mainly focuses on performance related metrics, it provides lower hit ratios than Belady and Belady-AD. However, comparing with online algorithms LRU and W-TinyLFU, Chopt still provides higher hit ratios across all workload categories. Though not directly correlated, higher hit ratio indicates higher endurance on slower memory through mitigating swaps between layers. Observe that the difference of hit ratios between Belady and Belady-AD are not significant, which supports our idea to evaluate cost-aware metrics instead of only hit ratios when considering the **A2** assumption.

Running Time. Table 4.1 also shows the average execution time for each workload category. *Memory* workloads take 36 hours on average to calculate Chopt decisions, which is shorter than for the *Storage* and *CDN* categories. We remark that *Storage* workloads have very high ratios of unique requests. Chopt is based on solving the MCMF problem, so the execution time depends on the complexity of constructed network. The complexity is affected by many factors. Larger trace length increases nodes of the graph, for instance, and having more unique items increases the number of edges in the graph.

Lessons. There are some common workload related performance patterns we exhibit in our results. In the examples of *Memory* workloads, different algorithms perform differently when cache size is small, and converge when cache size gets larger. In the *Storage* and *CDN* workloads, many of the examples show a relatively bigger gap between algorithms. As discussed above, this is affected by many factors like unique requests in workloads. When there are too many unique requests, meaning that the frequency for each item is relatively low, the optimal placement policy may decide to bypass most of these requests. However, since the caching performance is highly workload related, workloads from the same category may also perform differently. For example, in Figure 4.5, *Storage1* and *Storage2* provides different patterns. This illustrates that determining optimal data placement is non-trivial.

$NAAL$ reflects how placement decisions affect latency, where a small $NAAL$ indicates possible less unnecessary movement between memory layers. For example, when the cache size is big enough and all objects are frequent, each request will be swapped into faster memory layer initially and not evicted, implying that the $NAAL$ will be close to 1. When the cache size is relatively small, however, caching any of infrequent request may hurt the overall latency so the optimal decision is to keep all objects in the slower memory layer, where the $NAAL$ is around 5. Yet a big $NAAL$ does not imply poor caching performance. For example, in *Storage2* from Figure 4.5, $NAAL$ for CHOPT is more than 2.5. This indicates that CHOPT decides to bypass many infrequent requests. We discuss special workload behaviors in next section.

4.4.4 Spatial Sampling Accuracy Results

Method. To thoroughly evaluate the accuracy of spatial sampling with CHOPT, we generate a total of $4,080$ sampled traces as follows. First, in each category, we either use the original traces as is (7 *CDN* traces), or we trim them down (to the *initial* $10M$ and $2M$ requests for 15 *Memory* and randomly selected 12 *Storage* traces (due to limited time). We then generate 30 different sampled traces for each of a variety of sampling ratios (1%, 5%, 10%, 20%) by varying the random seed used for sampling. Finally, we run CHOPT on all generated traces with varied cache sizes and calculate the $NAAL$. All in all, we ran more than $20,000$ separate simulation experiments, including full CHOPT without sampling, to characterize

Workload	**RLE**	**Sampling Ratio**			
		1%	5%	10%	20%
Memory	Avg %	0.20	0.16	0.14	0.03
	Max %	0.90	0.42	0.36	0.24
Storage	Avg %	3.67	2.06	1.18	0.50
	Max %	7.25	5.40	4.90	3.77
CDN	Avg %	4.08	2.17	1.62	0.93
	Max %	7.95	6.22	5.74	4.74

Table 4.5: Average and maximum Relative Latency Error (RLE) results for various sampling ratios over all workload categories. Each value represents the average of multiple experiments with different cache size configurations varied from very small to large enough to fit all unique items in the trace.

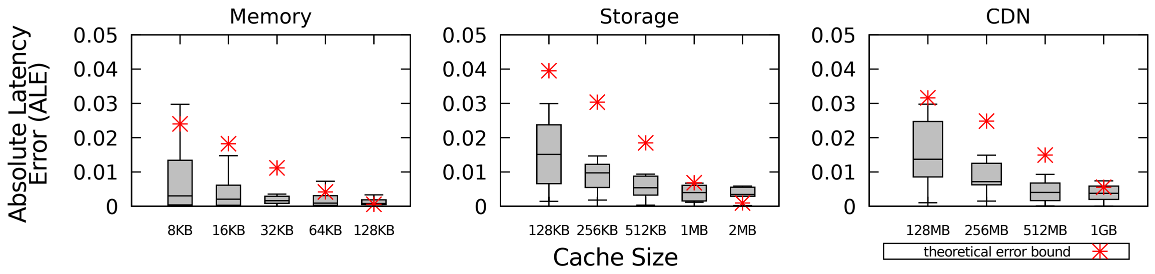


Figure 4.7: Absolute Latency Error (ALE) due to spatial sampling with Chopt. We show detailed results with a sampling ratio of 1% across selected cache sizes. For each cache size, we plot the theoretical error bound from the spatial sampling theorem, and a box plot of the distribution of ALE for different traces in each category.

sampling error. We also evaluate if the accuracy results approximate the theoretical bound as shown in Corollary 4.3.1.1.

Metrics. We use Relative Latency Error (RLE) to analyze the accuracy of sampled traces. According to our analysis, we evaluate sampled traces with sampling ratio α and cache size s through comparing the $NAAL$ with the original trace at cache size of $s \cdot \frac{1}{\alpha}$. We also calculate Absolute Hit Ratio Error ($AHRE$) as the difference in absolute hit ratios from original workload. This way, we can compare our sampling accuracy with prior results [159]that also evaluated the spatial sampling accuracy on the *Storage* workloads we used in our experiments. Additionally, we use Absolute Latency Error (ALE) for analyzing our theoretical bounds as shown in Corollary 4.3.1.1.

Sampling Accuracy. Table 4.5 shows the result on RLE for each workload category. The RLE on *Memory* workloads is only 0.2% at sampling ratio 0.01. For *Storage* and *CDN* workloads, RLE is larger 3.67% and 4.08% respectively. Table 4.6 shows the result on hit ratio and *AHRE* for each workload category. With the same sampling ratio, the *AHRE* is similar to *RLE* for all workload categories. We compare the *AHRE* with recent results [159] that claim absolute miss ratio error of 0.01 with sampling ratio of 0.01. Note that the main metric for CHOPT is *NAAL*, so we have to translate the accuracy with relative errors into a percentage. Given the variety of *Storage* workloads, we assume the miss ratio of workloads as $0.2 - 0.5$, so absolute error at 0.01 represents relative error of $2\% - 5\%$. In comparison, our sampling accuracy matches prior results by Waldspurger *et al.* [159].

Figure 4.7 shows *ALE* on each workload category for the 1% sampling ratio cases. We have already discussed how both larger sampling ratios and cache sizes contribute to better accuracy so here we focus on the edge cases where both sampling ratio and cache size are relatively small. Figure 4.7 demonstrates that our accuracy is close to the theoretical bounds in Corollary 4.3.1.1.

Lessons. As the results show above, spatial sampling is more accurate on *Memory* workloads than *Storage* and *CDN* traces. This is because those workloads usually have many unique requests where spatial sampling is limited to patterns in the original trace. We experienced in some cases that when sampled trace has a relatively small trace length but with many unique requests, the error on *RLE* and *ALE* can be large. To make running time feasible we apply relatively shorter original traces in our evaluation to curb running time. Comparing with our theoretical error bounds shown in Corollary 4.3.1.1, which is almost 0 when cache size and sampling ratio are relatively big, we still encounter some errors. In real-world use cases, where we usually apply sampling on very long traces like our main evaluation on CHOPT showed above, unstable patterns caused by sampling can be avoided.

Finally, as the results shown in Figure 4.7, errors can still exceed the theoretical bounds which compute an average case. For example, when cache size is $2MB$ for *Storage* workloads, the theoretical expected *ALE* is only about 0.0001 while experimental results have the expected *ALE* at around 0.005. We discuss this scenario in the next section.

Workload		**Sampling Ratio**			
		1%	5%	10%	20%
Memory	Hit Ratio %	92.51	90.43	89.17	87.81
	AHRE	0.25	0.24	0.24	0.22
Storage	Hit Ratio %	27.95	24.77	23.49	22.31
	AHRE	3.83	2.43	1.35	1.18
CDN	Hit Ratio %	88.07	80.95	78.54	76.28
	AHRE	4.46	1.99	1.89	1.38

Table 4.6: Hit ratio and Absolute Hit Ratio Error (*AHRE*) result over all workload categories.

4.5 Discussion

We provide some discussions on potential future directions and some limitations.

Chopt Provides Optimal Placement. CHOPT simultaneously addresses two fundamental caching questions: *When should an object be cached?* and *Which cached object should be evicted?* Each question embeds a notion determining of whether an object will be "hot" in the future. CHOPT considers all objects and the entire time line simultaneously to make optimal decisions, whereas practical online algorithms tend to view objects independently and instantaneously, and have access only to the past access history. By comparison, the offline BELADY algorithm, colloquially known as MIN, also looks towards the future but must accept all requests (no cache bypass as per **A1**) and ignore read/write cost asymmetry (as per **A2**). The intermediate variant, BELADY-AD, incorporates a cache bypass policy, allowing BELADY to reject cache requests for low utility objects that would immediately have been evicted. Yet this mechanism is a heuristic in that it implicitly assumes that all future requested objects will be cached. CHOPT takes such considerations while also addressing **A2**. The *RLI* results over BELADY-AD indicate that these assumptions are needed for making ideal placement decisions.

Improving Online Algorithms. The BELADY-AD algorithm is clairvoyant with respect to the *reuse distance*, a measure equaling stack distance for LRU and that is commonly tracked by online cache eviction algorithms. The *RLI* results over BELADY-AD and other algorithms suggest that orthogonal factors to reuse distance, such as request frequency, may be beneficial when evaluating eviction decisions on a range of workloads. By design, CHOPT considers all possibilities for data placement during its global optimization. The next goal is to produce simple heuristics that capitalize on patterns at both the local request and the global workload level.

Request Frequency Trends. Many cache algorithms like LFU and TinyLFU use object request frequency to determine the object locality. LFU performed significantly worse than other algorithms on our workloads, suggesting that frequency is not a panacea and must instead be considered dynamically. To this end, TinyLFU incorporates reuse history through "count decayed frequencies" that span multiple past intervals. Our analysis suggests that the frequency *trends* give an improved proxy for object locality on our workloads, but note that parameters such as interval width (number of requests per epoch) and number of intervals play a crucial role.

