Optimal Data Placement for Heterogeneous Cache, Memory, and Storage Systems

LEI ZHANG, Emory University, USA
REZA KARIMI, Emory University, USA
IRFAN AHMAD, Magnition, USA
YMIR VIGFUSSON, Emory University, USA

New memory technologies are blurring the previously distinctive performance characteristics of adjacent layers in the memory hierarchy. No longer are such layers orders of magnitude different in request latency or capacity. Beyond the traditional single-layer view of caching, we now must re-cast the problem as a data placement challenge: which data should be cached in faster memory if it could instead be served directly from slower memory?

We present Chopt, an offline algorithm for data placement across multiple tiers of memory with asymmetric read and write costs. We show that Chopt is optimal and can therefore serve as the upper bound of performance gain for any data placement algorithm. We also demonstrate an approximation of Chopt which makes its execution time for long traces practical using spatial sampling of requests incurring a small 0.2% average error on representative workloads at a sampling ratio of 1%. Our evaluation of Chopt on more than 30 production traces and benchmarks shows that optimal data placement decisions could improve average request latency by 8.2%-44.8% when compared with the long-established gold standard: Belady and Mattson's offline, evict-farthest-in-the-future optimal algorithms. Our results identify substantial improvement opportunities for future online memory management research.

CCS Concepts: • **Theory of computation → Caching and paging algorithms**; • **General and reference** → *Metrics*; *Performance*; • **Software and its engineering → Memory management**; • **Information systems → Hierarchical storage management**.

Additional Key Words and Phrases: Data Placement, Cost-aware Cache Replacement, Memory Hierarchy, Offline Optimal Analysis, Spatial Sampling, Non-Volatile Memory

ACM Reference Format:
Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. 2020. Optimal Data Placement for Heterogeneous Cache, Memory, and Storage Systems. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 1, Article 6 (March 2020), 27 pages. https://doi.org/10.1145/3379472

1 INTRODUCTION

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

Authors' addresses: Lei Zhang, Emory University, Atlanta, GA, USA, lei.zhang@emory.edu; Reza Karimi, Emory University, Atlanta, GA, USA, rkarimi@emory.edu; Irfan Ahmad, Magnition, Redwood City, CA, USA, mr.irfan@gmail.com; Ymir Vigfusson, Emory University, Atlanta, GA, USA, ymir@mathcs.emory.edu.

2476-1249/2020/3-ART6 $15.00
https://doi.org/10.1145/3379472

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 [24, 33]) incurring $2 - -5\times$ the read latency and $4 - -10\times$ the write latency of DRAM (Table 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.

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, our paper 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* [47] or *cache admission* [15] techniques (Figure 1), discussed further below.

A2 (**Performance-Asymmetry**) Next, read (load) and write (store) operations can have asymmetric performance characteristics (Table 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.

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 [3, 22–24, 36, 38, 41, 47]. 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 [14], accounting for cache write-back policies [8], and enabling caches to dynamically adjust their capacity [40].

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 [9, 58, 59].

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),

	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 1. *Memory Hierarchy Characteristics of DDR4 DRAM, NVM (Intel Optane DC Persistent Memory), NVMe block device (Intel Optane SSD DC), and TLC flash device [5, 24, 33]*

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). 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.

Our key contributions can be summarized as follows.

- We frame the challenges of data placement in modern memory hierarchies in a generalized paging model outside of traditional assumptions.
- We design Chopt, an offline data placement algorithm for providing optimal placement decisions as the upper bound of performance gain for any data placement algorithm.
- We apply spatial sampling to enable Chopt to support long traces efficiently.
- We show analytically that spatial sampling gives a high-fidelity approximation.
- We present trace-driven simulation results in context of a two-layer memory hierarchy for Chopt relative to Belady's MIN. Our results show an opportunity to improve latency performance ranging between 8.2% and 44.8% on average versus the MIN clairvoyant algorithm. We also show how our revisited assumptions **A1** and **A2** contribute to these improvements.
- We evaluate the performance of applying spatial sampling on Chopt, showing an error of only 0.2% in average latency at 1% sampling ratio on the PARSEC benchmarks [16].

2 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 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.