Reuse Distance Distribution. We compared the reuse distance distribution for requests where Chopt and other algorithms make different decisions. In some workloads, particularly in the *Memory* category, Chopt commonly rejects requests at a specific reuse distance that other algorithms accept into the cache, either with or without considering **A1**. The reuse distance profile may therefore be a crucial feature for classifying requests under our assumptions. A special case is the first occurrence of an object which we found Chopt to commonly ignore – a choice in line with earlier reasoning in the literature [67].

Global Patterns. Considering objects only in isolation overlooks the correlations that are exhibited in real-world traces. Patterns, such as sequences of frequent (a burst) or infrequent (a scan) requests for objects within an interval, are commonplace in *Storage* and *CDN* workloads. A useful heuristic is whether the data placement policy can identify a burst or scan globally and help with intelligent placement decisions. Optimal placement on a scan should bypass every request in the scan (like, *e.g.*, ARC [120]) instead of polluting the cache with unpopular objects. Bursts within relatively large object spaces present similar problems as scans, where the cache can be subverted by unnecessary swaps. For example, if the cache size is s and a burst contains $s+1$ frequently requested objects, Chopt admits only the most frequent s objects. Note that a near-optimal strategy is to cache any s objects only, so long as they are all sufficiently popular. Then the performance loss is driven by accessing one object directly from the slower layers. In contrast, other algorithms might accept all incoming requests that exhibit high locality of reference, either without considering **A1** or even awareness of a burst. The situation leads to suboptimal decision-making: unnecessary swaps for some frequent objects between memory layers, which Chopt avoids.

Extending Chopt. Chopt design transforms the data placement problem into a network flow problem, providing flexibility for more detailed questions. For example, we implement **A2** in Chopt by simply setting different weights on *retention links*. Other placement problems can be solved by defining proper configurations for performance, or by removing some links to account for hardware restrictions. For example, if we set *retention links* with large weights, to imply that caching any object amounts to huge latency savings, then Chopt attempts to provide placement results with minimal bypassing. Further, Chopt supports expanding problems to account for other metrics than latency. The main limitation of Chopt is the sharing model, where we assume an exclusive caching model whereby each object can only belong in one layer at a time. Although this assumption accords with similar work [49], expanding the underlying sharing model is an interesting future direction.

Spatial Sampling Accuracy. We evaluated the sampling accuracy, both theoretically and empirically. Our derivations showed that spatial sampling retains self-similarity of the original hit ratio curves with two types of error: distortion at low sampling ratios, and uncertainty for small cache sizes.

Figure 4.7 shows cases where the empirical error exceeds the bound on expected error from Theorem 4.3.1. We also witnessed that as sampling ratio α and cache size s increase and the theoretical error bound $e^{-\alpha s/8}$ rapidly converges to zero, empirical errors still occur. This phenomenon was also encountered by Waldspurger et al. [159] where they defined "sample size" to reflect the $\alpha \cdot s$ product of the sampling ratio and cache size. We believe better bounds on the higher moments of the error distribution, or even concentration bounds on the probability of error rather than only on the average, could shed more light on this phenomenon.

4.6 Related Work

Non-Volatile Memory. NVM technologies are already coming out of the labs to be used in production, sitting between DRAM and SSD from the performance characteristics point-of-view (latency, bandwidth, and density) [89, 12]. One of the key characteristics of NVM is being directly accessible, which enables CPU and DMA controller to access NVM without involving DRAM. The Linux community and Microsoft have already implemented direct access support on file systems [152] and there has been work towards representing NVDIMMs as volatile NUMA nodes transparently [165]. Read and write asymmetry is another characteristic of NVMs that has been studied to mitigate endurance problems and to improve the write performance [174, 136, 59, 184]. Philipp et al. [131] considered NVM asymmetries through clustering rather than secondary indexes and used heap organization of block contents to save unnecessary writes from DRAM to NVM. Sala et al. [139] proposed to perform a single read with a dynamic threshold to adapt to time-varying channel degradation for resolving NVM endurance problems caused by asymmetries. NVM is also widely used in building general purpose storage systems [99], storing deep learning models [70], and graph analysis [116].

Memory Hierarchy. NVM augments the memory hierarchy and may contribute various types of memory systems, typically with DRAM serving as a filter or a faster layer in the hierarchy for flexibility of performance trade-offs. Agarwal and Wenisch [33] presented huge-page aware classification in DRAM-NVM hierarchy for trade-offs between memory cost and performance overhead. Kannan et al. [96] provided guest-OS awareness during page placement under heterogeneous memories at compile time, enabling applications to control migrations only for performance-critical pages. Li et al. [105] estimated the benefit of page migrations between different memory types by considering access frequency, row buffer locality, and memory-level parallelism. Eisenman et al. [69] used NVM block devices in a commercial key-value storage system for reducing DRAM usage and total cost with comparable latency and QPS. Another common multi-level memory model is the exclusive caching model, which removes data redundancy and save spaces within cache layers, and problem is transferred to manage data placement and migration between layers as one. Wong and Wilkes [164] proposed DEMOTE techniques where data blocks can be ejected to lower level caches and managed by a global MRU. Gill [80] analyzed the insights into optimal offline performance of multi-level caches and provided an improved technique, named PROMOTE, considering inner-cache bandwidths and response times which could be problematic in DEMOTE. Some other works [180, 78] also discussed caching algorithms in exclusive models. While those works also focus on better cache utilization, they are still bounded with layering restrictions, whereas Chopt provides optimal placement bounds in a global form. Besides coordinating DRAM and NVM, researchers also have investigated into other memory hierarchies like using NVM for last level cache as replacement of SRAM [98], controlling flash write amplification with DRAM [71], and intelligent placing of packet headers near CPU to reduce tail latency in the LLC-DRAM hierarchy.

Cache Admission Control. New caching techniques have to consider cost-aware data placements under different characteristics in the denser memory hierarchy. Specifically, one must consider caching more costly objects but also bypassing objects of less value for future latency costs. Mittal [123] surveyed the power of cache bypass in different heterogeneous systems. Admission control is a caching technique that employs cache bypassing in practice. Einziger et al. [67, 68] proposed the state-of-the-art cache admission control policy, TinyLFU, to filter out infrequent requests and replace cached items with old history records, extended as Window-TinyLFU by adding an LRU filter to tame sparse bursts. Eisenman et al. [71] implemented admission control through a Support Vector Machine to classify objects with historical patterns. Recent papers also considered cache write-back cost for efficiency and endurance through identifying frequent written-back blocks and keeping them in LLC via partitioning [133, 161].

Optimal Placement Analysis. Various researchers have conducted theoretical analysis of optimal offline data placement. Farach-Colton and Liberatore [73] initially proposed to use network flow formulation for modeling local register allocation problems. Several other authors provide theoretical approximation and heuristics for optimal placement or general caching problems and also show that most variants of the optimal placement problem are NP-Hard to compute. Albers et al. [34] formulated general caching problems as integer linear programming questions, and proposed a relaxation of optimal placement problems. Bar-Noy et al. [41] proposed general approximation for resource allocation and scheduling problems with local ratio technique, which can also be applied to general caching problems. Carlisle and Lloyd [54] provided an algorithm on *k-coloring problem* which can be expended on weighted intervals and further solve job scheduling or register allocation problems. While these works provide insightful heuristics for contemplating caching problems, they lack practical algorithms or policies for real-world data placement analysis [46].

Offline Optimal Placement Policies. There are many practical works on offline optimal placement or caching analysis. Belady's MIN [45] is known as the standard offline optimal caching algorithm for basic cache assumptions. To the best of our knowledge, Berger et al. [49] and Li et al. [104] are two state-of-the-art offline optimal placement analysis results, with both papers focusing on variable object sizes caching problems. Our network flow approach was conceived independently of prior work [49] that had used it to model offline optimal variable-size cache eviction. Berger et al. provide a method to calculate offline optimal bounds *FOO* as well as a practical approximation for such bounds *PFOO* for real world storage and CDN workloads through rounding, rather than sampling as in our approach. Li et al. [104] proposed an offline optimal caching algorithm *OSL* which statistically predicts object lifetime with histories and assigns leases for cached objects. Offline optimal placement for variable-size objects are complementary to our problem with somewhat different assumptions. Both works also characterize the problem using weighted intervals, where object sizes are respectively represented by weights and the dynamic of placements depends on whether objects are cached or not. It is unclear how the approach can be generalized to memory hierarchies where interval weights also would need to characterize placement decisions, and a successful approach will likely require a direct integer program formulation of the problem.

Practical Data Placement Policies. Alongside TinyLFU [67], several recent papers have designed practical placement policies or algorithms, all of which rely on predicting future cache behavior based on the history. Beckmann and Sanchez [43] proposed prioritizing object eviction by their economic value added (EVA), an estimate of the expected number of hits for the object beyond that of an average object. The idea was then expanded for variable sized objects [44]. Some papers leverage offline analysis of past accesses by creating variants of Belady's MIN algorithm. Jain and Lin [91] predicted object futures by reconstructing Belady's MIN solutions over a window of past accesses. Jain and Lin [92] uses a similar idea to provide Demand-MIN for cache prefetching.

Sampling. Sampling techniques have been proven empirically to be efficient for measuring cache utilities with low overhead. Many recent caching or placement works have deployed spatial sampling [159, 97, 42, 134, 135, 158, 85] or temporal sampling [51, 163] to improve the efficiency. Spatial sampling has been cited as a remarkably robust statistic for constructing miss ratio curves to better reflect the common cache metrics like reuse distances, compared with temporal sampling. Our work is inspired by Waldspurger et al. [158], and expands on the literature by providing a solid theoretical foundation under these empirical results.

4.7 Takeaway

We considered the challenge of data placement between adjacent memory hierarchy layers when we move away from the established assumptions of always needing to bring in data to faster memory (Cache-Bypass), and that all requests are equally impacted by being served from slower memory (Performance-Asymmetry). After generalizing the memory model, we found that miss ratio (or hit ratio) no longer suffices as a proxy for average access latency, and that Belady's traditional optimal replacement policy MIN was inadequate. To measure the extent to which new algorithms need to be designed for this problem space, we presented a clairvoyant algorithm (Chopt) for optimal offline data placement algorithm and proved that Chopt can correctly provide an upper bound of performance gain for any data placement algorithm. To make Chopt feasible to run on large real-world traces, we proved analytically that spatial sampling gives a good approximation – a result of potential independent interest.

We ran Chopt on a variety of system workload traces, including main memory accesses of PARSEC benchmarks, block traces from multi-tier storage systems, and web cache traces from a CDN, and compared it with several cache replacement and data placement policies. Our simulation results on our offline data placement algorithms show that average latency improvements range between $8.2\% - 44.8\%$ beyond where OPT would have marked a line in the sand. We also evaluated spatial sampling performance empirically by running over $20,000$ simulations, showing it can approximate average latency with an average error of only 0.2% at 1% sampling ratio on the PARSEC benchmarks, and at most 2.17% for the sampling ratio of 5% across three classes of workloads. We conclude that CHOPT can efficiently calculate data placement decisions for diverse workloads on a two-tier DRAM-NVM memory hierarchy, and opens up a space for improving overall system performance and latency through new data placement algorithms that are now equipped with a critical performance yardstick: offline CHOPT.

By this chapter, we have answered the question on how to practically analyze cache system performances through understanding the optimal cache strategies, towards better building distributed systems.

Chapter 5

Estimation of Cache Warmup Time

Problem. In distributed storage systems or CDNs, operators may wish to discern how quickly after downtime or maintenance the server becomes useful again for serving content. They may wish to reason about how long to duplicate cache traffic to a new or recently restarted node before it can serve real clients at an acceptable hit rate. During recovery or reconfiguration of cache nodes, they may also wish to estimate how long the back-end storage servers must sustain additional load. In this manner, warmup time estimation allows CDN operators to compute the required redundancy and extra capacity to maintain a level of service in failure scenarios. In a shared memory or storage system, as another example, dynamic cache partitioning is often used to allocate storage resources to different processes or *tenants*. Here, when the partitioning controller decides to allocate more space to a tenant, it takes a period of time until the steady-state cache performance catches up. The controller needs to be cognizant of this delay to avoid instability whereby the partition keeps changing based on incomplete feedback gathered before steady-state has converged.

Secret Sauce. Caching systems, including storage systems, distributed databases, and content-delivery networks (CDN), are large and abound with configuration options that can be opaque not only to the engineers operating these systems, but also to their designers [53, 52, 106, 154]. Two tendencies are to either ignore the complexity and set parameters from ignorance and experience, or to treat the system as such a complex black box that it requires another black box, such as machine learning models, to interpret the potential impact of changes. Between these extremes are simple and intuitive approximate models, or *rules of thumb*, that are invaluable in many engineering fields to create intuitive and *sufficiently* correct understanding of the system.

Solution.[1]**.** I derive a rule of thumb expression for cache *warmup times*, specifically how long caches in storage systems and CDNs need to be warmed up before their performance is deemed to be stable. We first provide a concrete definition of cache warmup time, that is, a cache server has warmed up when its cache hit rate over time is and stays comparable (within ε error) to that of an identical cache service that processed the same workload but suffered no downtime. We then analyze dozens of traces across workloads collected from diverse systems, ranging from block accesses of virtual machines in storage systems to cache accesses of large CDN providers. We derive the following rule of thumb expression for operators to estimate warmup time of an LRU-style cache:

$$\text{warmup-time}(s, \varepsilon) \propto s^{p_s} e^{-p_e \varepsilon},$$

where s represents the cache size, and $\varepsilon > 0$ controls how closely hit rate should match that of a hypothetical cache server which was continuously running. Our experiments show that the p_s and p_e parameters concentrate at specific values for each type of workload. Our simulation results indicate that the formula provides an accurate expression for operators to estimate their cache server warmup time.

5.1 Dynamic Cache Behavior

Distributed memory caches are the cornerstone of today's content distribution networks (CDNs) and cloud storage systems for improving web service performance. A common architecture for a distributed cache is a collection of high-memory servers which is interposed between client nodes (sometimes actual end-users), and a storage service that interfaces with slower media, such as a disk-bound key-value database. When the server memory (or the memory dedicated to the tenant on a shared cache server) is exhausted, the server makes

[1]This work revises the previously published paper: *When is the Cache Warm? Manufacturing a Rule of Thumb* at HotCloud'20 [176]

space by evicting older data according to a replacement policy, which in practice is normally a variant of LRU—evict the least-recently used key-value pair [40]. A central feature of the distributed cache design is the complete independence of servers from one another. Independence reduces operation and implementation complexity, facilitates scalability, and allows reasoning about each cache server in isolation.

Operational dynamics. Most research on caches assumes they operate in steady-state. Yet understanding the cache behavior under exceptional circumstances is often crucial.

Failure recovery. First, distributed caches can comprise a vast number of servers [126], where individual server failures are common. Accurate assessment of recovery time becomes increasingly important for operators to decide when servers are ready for serving clients without imposing significant load on the storage layer or end-user perceived latency. We assume that the cache memory on the server is empty (*cold*) after recovery because stale cache data can produce application-level inconsistencies, even with sophisticated application-specific cache invalidation pipelines [109].

Load balancing. Second, consistent hashing does not account for key popularity, so some cache servers can become heavily loaded relative to others [87]. Manual or automatic adjustment of hash ranges to balance load [84] implies that some cache servers are responsible for key-value pairs they have not encountered before, thus impacting cache hit rate.

Cache sharing. Third, large cache installations are often shared between multiple applications or tenants to improve efficiency and quality of service, either implicitly [40] or explicitly [60]. Explicit sharing is implemented via cache space partitioning mechanisms [63] which means cache space allocation for tenants may change over time. Operators must estimate how regularly cache space can be re-partitioned, which in turn depends on how quickly the enlarged cache space for tenants becomes useful and indicative of the tenant's cache hit rate performance under steady-state [61].

Cache dynamics. Operators facing these scenarios would benefit from a rule of thumb to estimate when partially full cache memory has reached a "useful" steady-state and when applications can use the cache without burdening the storage layer or imposing miss latency on clients. Yet, quantifying cache warmup time is challenging due to several factors.

Cache hit rate performance is determined by the workload. Decades of effort has been spent on characterizing cache workloads, but historically focused on programmatic workloads (such as CPU caches) rather than in the context of human-driven behavior (such as web or CDN workloads) [86, 143].

Cache workloads are not static. As mentioned earlier, considering a cache server to be warmed up when a particular fixed hit rate threshold is reached ignores temporal popularity dynamics [155] and diurnal variability exhibited in CDN traces [144], among others. Even defining hit rate relative to the start of a trace embodies the same problems.

Cache performance depends crucially on the cache size. Recent attention on efficiently computing so-called hit rate curves – hit rate as a function of cache space – has illuminated how the relationship tends to be nuanced and volatile in real-world workloads [62, 138, 159]. Bonfire [177] uses temporal and spatial behaviors for doing proactive cache warmup, but does not take cache size into account when defining warmup time.

5.2 Understand Cache Warmup Process

Interval hit ratio. Warmup time must capture the notion of a cache "being useful", which in turn is related to its hit rate. But the classical notion of "hit rate", defined as the number of hits received over a number of accesses in a trace, relies on requests since the beginning of measurement being predictive of upcoming request—a degree of stability not present when cache workloads change dynamically.

To address variability in workloads, we measure cache performance by the *interval hit ratio* (IHR), defined as the ratio of cache hits in a relatively short past time window divided by the total number of requests in that window. This focus on the recent past adapts the metric to measure current performance with the ongoing dynamics. The IHR can be considered over subtraces of the full cache trace. The added flexibility allows us to also consider cache downtime, represented by a specific interval during the workload. We use $IHR(st, et, s)$ to denote the hit ratio computed for a short interval between start time st and end time et at cache size s. Below, each interval spans 1/1000 of the trace length.

Our analysis shows that even within the same workload type, workloads usually behave

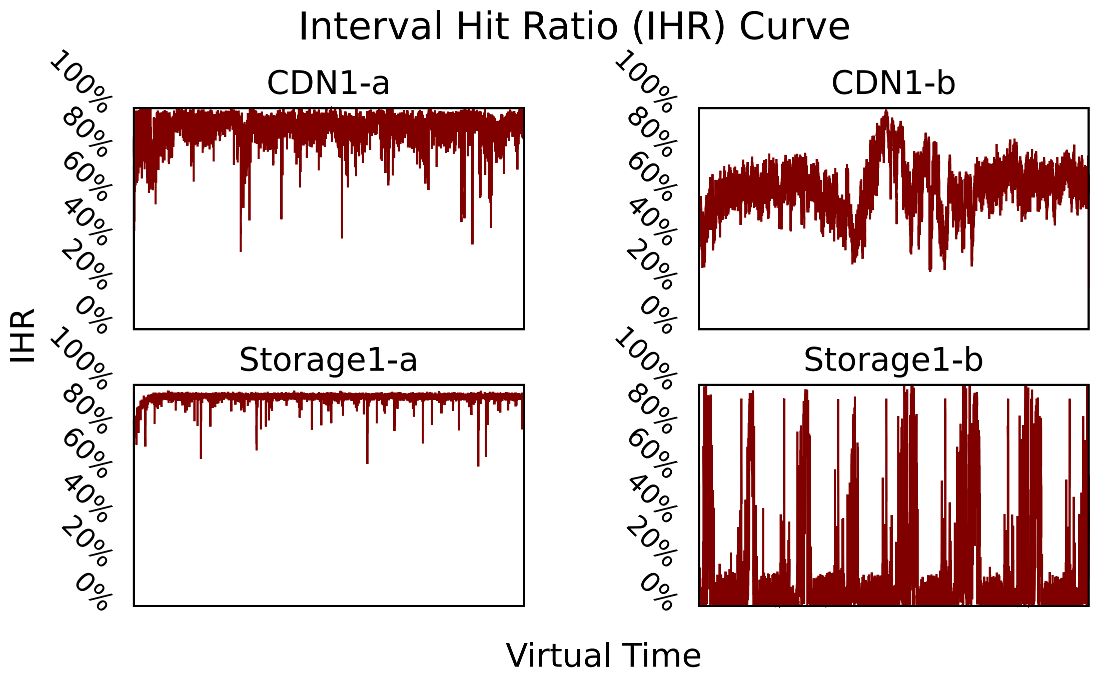


Figure 5.1: Examples of Interval Hit Ratio (IHR) Curves. Each interval is 1/1000 of the original trace length.

differently in terms of smoothness of the IHR curve. Figure 5.1 depicts the IHR curves of four workloads, two from Storage1 workloads and the other two from CDN1. We can see that the IHRs of CDN1-a and Storage1-a workloads remain high in most intervals, but CDN1-b and Storage1-b workloads are generally more fluctuated, even considering the periodic processes as a multi-day trace in the Storage1-b workload.