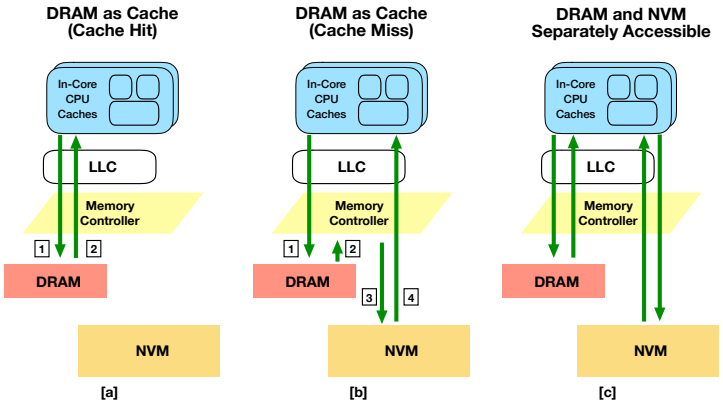

Fig. 1. ***Supported memory modes for DRAM and NVM.*** *If memory access is virtualized by the memory controller, DRAM acts as an architecturally invisible "cache" (a,b). However, unlike a traditional cache, data may also be loaded directly into the last-level cache (LLC) from NVM. The memory controller 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 [1].*

Non-Volatile Memory. Intel recently released the Optane DC Persistent Memory [33] 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 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 [2, 6], 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 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 [24].

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` [27], and on the wider Internet through large-scale content delivery networks (CDN) operating on geographically distributed cache servers [32]. 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 [55]. Each server in a distributed cache is internally addressable, allowing for CACHE-BYPASS, and *gets* and *puts* of objects may have asymmetric performance [15].

3 OPTIMAL DATA PLACEMENT

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.

3.1 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 3.5.

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 paper. 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 1.

3.2 Why Investigate Offline Performance?

Identifying the optimal cache replacement strategies, Belady's MIN and Mattson's OPT (evict-farthest-in-the-future) [11, 44], 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 [13], for example, applied supervised learning to practically map object features to optimal decisions learned from offline analysis. Shi et al. [56] 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 [14, 15], when the cache size is dynamic [40], or when write endurance of the memory needs to be considered [19], 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?*

3.3 Designing an Offline Optimal Algorithm

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. [14] 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.

3.4 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*.

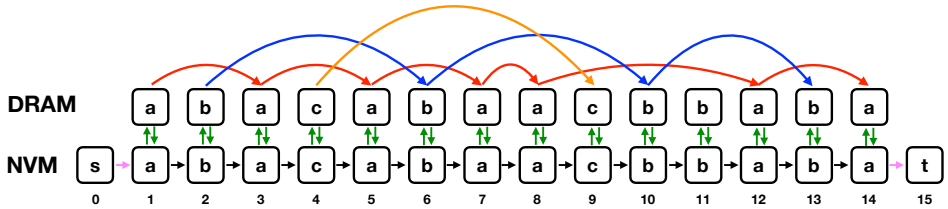

Fig. 2. *Example of a network flow constructed by* CHOPT*, on a trace covering 14 requests for three unique items. Nodes and colored edges are described in Section 3.4.*

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

Optimization. We now run a MCMF algorithm on G [30]. 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 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 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.

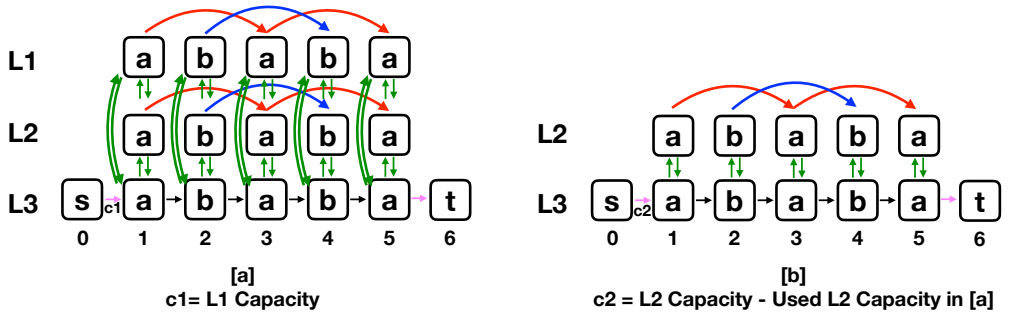

Fig. 3. ***Extending CHOPT to more layers****. First, set* choke links *with capacity* c_1*, apply the* CHOPT *algorithm on* ***(a)*** *to solve for data placement, reconstruct the graph as per* ***(b)****, modify the* choke link *capacity, apply* CHOPT *algorithm again, and finally combine the placement solutions from* ***(a)*** *and* ***(b)****.*

- 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 [46]. We provide proofs of key assertions in Appendix B.

3.5 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 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 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 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 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 [30]. Yet analyzing offline optimal placement is primarily of interest for large-scale real-world workloads, often comprising at least $10^7 - 10^9$ requests [14, 40]. 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 [9, 58, 59] 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 4.9.

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.* [44], we define stack algorithms as follows.

Definition 4.1. 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 [44].

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 4.2. $\mathcal{P}$ 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 4.3. Define r_t as the *stack distance* of element x_t at time t, s.t. $r_t = \left|\{i \in \mathcal{I} : i \prec_t x_t\}\right|$.

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* [59]. Let $\mathbb{1}[A] \in \{0, 1\}$ denote the indicator function for predicate A, such that $\mathbb{1}[A] = 1$ iff A is true, with the standard generalization to random variable A in a probability space.

Definition 4.4. 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

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

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 [58]; 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 4.5. The *sampled stack distance* $\hat{r}_t$ is a random variable, defined as $\hat{r}_t = |\{i \in \mathcal{I}_S : i \prec_t x_t\}|$.

Lemma 4.6. *Either $r_t = \hat{r}_t = \infty$, or the sampled stack distance $\hat{r}_t \approx Binom(r_t, \alpha)$. 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

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

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 4.7. 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,

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

Our main result is the following.

Theorem 4.8. *(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*

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

Corollary 4.9. *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 4.8 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.

5 EVALUATION

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?

5.1 Traces

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. Detailed workload characteristics are described in Appendix A. 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 5.3, we sample from the original traces to reduce runtime as described in Table 7. In Section 5.4, however, our experimental runtime would be astronomically high even with sampling so we choose to trim the traces before sampling.

5.2 Experimental setup

Implementation. We implemented an offline simulator for Chopt in C++, where we apply the Bellman-Ford algorithm for solving the MCMF problem [61]. 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 7.

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

	CHOPT	BELADY	BELADY-AD	LRU	W-TINYLFU
A1	√	×	√	×	√
A2	√	×	×	×	×
Online	×	×	×	√	√

Table 2. *Modeling assumptions for different algorithms.*

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

5.3 CHOPT Simulation Results

CHOPT Performance. Among all workloads in the experiments, CHOPT provides better *NAAL* than any other algorithm at any cache size. Figure 4 presents several results of *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 5 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 also represent the same trace as in Figure 5. Table 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