We note that in our analysis, we internally compute *hit rate curves*, or hit rate as a function of cache size, which can be efficiently generated through spatial sampling of the cache trace [159, 175]. Spatial sampling could be used for online computation of the warmup time if needed.

Cache warmup time. We are now ready to define cache "warmup" time using the interval hit ratio. At a high-level, we declare a particular moment in the trace as when the server comes up (with an empty cache) after downtime. We then compare the IHR of that server (a *downcache*) to that of a server that did not go down at all (an *upcache*) while processing the exact same workload. When the IHRs of these two caches are sufficiently

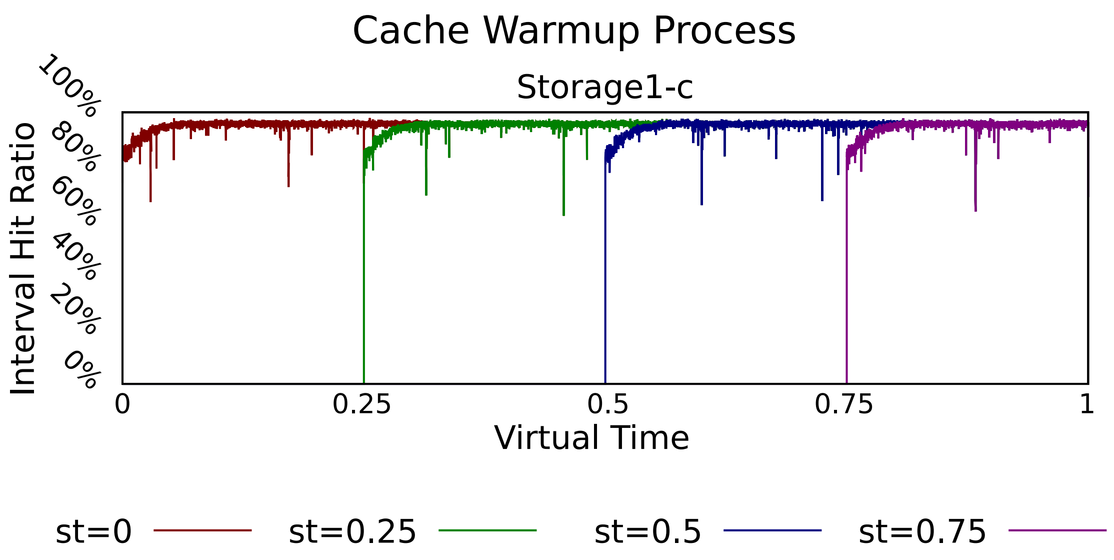


Figure 5.2: **Cache warmup process**, showing convergence of IHRs. Cache size is set to 25% of total unique items for better showing the convergence. Horizontal axis represents virtual time of the trace as a sequence of accesses. The relative start time (st) of a curve, say 0.25 means that it begins at 25% of the entire trace.

close, they have converged (as shown in Figure 5.2). This two-cache comparison overcomes the dynamical issues from above: *a cache is consider warmed up if it behaves practically like one that did not go down.*

Formally, we measure the difference in performance of a downcache that resumed operations at time st and an upcache by measuring the difference $|IHR(st, t, s) - IHR(0, t, s)|$ at time t. To capture the IHR of the upcache and downcache staying close, we define a tolerance parameter ε to express the maximum percentage difference we accept after warmup.

Definition 7. For cache size s and tolerance level $\epsilon > 0$, a downcache that recovers at time st is considered ***warmed up*** at time t if for any end time $et > t$, we have

$$|IHR(0, et, s) - IHR(st, et, s)| < \epsilon.$$

Cache warmup time therefore depends on static factors, including cache size and tolerance levels, and dynamic factors dependent on the trace-based characteristics.

Comparing cache warmup to fill up times. The definition further highlights that

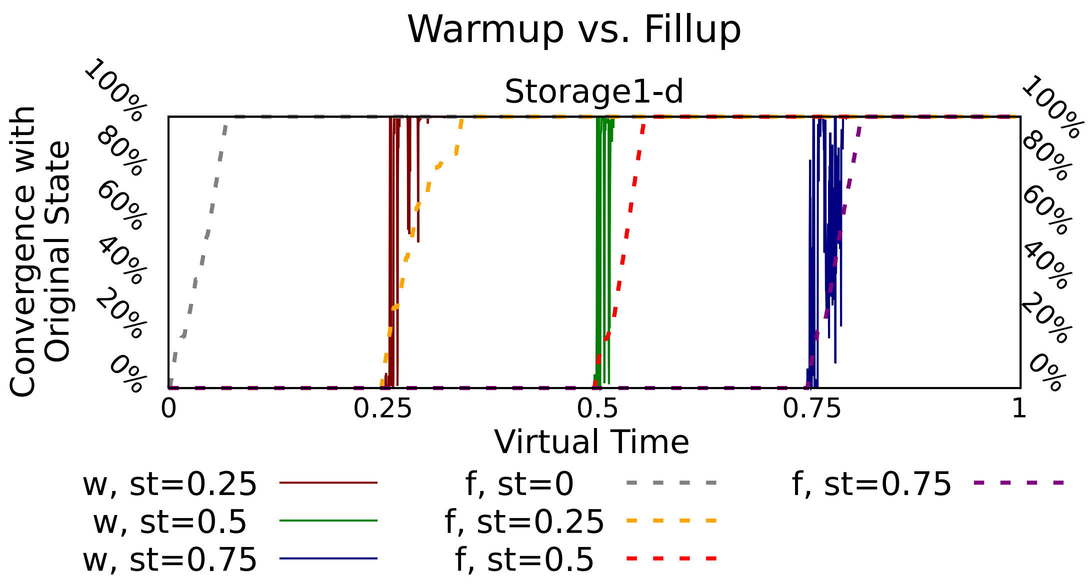


Figure 5.3: **Caches warm up faster than they fill up.** Comparing warm up and fill up with horizontal axis and four *st* as per Figure 5.2. The *f* and *w* curves respectively represent fill up and warm up. The left vertical axis shows the convergence between each downcache and the upcache; the right vertical axis shows the rate of cache capacity filled by the newly started cache.

downcache need not necessarily be filled for the cache to be considered warmed up: the rate of requests to items to which only the upcache was privy may simply be sufficiently limited that the downcache already contains the current working set and can be considered warm. An example is shown in Figure 5.3. Here we define cache is filled up when the cache capacity is fully occupied after a restart, whereas warmed up refers to the definition with $\epsilon = 1\%$. Across all our traces, the cache warms up faster than it fills up, with on average 39.1% and 36.8% for CDN1 and CDN2 workloads, and 16.6% and 23.8% for Storage1 and Storage2 workloads. These results underscore the opportunity for reconsidering cache warmup times in practice.

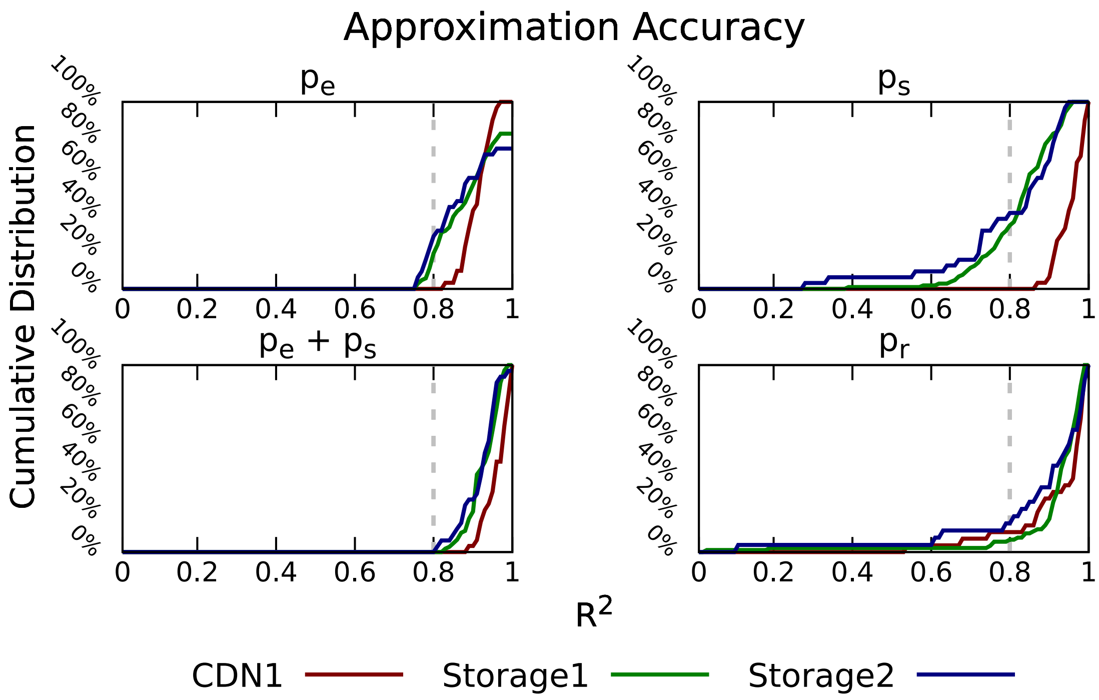


Figure 5.4: **Approximation accuracy**, evaluated with R^2 cumulative distribution for p_e, p_s, combined $p_e + p_s$, and p_r. We consider 80% as R^2 threshold of a significance (grey dotted vertical line). Results span all workloads except CDN2 which comprises only one trace.

5.3 Towards a Rule-of-Thumb Cache Warmup Time Estimation

We analyze cache warmup time on several workloads to derive a useful estimation formula. Specifically, we look for a rule of thumb that embodies the following attributes.