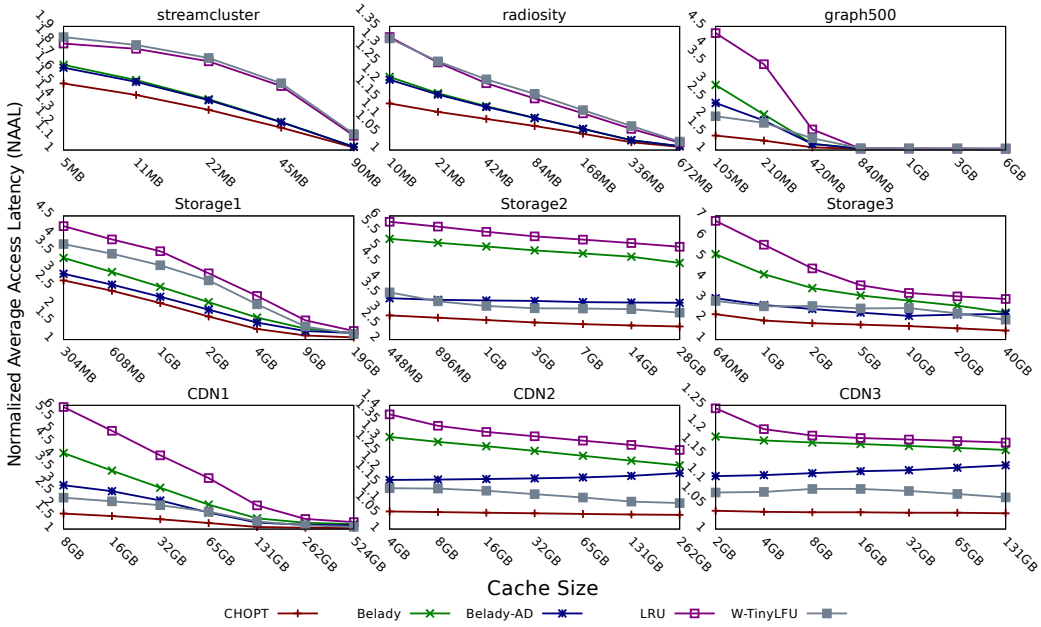

Fig. 4. *Normalized Average Access Latency (NAAL) results on three randomly chosen workloads from each workload category. Cache size is varied from very small to large enough to fit all unique items in the trace.*

	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 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.*

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 2, Chopt, Belady-AD, and W-TinyLFU considers **A1** while Belady and LRU do not. From Figure 4, 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 5 also suggests that algorithms considering **A1** provide lower latency in several workloads, especially in the *Storage* and *CDN* categories. In Table 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

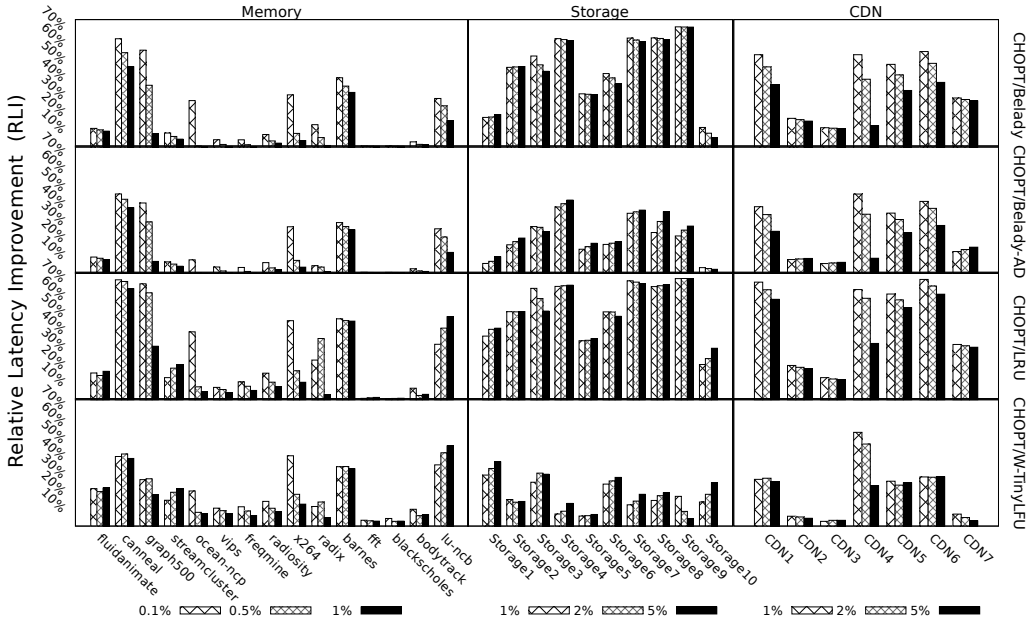

Fig. 5. *Relative Latency Improvement (RLI) results on single workloads over all workload categories (each in a column) algorithms (each in a row). Showing all Memory and CDN workloads, and 10 of 106 randomly chosen Storage workloads. Cache size is varied as* 0.1%, 0.5%, *and* 1% *of unique requests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

Workload	Parameter		Belady		Belady-AD		LRU		W-TinyLFU	
	NVM_s	NVM_ℓ	Avg %	Max %	Avg %	Max %	Avg %	Max %	Avg %	Max %
Memory	5	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	5	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	5	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. *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* NVM_s *as* 2 *and* 5 *respectively.*

of **A1** lies in comparing the controlled results between Belady and Belady-AD, where **A1** is the only variable. From Figure 4 and Figure 5, all workloads show superior *NAAL* and *RLI* on Belady-AD over Belady. From Table 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 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 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 7 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, *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, *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.

5.4 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 12 *Storage*[1] traces, respectively). 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 sampling error. We also evaluate if the accuracy results approximate the theoretical bound as shown in Corollary 4.9.

[1]We randomly selected 12 out of the available 106 *Storage* traces due to limited time available to run tests.

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 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.*

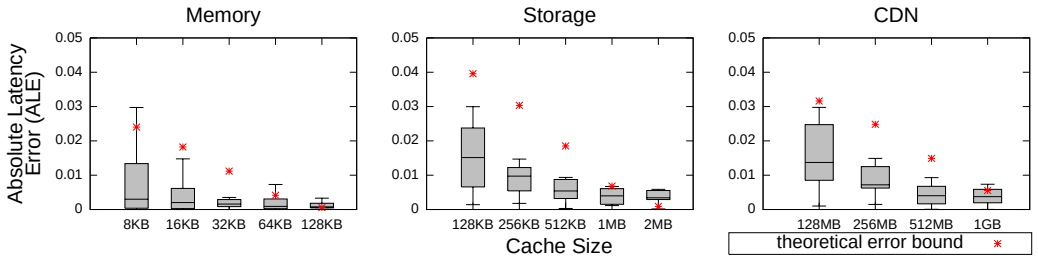

Fig. 6. *Absolute Latency Error (ALE) due to spatial sampling with CHOPT. We show detailed results with a sampling ratio of 1% across selected cache sizes. For each cache size, we plot the theoretical error bound from the spatial sampling theorem, and a box plot of the distribution of ALE for different traces in each category.*

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 [59]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 4.9.

Sampling Accuracy. Table 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 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 [59] 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.* [59].

Figure 6 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 6 demonstrates that our accuracy is close to the theoretical bounds in Corollary 4.9.

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

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

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 4.9, 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 6, 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 Section 6.

6 DISCUSSION

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 [22].

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 [45]) 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 [14], 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 6 shows cases where the empirical error exceeds the bound on expected error from Theorem 4.8. 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. [59] 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.

7 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) [1, 33]. 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 [57] and there has been work towards representing NVDIMMs as volatile NUMA nodes transparently [64]. Read and write asymmetry is another characteristic of NVMs that has been studied to mitigate endurance problems and to improve the write performance [20, 51, 65, 67]. Philipp et al. [49] 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. [54] 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 [39], storing deep learning models [25], and graph analysis [43].

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 [3] presented huge-page aware classification in DRAM-NVM hierarchy for trade-offs between memory cost and performance overhead. Kannan et al. [36] 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. [41] estimated the benefit of page migrations between different memory types by considering access frequency, row buffer locality, and memory-level parallelism. Eisenman et al. [24] 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 [63] proposed DEMOTE techniques where data blocks can be ejected to lower level caches and managed by a global MRU. Gill [29] 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 [28, 66] 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 [38], controlling flash write amplification with DRAM [23], 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 [47] 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. [21, 22] 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. [23] 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 [50, 60].

Optimal Placement Analysis. Various researchers have conducted theoretical analysis of optimal offline data placement. Farach-Colton and Liberatore [26] 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. [4] formulated general caching problems as integer linear programming questions, and proposed a relaxation of optimal placement problems. Bar-Noy et al. [7] 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 [18] 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 [12].

Offline Optimal Placement Policies. There are many practical works on offline optimal placement or caching analysis. Belady's MIN [11] is known as the standard offline optimal caching algorithm for basic cache assumptions. To the best of our knowledge, Berger et al. [14] and Li et al. [40] 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 [14] 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. [40] 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 [22], 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 [10] 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 [8]. Some papers leverage offline analysis of past accesses by creating variants of Belady's MIN algorithm. Jain and Lin [34] predicted object futures by reconstructing Belady's MIN solutions over a window of past accesses. Jain and Lin [35] 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 [9, 31, 37, 52, 53, 58, 59] or temporal sampling [17, 62] 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. [58], and expands on the literature by providing a solid theoretical foundation under these empirical results.