- **Simplicity.** Contain only a small number of parameters and as few as possible to capture the dependencies while being intuitive and practical to compute.
- **Accuracy.** Closely approximate warmup time.
- **Generality.** Yield insightful warmup time estimates for other, similar workloads.

Step 1: Relaxing Dynamic Factors. Our problem space is still large and unwieldy for operators to navigate in practice. Rather than capturing the full range of the workload's dynamic characteristics, captured by the start time st parameter, we simplify the definition to compute the maximum warmup time over all possible start times.

Definition 8. Given cache size s and tolerance degree ε, the ***warmup time*** of a cache server, $\textit{warmup-time}(s, \varepsilon)$, is the smallest t such that for every start time st and any $et > t$,

$$|\mathit{IHR}(0, et, s) - \mathit{IHR}(st, et, s)| < \varepsilon.$$

The above definition is the expression for which we will derive a rule of thumb. Simplifying is critical to minimize the number of parameters and create a practical formula.

Step 2: Approximating Static Factors. Armed with a compact definition, we can now analyze how cache size and tolerance degree affect cache warmup time on our traces. In search for a simple representation, we apply log-linear regression to model the relationship between warmup time, the cache size and the tolerance degree.

We observed that the cache warmup time has a piece-wise linear relationship to size until it reaches a plateau at larger sizes. There, the cache takes longer to warm up, but only until the working set of the trace is captured. The relationship between cache warmup time and tolerance is approximately log-linear; plotting warm-up time on a log-scale vs. tolerance on a linear scale produced a straight line. Larger tolerance degrees produce shorter warmup times as expected.

These observations suggest the following relationship, where C, p_e, and p_s are free parameters:

$$\text{warmup-time}(s, \varepsilon) = C \cdot e^{-p_e \varepsilon} \cdot s^{p_s}. \tag{5.1}$$

5.4 Results

We now consider our proposed rule of thumb and how it measures up against our desired attributes.

- **Simplicity.** The equation above says it all: there are only three free variables and one term.

Param	Value	Traces with parameter value in range			
		CDN1	CDN2	Storage1	Storage2
p_s	0-2	58.1%	100%	49%	84.2%
p_e	0.5-1.5	64.5%	100%	64.3%	78.9%
p_r	1-1.5	84%	100%	78.3%	66.7%

Table 5.1: Proportion of traces whose cache warmup times passed 80% goodness-of-fit-tests within value range for cache size parameter p_s, tolerance degree parameter p_e, and resize parameter p_r.

- **Accuracy.** To determine accuracy of the model fit we use a standard R^2 likelihood test. The R^2 distribution is shown in Figure 5.4. We consider 80% as R^2-threshold of a significance fit, so passing the test means the formula is accurate for use. As shown in the result, most of our CDN traces passed the R^2 likelihood test, together with a branch of storage traces. The accuracy is higher when considering both parameters together.

- **Generality.** Because C is a normalization parameter driven by time resolution, we investigate the ranges of parameters p_e and p_s as shown in Table 5.1. Note that here we only consider the traces that passed the test. These results meet our generality goal for the proposed method.

Traces. To derive and evaluate our rule of thumb formula, we analyze a variety of CDN and storage workloads. CDN1 includes 31 HTTP request traces from Akamai, within a single geographic region, comprised of 34 servers located in 23 logical data centers. The workload is a mixture of traffic across a wide variety of different content types, including streaming media, file downloads, and typical web content such as HTML, images, JavaScript, and CSS. CDN2 are traces from the Wikipedia CDN servers [150]. Storage1 comprises 106 week-long hypervisor-observed disk access traces in production storage systems [159]. Storage2 consists of 32 file system traces released by MSR Cambridge [125].

Implementation. We implement the warmup analysis tool on top of Mimircache [169], an open source Python cache profiler that helps to calculate IHRs of traces, making the simulation process lightweight. We use an Intel Xeon CPU E5-2670 v3 2.30GHz system for our experiments.

Applying the rule. A recent set of papers focused on offline optimal analysis of caches have shown that workload characteristics and object features are helpful for quantifying cache behaviors and further improving cache algorithms [175, 47, 71]. In line with those ideas, the warmup time of a workload can be estimated in a two-step process. First, we calculate the warmup times through an offline simulator on workloads, or a sampled workloads for efficiency. This step could be implemented through a simple API like:

$$\text{offline-results} = \mathsf{SIMULATE}(\text{workload}, \text{params})$$

Here the parameters s and ε are varied in a wide range. We then apply regression over the offline results to optimize a simple model to express warmup time over these parameters, for instance using the following API:

$$\text{warmup-time} = \mathsf{ANALYZE}(\text{offline-results}, \text{params})$$

A key problem is how to make this process efficient. We have shown that cache warmup time can be successfully estimated with a lightweight method, and that simple regression can provide sufficiently accurate results. With a rule of thumb formula, operators and designers can estimate warmup time with only a few parameters. We note that warmup time is calculated for each workload and reflects the internal characteristics and behavior of that workload, so a system operator may only need to follow this process if the workload behavior changes drastically. Also, we found that workloads that share similar behaviors also yield similar parameters for their rule of thumb formulas. For instance, two CDN workloads for the same service are likely to share the rule of thumb parameters.

Extension: Enlarging a Cache We have assumed thus far that a downcache starts off empty, which is reasonable in cases where a failure occurred since items may be stale, expired or awaiting invalidation. When a cache is resized, however, such as during cache partitioning, tenants retain existing content in the cache after it is enlarged. How would growing the capacity of a cache that already contains useful data affect the warmup time?

To define warmup time in the context of cache enlargement, we want to compare a "downcache" (to be resized) with the state of the "upcache" (fully resized). We augment the interval hit ratio definition to $IHR(\mathrm{st}, \mathrm{et}, s, \mathrm{rt}, m)$, where rt expresses the time when the cache is to be resized, and $m \geq 1$ expresses a multiple of its current cache size s. Chronologically over a request stream, a cache of size s begins at time st, its capacity is grown at time rt to a new size $m \cdot s$ and ends at time et. Now:

Definition 9. Given cache size s, size multiple m, and tolerance degree ε, assume a cache server is initially run at size s and then resized at time rt to $m \cdot s$. The resized cache server is consider to be warmed up at time t if for every resize time rt and all $\mathrm{et} > t$,

$$|\mathit{IHR}(0, \mathrm{et}, m \cdot s) - \mathit{IHR}(0, et, s, rt, m)| < \varepsilon.$$

Here, the warmup time is driven primarily by the starting size, multiple, and tolerance level. Focusing on the first two parameters that relate directly to the cache size change, we fix tolerance level to 1% in the following experiments.

A primary difference between the recovery and resizing cases is that a cache that went down will be fully able to serve content after collecting all s items, whereas $(m-1) \cdot s$ items are missing in the resized cache. We therefore consider whether there is a log-linear relationship between warm-up-time in the resized context and $(m-1) \cdot s$, and use the above methodology to obtain (for C, p_r as free parameters):

$$\text{resized-warmup-time}(s) = C \cdot ((m-1) \cdot s)^{p_r}$$

Our experiments considered $m \in \{2, 3, 4\}$ and varied s. The R^2 distribution (Figure 5.4) shows that most traces passed the R^2 likelihood tests using spatial sampling rate of 1% per trace. Our results also show that the p_r exponent parameter still concentrates differently for each workload catalog (Table 5.1).

5.5 Discussion

There are some choices that we made towards simplifying the warmup estimation process, which could be further improved. When computing interval hit ratios, for example, a practical question is how to choose the sliding window size. A large window size definitely downgrades the accuracy of reflecting dynamic workloads, while a small window size adds much overhead to the offline computation. This is a trade-off between accuracy and efficiency. In our practice, the estimation results with real workloads are not affected with varied window size, thus we arbitrarily choose a fixed window size across all our experiments.

Understanding warmup time in cache servers is a practical operation challenge in quality of service sensitive production systems. In our working scenarios, we always assume the cache spaces, either restarted or newly allocated, are empty. Cache servers are actually stateful with contents, so even with a restarted cache server, it may still contain some cache content that was loaded before. Another complex scenario is if the cache size is decreased, where the remaining cache slots are not new, but hard to evaluate which contents are left. We realize that it's challenging to estimate the remaining cache states, which should relate to some hardware researches. However, assuming an empty cache could only maximize the potential warmup time thus won't hurt the estimation for critical operation decisions.

5.6 Related Work

Warmup is an important component of the systems or frameworks of several recent systems, yet many papers either define a warmup period arbitrarily, discard the first portion of a workload [132, 150, 160], or apply a warmup mechanism without quantifying or evaluating such methods [145, 153, 166, 168]. Zhang *et al.* [177] provided a cache warmup mechanism based on cache recency and is closest to our work. Their method does not consider workload dynamics. To the best of our knowledge, our work is the first to provide a practical method for estimating cache warmup time, and derive a simple expression for engineers and scientists to use.

5.7 Takeaway

There are many scenarios where operators of large distributed caches must implicitly or explicitly reason about warmup time of a cache server. Here, we derive a novel rule of thumb equation based on empirical results on a variety of real-world traces that demonstrates a power-law relationship between warmup time and cache size, coupled with an inverse exponential discount based on the desired tolerance level.

We build an offline simulator to fit free parameters of the formulas, which is shown to be concentrated within each workload category, to provide a useful expression for back-of-the-envelope calculations for the expected warmup time of cache servers without unduly impacting end-user clients or storage servers with miss penalties.

By this chapter, we have provided a practical example on how offline optimal analysis can serve towards better building distributed systems.

Chapter 6

Conclusion

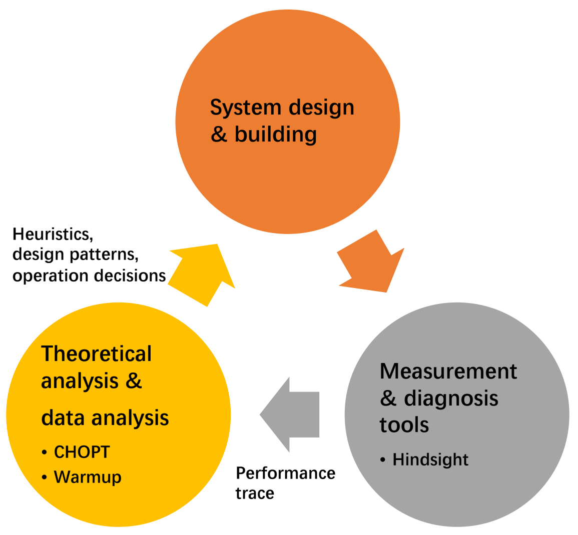

System design & building
Heuristics, design patterns, operation decisions
Theoretical analysis & data analysis
• CHOPT
• Warmup
Performance trace
Measurement & diagnosis tools
• Hindsight

Figure 6.1: Thesis conclusion: developing distributed systems.