8 CONCLUSION

In this paper, 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.

ACKNOWLEDGMENTS

We thank our shepherd, Daniel S. Berger, for insightful and generous comments on the manuscript, Anna Blasiak and Mike McCall for helpful discussions, and the anonymous SIGMETRICS reviewers for their careful feedback. This work was supported by NSF CAREER Award #1553579.

REFERENCES

[1] 2018. *Intel Optane DC Persistent Memory Operating Modes.* Retrieved Aug 7 2019 from https://itpeernetwork.intel.com/intel-optane-dc-persistent-memory-operating-modes

[2] 2019. *Intel 64 and IA-32 Architectures Optimization Reference Manual.* Retrieved Aug 7 2019 from https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-optimization-reference-manual

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

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

[5] Qasim Ali and Praveen Yedlapalli. 2019. Persistent Memory Performance in vSphere 6.7. (2019).

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

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

[8] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 389–403.

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

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

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

[12] Daniel S Berger. 2018. Design and Analysis of Adaptive Caching Techniques for Internet Content Delivery. (2018).

[13] Daniel S Berger. 2018. Towards Lightweight and Robust Machine Learning for CDN Caching.. In *HotNets*. 134–140.

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

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

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

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

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

[19] Yue Cheng, Fred Douglis, Philip Shilane, Grant Wallace, Peter Desnoyers, and Kai Li. 2016. Erasing Belady's limitations: In search of flash cache offline optimality. In *USENIX Annual Technical Conference (ATC 16)*. 379–392.

[20] Sangyeun Cho and Hyunjin Lee. 2009. 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 (MICRO 42)*. ACM, New York, NY, USA, 347–357. https://doi.org/10.1145/1669112.1669157

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

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

[23] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. 2019. Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification.. In *NSDI*. 65–78.

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

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

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

[27] Brad Fitzpatrick. 2009. *Memcached*. Retrieved Aug 7 2019 from http://memcached.org

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

[29] Binny S Gill. 2008. 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*. USENIX Association, 4.

[30] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. 2016. Firmament: fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 99–115.

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

[32] Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. 2013. An analysis of Facebook photo caching. In *24th ACM Symposium on Operating Systems Principles (SOSP 13)*. ACM, 167–181.

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

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

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

[36] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS – OS design for heterogeneous memory management in datacenter. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 521–534.

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

[38] Kunal Korgaonkar, Ishwar Bhati, Huichu Liu, Jayesh Gaur, Sasikanth Manipatruni, Sreenivas Subramoney, Tanay Karnik, Steven Swanson, Ian Young, and Hong Wang. 2018. 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)*. IEEE Press, 315–327.

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

[40] Pengcheng Li, Colin Pronovost, William Wilson, Benjamin Tait, Jie Zhou, Chen Ding, and John Criswell. 2019. Beating OPT with Statistical Clairvoyance and Variable Size Caching. In *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 19)*. ACM, 243–256.

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

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

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

[44] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Systems Journal* 9, 2 (1970), 78–117.

[45] Nimrod Megiddo and Dharmendra S Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache.. In *FAST*, Vol. 3. 115–130.

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

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

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

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

[50] Hanfeng Qin and Hai Jin. 2017. 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)*. IEEE, 233–236.

[51] Moinuddin K Qureshi, Michele M Franceschini, and Luis A Lastras-Montano. 2010. 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*. IEEE, 1–11.

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

[53] Moinuddin K Qureshi and Yale N Patt. 2006. 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)*. IEEE, 423–432.

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

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

[56] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying Deep Learning to the Cache Replacement Problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 413–425.

[57] Steven Swanson. 2019. Redesigning File Systems for Nonvolatile Main Memory. *IEEE Micro* 39, 1 (2019), 62–64.