This thesis argues the importance of measuring and analyzing performance problems in distributed systems towards better building systems with high performance. Furthermore, the thesis introduces practical methods that I developed towards the problem. Figure 6.1 concludes the major works introduced in this thesis and their relationships.

For measurement methods, I developed Hindsight as a lightweight tracing tool for collecting edge-case performance problems in distributed systems. For analysis methods, I developed CHOPT as an optimal placement policy for analyzing optimal performance patterns in modern memory hierarchy. I also introduced practical solutions to solve real-world cache warmup problems through offline analysis.

As also discussed in chapter 1, this thesis serves as filling in some missing parts towards practically understanding distributed system performance problems. Developing measurement and diagnosis tools effectively provides performance traces, which are necessary for any future analysis. Applying theoretical analysis and data analysis, though they are usually offline based, can provide heuristics for system design and operations. Especially in performance critical systems like memory and cache systems, those are already proved useful towards online system design.

Both my thesis goal and my individual research works discussed in this thesis propose the idea to apply theoretical analysis to system design and building. I believe this is trending towards today's distributed systems development. Compared with a decade before, building a large-scale system to provide specific functions is not as challenging as it was. Research and industry focuses have switched to either providing better performance or improving the system development process. Such improvement could be through automating some critical processes like testing and debugging by building practical tools. Besides, as the demand for performance keeps growing, new systems are by default required to operate speedy online feedback. In those scenarios, theoretical methods turn out to be helpful by simplifying some processes with concluded models. Also as discussed in many sections, theoretical analysis can also dig into the performance and find out indiscoverable problems, like some complex problem patterns. Overall, as system developments are becoming more intelligent, theoretical analysis and data analysis should play more important role in the near future.

6.1 Future Directions

Improving performance problems in distributed systems is a significant problem and still requires a lot of effort. This thesis aims at not only providing practical solutions to fill in some missing slots for the state-of-the-art solutions, but also opening some future directions towards the long term goal.

Tracing with broader provenance towards root cause analysis. With tracing systems like Hindsight, system developers can efficiently capture request traces across to figure out what happened to problematic requests. However, root cause evidence could exist out of the scope of captured requests, like poor queuing or scheduling mechanisms. Most traditional tracing methods, that rely on request sampling, are even limited to zoom in on these information because the probability of having all queued requests in the samples is almost 0. Hindsight provides the ability to overcome this limitation, and the major challenge comes to efficiently capture more information. A meaningful direction is to look at this problem as an extension of Hindsight work. To solve it, tracing systems should not only look at the application level but deeper to the host system/kernel level, which might be supported by low level tracing tools like eBPF or Intel Processor Tracing(IPT). Then with the additional data load, tracing should dynamically collect useful but enough information and analyze potential provenances in runtime. If successful, this method can provide a very broad scope of system performance problems which is fundamentally helpful for root cause analysis.

Performance aware caching system. Cache algorithms and strategies are usually designed to theoretically optimize the cache hit ratio, which ignores the real cost effective to the whole system in practice. This would raise the risk especially in complex cache systems, which typically have multiple layers and shared memory spaces, that cache may not perform as designed and expected. It's meaningful to bridge the gap between theory and practice by building performance aware cache systems. Such a cache system should provide intelligent runtime cache decisions with the knowledge of real time cache states and the exact performance impact of a potential cache operation. Another key challenge is, since cache itself is designed for high performance, such a method must also be efficient. This could be potentially supported with my tracing experience which also tackles down data collection and processing problems at low level. If successful, this proposed direction can provide a fundamental improvement on practical cache systems.

Resource disaggregation. Distributed caching systems are based on sharding, which leads to hotspots for highly skewed workloads. Disaggregation can make under-utilized resources available to a hot-spot machine. It's shown by literature that CPU and memory utilization are commonly reported as lower than 50%. Another example is multi-tenant shared memory, where dynamically adjusting reserved memory space for tenants can help improve overall performance. This requires a two-fold solution. First, we need to practically monitor performances in real time. I propose to build tracing systems to solve this. Second, we need to build analysis tools to make disaggregation decisions. Many recent works have also shown the power to apply machine learning techniques to this problem, where ML could serve as practical estimators or simplifying some time consuming processes.

Bibliography

[1] Akamai Online Retail Performance Report: Milliseconds Are Critical. Retrieved Jan 2021 from https://www.akamai.com/uk/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp.

[2] AWS Solutions Implementation overview. Retrieved Oct 2021 from https://aws.amazon.com/cn/solutions/implementations/aws-perspective/5.

[3] Scaling the Facebook data warehouse to 300 PB. Retrieved Jan 2021 from https://engineering.fb.com/2014/04/10/core-data/scaling-the-facebook-data-warehouse-to-300-pb/.

[4] Memcached. Retrieved Aug 2019 from http://memcached.org, 2009.

[5] HDFS-3751: DN should log warnings for lengthy disk IOs. Retrieved May 2021 from https://issues.apache.org/jira/browse/HDFS-3751, 2014.

[6] HBASE-8228: Investigate time taken to snapshot memstore. Retrieved May 2021 from https://issues.apache.org/jira/browse/HDFS-8228, 2015.

[7] HDFS-9184: Logging HDFS operation's caller context into audit logs). Retrieved December 2020 from https://issues.apache.org/jira/browse/HDFS-9184, 2015.

[8] HBASE-8744: Enable HBase to log the entire latency profile for HDFS packets resulting in slow writes. Retrieved May 2021 from https://issues.apache.org/jira/browse/HDFS-8744, 2016.

[9] HDFS-11461: DataNode Disk Outlier Detection. Retrieved May 2021 from https://issues.apache.org/jira/browse/HDFS-11461, 2017.

[10] Jaeger Issue 365: Adaptive Sampling. Retrieved December 2020 from https://github.com/jaegertracing/jaeger/issues/365, 2017.

[11] Zipkin Issue 1835: Zipkin Collector Sampling based on Traces/End To End Transaction & not based on individual spans. Retrieved December 2020 from https://github.com/openzipkin/zipkin/issues/1835, 2017.

[12] Intel Optane DC Persistent Memory Operating Modes. Retrieved Aug 2019 from https://itpeernetwork.intel.com/intel-optane-dc-persistent-memory-operating-modes, 2018.

[13] HDFS-6110: adding more slow action log in critical write path. Retrieved May 2021 from https://issues.apache.org/jira/browse/HDFS-6110, 2018.

[14] Intel 64 and IA-32 Architectures Optimization Reference Manual. Retrieved Aug 2019 from https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-optimization-reference-manual, 2019.

[15] Jaeger Issue 1861: Delayed Sampling. Retrieved December 2020 from https://github.com/jaegertracing/jaeger/issues/1861, 2019.

[16] Jaeger Issue 1954: Average response time, throughput graphs. Retrieved December 2020 from https://github.com/jaegertracing/jaeger/issues/1954, 2019.

[17] OpenTelemetry Specification Issue 307: Allow samplers to be called during different moments in the Span lifetime. Retrieved December 2020 from https://github.com/open-telemetry/opentelemetry-specification/issues/307, 2019.

[18] Measure, Then Build. Retrieved Oct 2021 from https://www.usenix.org/conference/atc19/presentation/keynote, 2019.

[19] Jaeger: Open Source, End-to-End Distributed Tracing. Retrieved December 2020 from https://www.jaegertracing.io/, 2020.

[20] Jaeger-Analytics Issue 42: Anomaly Metrics. Retrieved December 2020 from https://github.com/jaegertracing/jaeger-analytics-java/issues/42, 2020.

[21] OpenTelemetry: An Observability Framework for Cloud-Native Software. Retrieved December 2020 from http://opentelemetry.io/, 2020.

[22] OpenTracing: Vendor-Neutral APIs and Instrumentation for Distributed Tracing. Retrieved December 2020 from http://opentracing.io/, 2020.

[23] OpenTelemetry Enhancement Proposal 115: Allow Additional Sampling Hooks. Retrieved December 2020 from https://github.com/open-telemetry/oteps/pull/115, 2020.

[24] Zipkin: A Distributed Tracing System. Retrieved December 2020 from http://zipkin.io/, 2020.

[25] ACCUMULO-3725: Majc trace tacked onto minc trace. Retrieved May 2021 from https://issues.apache.org/jira/browse/ACCUMULO-3725, 2021.

[26] CASSANDRA-7644: Tracing does not log commitlog/memtable ops when the coordinator is a replica. Retrieved May 2021 from https://issues.apache.org/jira/browse/CASSANDRA-7644, 2021.

[27] CASSANDRA-7657: Tracing doesn't finalize under load when it should. Retrieved May 2021 from https://issues.apache.org/jira/browse/CASSANDRA-7657, 2021.

[28] Google's datacenter volume. Retrieved Oct 2021 from https://what-if.xkcd.com/63/, 2021.

[29] HBASE-13077: BoundedCompletionService doesn't pass trace info to server. Retrieved May 2021 from https://issues.apache.org/jira/browse/HBASE-13077, 2021.

[30] HBASE-15880: RpcClientImpl#tracedWriteRequest incorrectly closes HTrace span. Retrieved May 2021 from https://issues.apache.org/jira/browse/HBASE-15880, 2021.

[31] HTRACE-330: Add to Tracer, TRACE-level logging of push and pop of contexts to aid debugging "Can't close TraceScope..". Retrieved May 2021 from `https://issues.apache.org/jira/browse/HTRACE-330`, 2021.

[32] Spring Cloud Sleuth 410: Trace ID problem when using Spring ThreadPoolTaskExecutor. Retrieved May 2021 from `https://github.com/spring-cloud/spring-cloud-sleuth/issues/410`, 2021.

[33] Neha Agarwal and Thomas F Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 631–644. ACM, 2017.

[34] Susanne Albers, Sanjeev Arora, and Sanjeev Khanna. Page replacement for general caching problems. In *SODA*, volume 99, pages 31–40. Citeseer, 1999.

[35] Qasim Ali and Praveen Yedlapalli. Persistent memory performance in vsphere 6.7. 2019.

[36] Peter Alvaro, Joshua Rosen, and Joseph M Hellerstein. Lineage-driven fault injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 331–346, 2015.

[37] Mohamed Arafa, Bahaa Fahim, Sailesh Kottapalli, Akhilesh Kumar, Lily P Looi, Sreenivas Mandava, Andy Rudoff, Ian M Steiner, Bob Valentine, Geetha Vedaraman, et al. Cascade lake: Next generation intel xeon scalable processor. *IEEE Micro*, 39 (2):29–36, 2019.

[38] Dan Ardelean, Amer Diwan, and Chandra Erdman. Performance analysis of cloud applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, pages 405–417, 2018.

[39] Narayanan Arunachalam. Zipkin Secondary Sampling. Retrieved December 2020 from `https://github.com/openzipkin-contrib/zipkin-secondary-sampling`, 2019.

[40] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[41] Amotz Bar-Noy, Reuven Bar-Yehuda, Ari Freund, Joseph Naor, and Baruch Schieber. A unified approach to approximating resource allocation and scheduling. *Journal of the ACM (JACM)*, 48(5):1069–1090, 2001.

[42] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *21st IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–75. IEEE, 2015.

[43] Nathan Beckmann and Daniel Sanchez. Maximizing cache performance under uncertainty. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 109–120. IEEE, 2017.

[44] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, 2018.

[45] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[46] Daniel S Berger. Design and analysis of adaptive caching techniques for internet content delivery. 2018.

[47] Daniel S Berger. Towards lightweight and robust machine learning for cdn caching. In *HotNets*, pages 134–140, 2018.

[48] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 483–498, 2017.

[49] Daniel S Berger, Nathan Beckmann, and Mor Harchol-Balter. Practical bounds on optimal caching with variable object sizes. *Proceedings of the ACM Measurement and Analysis of Computing Systems*, 2(2):32, 2018.

[50] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th International Conference on Parallel Architectures and Compilation Techniques (PACT 08)*, pages 72–81. ACM, 2008.

[51] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. mPart: miss-ratio curve guided partitioning in key-value stores. In *ACM SIGPLAN Notices*, volume 53, pages 84–95. ACM, 2018.

[52] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 893–907, 2018.

[53] Zhen Cao, Geoff Kuenning, and Erez Zadok. Carver: Finding important parameters for storage system tuning. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 43–57, 2020.

[54] Martin C Carlisle and Errol L Lloyd. On the k-coloring of intervals. In *International Conference on Computing and Information*, pages 90–101. Springer, 1991.

[55] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 115–128, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341936. doi: 10.1145/2934872.2934910. URL `https://doi.org/10.1145/2934872.2934910`.

[56] Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. One primitive to diagnose them all: Architectural support for internet diagnostics. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 374–388, 2017.

[57] Mike Y Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. In *1st USENIX Symposium on Networked Systems Design & Implementation (NSDI'04)*, pages 23–23, 2004.

[58] 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 *USENIX Annual Technical Conference (ATC 16)*, pages 379–392, 2016.

[59] Sangyeun Cho and Hyunjin Lee. Flip-n-write: A simple deterministic technique to improve pram write performance, energy and endurance. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 347–357, 2009.

[60] Gregory Chockler, Guy Laden, and Ymir Vigfusson. Data caching as a cloud service. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware (LADIS 10)*, pages 18–21. ACM, 2010.

[61] Gregory Chockler, Guy Laden, and Ymir Vigfusson. Design and implementation of caching services in the cloud. *IBM Journal of Research and Development*, 55(6):9–1, 2011.

[62] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, 2016.

[63] Asaf Cidon, Daniel Rushton, Stephen M Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017.

[64] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. Rept: Reverse debugging of failures in deployed software. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 17–32, 2018.

[65] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013.

[66] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 1–14, 2019.

[67] Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):35, 2017.

[68] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive software cache management. In *19th International Middleware Conference (MIDDLEWARE 18)*, pages 94–106. ACM, 2018.

[69] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM footprint with NVM in Facebook. In *13th EuroSys Conference*, page 42. ACM, 2018.

[70] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using non-volatile memory for storing deep learning models. *arXiv preprint arXiv:1811.05922*, 2018.

[71] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *NSDI*, pages 65–78, 2019.

[72] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS)*, 30(4):1–35, 2012.

[73] Martin Farach-Colton and Vincenzo Liberatore. On local register allocation. *Journal of Algorithms*, 37(1):37–65, 2000.

[74] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. Reducing web latency: the virtue of gentle aggression. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 159–170, 2013.

[75] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. X-trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI'07)*, 2007.

[76] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, pages 3–18, 2019.

[77] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, pages 19–33, 2019.

[78] Jayesh Gaur, Mainak Chaudhuri, and Sreenivas Subramoney. Bypass and insertion algorithms for exclusive last-level caches. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 81–92. ACM, 2011.

[79] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *NSDI*, volume 7, pages 285–298, 2007.

[80] Binny S Gill. On multi-level exclusive caching: offline optimality and why promotions are better than demotions. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, page 4. USENIX Association, 2008.

[81] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, 2016.

[82] Wajih Ul Hassan, Adam Bates, and Daniel Marino. Tactical provenance analysis for endpoint detection and response systems. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1172–1189, 2020. doi: 10.1109/SP40000.2020.00096.

[83] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, pages 1–17, 2015.

[84] Yu-Ju Hong and Mithuna Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th annual Symposium on Cloud Computing (SOCC 13)*, page 13. ACM, 2013.

[85] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic modeling of data eviction in cache. In *2016 USENIX Annual Technical Conference (ATC 16)*, pages 351–364, 2016.

[86] Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An Analysis of Facebook Photo Caching. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*, 2013.

[87] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A Freedman, Ken Birman, and Robbert van Renesse. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets 14)*. ACM, 2014.

[88] Intel Corporation. *Intel 64 and IA-32 architectures software developer's manual*, volume 3 (3A, 3B, 3C & 3D): System Programming Guide. Intel, 2016.

[89] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the Intel Optane DC persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[90] Jaeger. Jaeger 1.26 Documentation: Sampling. Retrieved in Sep 2021 from `https://www.jaegertracing.io/docs/1.26/sampling`, 2021.

[91] Akanksha Jain and Calvin Lin. Back to the future: leveraging belady's algorithm for improved cache replacement. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 78–89. IEEE, 2016.

[92] Akanksha Jain and Calvin Lin. Rethinking belady's algorithm to accommodate prefetching. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 110–123. IEEE, 2018.

[93] Chris Jones, John Wilkes, Niall Murphy, and Cody Smith. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016. `https://landing.google.com/sre/sre-book/chapters/service-level-objectives/`.

[94] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)*, pages 34–50, 2017.

[95] Pravein Govindan Kannan, Nishant Budhdev, Raj Joshi, and Mun Choon Chan. Debugging transient faults in data centers using synchronized network-wide packet histories. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 253–268. USENIX Association, April 2021. ISBN 978-1-939133-21-2. URL `https://www.usenix.org/conference/nsdi21/presentation/kannan`.

[96] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. HeteroOS – OS design for heterogeneous memory management in datacenter. In *2017 ACM/IEEE*

44th Annual International Symposium on Computer Architecture (ISCA), pages 521–534. IEEE, 2017.

[97] Richard E. Kessler, Mark D Hill, and David A Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers*, 43(6):664–675, 1994.

[98] Kunal Korgaonkar, Ishwar Bhati, Huichu Liu, Jayesh Gaur, Sasikanth Manipatruni, Sreenivas Subramoney, Tanay Karnik, Steven Swanson, Ian Young, and Hong Wang. Density tradeoffs of non-volatile memory as a replacement for sram based last level cache. In *45th Annual International Symposium on Computer Architecture (ISCA 18)*, pages 315–327. IEEE Press, 2018.

[99] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast nvm storage with udepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 1–15, 2019.

[100] Pedro Las-Casas, Jonathan Mace, Dorgival Guedes, and Rodrigo Fonseca. Weighted Sampling of Execution Traces: Capturing More Needles and Less Hay. In *9th ACM Symposium on Cloud Computing (SOCC '18)*, 2018.

[101] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. Sifter: Scalable sampling for distributed traces, without feature engineering. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'19)*, pages 312–324, 2019.

[102] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[103] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[104] Pengcheng Li, Colin Pronovost, William Wilson, Benjamin Tait, Jie Zhou, Chen Ding, and John Criswell. Beating OPT with statistical clairvoyance and variable size

caching. In *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 19)*, pages 243–256. ACM, 2019.

[105] Yang Li, Saugata Ghose, Jongmoo Choi, Jin Sun, Hui Wang, and Onur Mutlu. Utility-based hybrid memory management. In *2017 IEEE International Conference on Cluster Computing (CLUSTER 17)*, pages 152–165. IEEE, 2017.

[106] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. Metis: Robustly tuning tail latencies of cloud systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 981–992, 2018.

[107] Lightstep. Opentelemetry: Best practices on sampling. Retrieved December 2020 from `https://opentelemetry.lightstep.com/best-practices/sampling/`, 2020.

[108] Lightstep. OpenTelemetry Issue #407: Tail-Based Sampling Scalability Issues. Retrieved December 2020 from `https://github.com/open-telemetry/opentelemetry-collector/issues/407`, 2020.

[109] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: measuring and understanding consistency at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP 15)*, pages 295–310. ACM, 2015.

[110] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.

[111] Liang Luo, Suman Nath, Lenin Ravindranath Sivalingam, Madan Musuvathi, and Luis Ceze. Troubleshooting transiently-recurring errors in production systems with blame-proportional logging. In *2018 USENIX Annual Technical Conference (USENIX ATC'18)*, pages 321–334, 2018.

[112] Dan Luu. A simple way to get more value from tracing. Retrieved December 2020 from `https://danluu.com/tracing-analytics/`, 2020.

[113] Jonathan Mace and Rodrigo Fonseca. Universal context propagation for distributed system instrumentation. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys'18)*, pages 1–18, 2018.