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

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

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

[61] Kevin D Wayne. 2002. A polynomial combinatorial algorithm for generalized minimum cost flow. *Mathematics of Operations Research* 27, 3 (2002), 445–459.
[62] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, and Andrew Warfield. 2014. Characterizing storage workloads with counter stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 335–349.
[63] Theodore M Wong and John Wilkes. 2002. My Cache Or Yours?: Making Storage More Exclusive. In *USENIX Annual Technical Conference, General Track*. 161–175.
[64] Fengguang Wu. 2018. PMEM NUMA node and hotness accounting/migration. In *Linux Kernel Mailing List Archive*. https://lkml.org/lkml/2018/12/26/138, Last accessed on 08-08-2019.
[65] Jianhui Yue and Yifeng Zhu. 2013. Accelerating Write by Exploiting PCM Asymmetries. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA) (HPCA '13)*. IEEE Computer Society, Washington, DC, USA, 282–293. https://doi.org/10.1109/HPCA.2013.6522326
[66] Yingjie Zhao, Nong Xiao, and Fang Liu. 2010. 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)*. IEEE, 1–6.
[67] Pengfei Zuo, Yu Hua, Ming Zhao, Wen Zhou, and Yuncheng Guo. 2018. 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)*. IEEE, 442–454.

A APPENDIX: TRACE CHARACTERISTICS

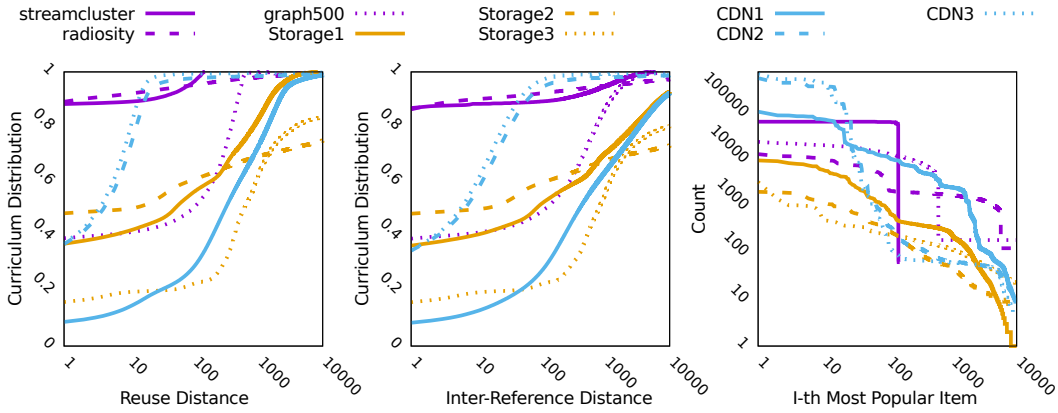

Fig. 7. *Trace characteristics, including reuse distance—the number of unique requests between two neighboring access of the same object, inter-reference distance—the number of requests between two neighboring access of the same object, and object popularity throughout the trace.*

For *Memory* workloads, we use PARSEC [16] 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 [48] benchmark at page level using a Pin [42]-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 [59]. 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 7 shows basic characteristics of our traces, and Figure 7 provides some characteristics of the workloads we used in our evaluation. The

	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 7. *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.*

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.

For *Memory* and *Storage* traces, we use variable sampling ratios to unify sampled trace 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 [14, 40] 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.

B APPENDIX: PROOFS

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

Proof of Lemma B.1. For a given time point $0 \le t \le T$, 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 $1 \le j \le k$. 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. □

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

Proof of Lemma B.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 $f_{(\hat{x}_{t-1},\hat{x}_t)} + f_{(\hat{h}_t,\hat{x}_t)} = N - |C_{t-1}| + \mathbb{1}[x_t = z_t]$. This value in turn equals the outgoing flow

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

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. □

Lemma B.3. *(Chernoff bound) Let $X_1, \ldots X_n$ 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,*

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

$$\mathbb{P}[X \leq (1-\delta)\mu] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\mu} \leq e^{-\frac{\min\{\delta^2,\delta\}\mu}{4}}$$

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

Proof of Theorem 4.8. First assume without loss of generality that $w_t = w$ for all $t \in [T]$. Then

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

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

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 (B.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 $\xi = |T_a|/T$. □