[114] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted Resource Management in Multi-Tenant Distributed Systems. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, 2015.

[115] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *25th ACM Symposium on Operating Systems Principles (SOSP'15)*, 2015.

[116] Jasmina Malicevic, Subramanya Dulloor, Narayanan Sundaram, Nadathur Satish, Jeff Jackson, and Willy Zwaenepoel. Exploiting NVM in large-scale graph analytics. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, page 2. ACM, 2015.

[117] Gideon Mann, Mark Sandler, Darja Krushevskaja, Sudipto Guha, and Eyal Even-Dar. Modeling the parallel execution of black-box services. In *Proceedings of USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'11)*, 2011.

[118] 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.

[119] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[120] Nimrod Megiddo and Dharmendra S Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[121] Pierre Michaud. Some mathematical facts about optimal cache replacement. *ACM Transactions on Architecture and Code Optimization*, 13(4), 2016.

[122] Pulkit A Misra, María F Borge, Íñigo Goiri, Alvin R Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. Managing tail latency in datacenter-scale file systems under production constraints. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

[123] Sparsh Mittal. A survey of cache bypassing techniques. *Journal of Low Power Electronics and Applications*, 6(2):5, 2016.

[124] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the Graph 500. *Cray Users Group (CUG)*, 19:45–74, 2010.

[125] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008.

[126] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, volume 13, pages 385–398, 2013.

[127] Krzysztof Ostrowski, Gideon Mann, and Mark Sandler. Diagnosing latency in multi-tier black-box services. In *4th International Workshop on Large-Scale Distributed Systems and Middleware (LADIS'11)*, 2011.

[128] João Paiva and Luís Rodrigues. On data placement in distributed systems. *ACM SIGOPS Operating Systems Review*, 49(1):126–130, 2015.

[129] Maulik Pandey. Building Netflix's Distributed Tracing Infrastructure. Retrieved December 2020 from `https://netflixtechblog.com/building-netflixs-distributed-tracing-infrastructure-bb856c319304`, 2019.

[130] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, and Rebecca Isaacs. *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices.* O'Reilly Media, 2020.

[131] Gotze Philipp, Baumann Stephan, and Sattler Kai-Uwe. An nvm-aware storage layout for analytical workloads. In *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*, pages 110–115. IEEE, 2018.

[132] Dai Qin, Angela Demke Brown, and Ashvin Goel. Reliable writeback for client-side flash caches. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 451–462, 2014.

[133] Hanfeng Qin and Hai Jin. Warstack: Improving llc replacement for nvm with a writeback-aware reuse stack. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 233–236. IEEE, 2017.

[134] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 423–432. IEEE, 2006.

[135] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News*, 35(2):381–391, 2007.

[136] Moinuddin K Qureshi, Michele M Franceschini, and Luis A Lastras-Montano. Improving read performance of phase change memories via write cancellation and write pausing. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–11. IEEE, 2010.

[137] Patrick Reynolds, Charles Edwin Killian, Janet L Wiener, Jeffrey C Mogul, Mehul A Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, volume 6, pages 9–9, 2006.

[138] Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC 14)*, pages 1–14. ACM, 2014.

[139] Frederic Sala, Ryan Gabrys, and Lara Dolecek. Dynamic threshold schemes for multi-level non-volatile memories. *IEEE Transactions on Communications*, 61(7):2624–2634, 2013.

[140] Raja R Sambasivan, Alice X Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R Ganger. Diagnosing performance changes by comparing request flows. In *8th USENIX Symposium on Networked Systems Design & Implementation (NSDI'11)*, volume 5, pages 1–1, 2011.

[141] Raja R Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H Sigelman, Rodrigo Fonseca, and Gregory R Ganger. Principled workflow-centric tracing of distributed systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 401–414, 2016.

[142] Stefan Saroiu, Krishna P Gummadi, Richard J Dunn, Steven D Gribble, and Henry M Levy. An analysis of internet content delivery systems. *ACM SIGOPS Operating Systems Review*, 36(SI):315–327, 2002.

[143] M Zubair Shafiq, Amir R Khakpour, and Alex X Liu. Characterizing caching workload of a large commercial content delivery network. In *Proceedings of the 35th Annual IEEE International Conference on Computer Communications (INFOCOMM 16)*, pages 1–9. IEEE, 2016.

[144] Muhammad Zubair Shafiq, Alex X Liu, and Amir R Khakpour. Revisiting caching in content delivery networks. In *ACM SIGMETRICS Performance Evaluation Review*, volume 42, pages 567–568. ACM, 2014.

[145] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. Didacache: an integration of device and application for flash-based key-value caching. *ACM Transactions on Storage (TOS)*, 14(3):1–32, 2018.

[146] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 413–425. ACM, 2019.

[147] Yuri Shkuro. *Mastering Distributed Tracing.* Packt Publishing, Feb 2019.

[148] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[149] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010. URL `https://research.google.com/archive/papers/dapper-2010-1.pdf`.

[150] Zhenyu Song, Daniel S Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, 2020.

[151] Kun Suo, Jia Rao, Luwei Cheng, and Francis CM Lau. Time capsule: Tracing packet latency across different layers in virtualized systems. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 1–9, 2016.

[152] Steven Swanson. Redesigning file systems for nonvolatile main memory. *IEEE Micro*, 39(1):62–64, 2019.

[153] Jianzhe Tai, Deng Liu, Zhengyu Yang, Xiaoyun Zhu, Jack Lo, and Ningfang Mi. Improving flash resource utilization at minimal management cost in virtualized flash-based storage systems. *IEEE Transactions on Cloud Computing*, 5(3):537–549, 2015.

[154] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. ibtune: individualized buffer tuning for large-scale cloud databases. *Proceedings of the VLDB Endowment*, 12(10):1221–1234, 2019.

[155] Linpeng Tang, Qi Huang, Amit Puntambekar, Ymir Vigfusson, Wyatt Lloyd, and Kai Li. Popularity prediction of Facebook videos for higher quality streaming. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017.

[156] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. Stardust: Tracking Activity in a Distributed Storage System. In *2006 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '06)*, 2006.

[157] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*, Bordeaux, France, 2015.

[158] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *USENIX Annual Technical Conference (ATC 17)*, pages 487–498, 2017.

[159] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC Construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, 2015.

[160] Kefei Wang and Feng Chen. Cascade mapping: Optimizing memory efficiency for flash-based key-value caching. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC 18)*, pages 464–476, 2018.

[161] Zhe Wang, Shuchang Shan, Ting Cao, Junli Gu, Yi Xu, Shuai Mu, Yuan Xie, and Daniel A Jiménez. Wade: Writeback-aware dynamic cache management for nvm-based main memory system. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):51, 2013.

[162] Kevin D Wayne. A polynomial combinatorial algorithm for generalized minimum cost flow. *Mathematics of Operations Research*, 27(3):445–459, 2002.

[163] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, and Andrew Warfield. Characterizing storage workloads with counter stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 335–349, 2014.

[164] Theodore M Wong and John Wilkes. My cache or yours?: Making storage more exclusive. In *USENIX Annual Technical Conference, General Track*, pages 161–175, 2002.

[165] Fengguang Wu. PMEM NUMA node and hotness accounting/migration. In *Linux Kernel Mailing List Archive*, 2018. `https://lkml.org/lkml/2018/12/26/138`, Last accessed on 08-08-2019.

[166] Xingbo Wu, Fan Ni, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, Zili Shao, and Song Jiang. Nvmcached: An nvm-based key-value cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems (ApSys 16)*, pages 1–7, 2016.

[167] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. Zeno: diagnosing performance problems with temporal provenance. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, pages 395–420, 2019.

[168] Ji Xue, Feng Yan, Alma Riska, and Evgenia Smirni. Storage workload isolation via tier warming: How models can help. In *Proeedings of the 11th International Conference on Autonomic Computing (ICAC 14)*, pages 1–11, 2014.

[169] Juncheng Yang. Mimircache. `http://mimircache.info/`, May 2018. (Accessed May 11, 2020).

[170] Stephen Yang, Seo Jin Park, and John Ousterhout. NanoLog: a nanosecond scale logging system. In *2018 USENIX Annual Technical Conference (ATC'18)*, pages 335–350, 2018.

[171] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Steve Sporea, Andrei Toma, and Sarah Sajedi. Log4perf: Suggesting logging locations for web-based systems' performance monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE'18)*, pages 127–138, 2018.

[172] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial

and open source systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, pages 159–172, 2011.

[173] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*, 2011.

[174] Jianhui Yue and Yifeng Zhu. Accelerating write by exploiting pcm asymmetries. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, pages 282–293, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-1-4673-5585-8. doi: 10.1109/HPCA.2013.6522326. URL `http://dx.doi.org/10.1109/HPCA.2013.6522326`.

[175] Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. Optimal data placement for heterogeneous cache, memory, and storage systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(1):1–27, 2020.

[176] Lei Zhang, Juncheng Yang, Anna Blasiak, Mike McCall, and Ymir Vigfusson. When is the cache warm? manufacturing a rule of thumb. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[177] Yiying Zhang, Gokul Soundararajan, Mark W Storer, Lakshmi N Bairavasundaram, Sethuraman Subbiah, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Warming up storage-level caches with Bonfire. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 59–72, 2013.

[178] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 456–468. IEEE, 2016.

[179] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. Log20: Fully automated optimal placement of log printing statements under specified

overhead threshold. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)*, pages 565–581, 2017.

[180] Yingjie Zhao, Nong Xiao, and Fang Liu. Red: An efficient replacement algorithm based on resident distance for exclusive storage caches. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6. IEEE, 2010.

[181] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 615–626, 2010.

[182] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure network provenance. In *Proceedings of the twenty-third ACM symposium on operating systems principles*, pages 295–310, 2011.

[183] Wenchao Zhou, Suyog Mapara, Yiqing Ren, Yang Li, Andreas Haeberlen, Zachary Ives, Boon Thau Loo, and Micah Sherr. Distributed time-aware provenance. *Proceedings of the VLDB Endowment*, 6(2):49–60, 2012.

[184] Pengfei Zuo, Yu Hua, Ming Zhao, Wen Zhou, and Yuncheng Guo. Improving the performance and endurance of encrypted non-volatile main memory through deduplicating writes. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 442–454. IEEE, 2018.