Caching with Delayed Hits

Nirav Atre
Carnegie Mellon University

Justine Sherry
Carnegie Mellon University

Weina Wang
Carnegie Mellon University

Daniel S. Berger
Microsoft Research

ABSTRACT

Caches are at the heart of latency-sensitive systems. In this paper, we identify a growing challenge for the design of latency-minimizing caches called delayed hits. Delayed hits occur at high throughput, when multiple requests to the same object queue up before an outstanding cache miss is resolved. This effect increases latencies beyond the predictions of traditional caching models and simulations; in fact, caching algorithms are designed as if delayed hits simply didn't exist. We show that traditional caching strategies – even so called 'optimal' algorithms – can fail to minimize latency in the presence of delayed hits. We design a new, latency-optimal offline caching algorithm called BELATEDLY *which reduces average latencies by up to 45% compared to the traditional, hit-rate optimal Belady's algorithm. Using* BELATEDLY *as our guide, we show that incorporating an object's 'aggregate delay' into online caching heuristics can improve latencies for practical caching systems by up to 40%. We implement a prototype, Minimum-AggregateDelay (*MAD*), within a CDN caching node. Using a CDN production trace and backends deployed in different geographic locations, we show that* MAD *can reduce latencies by 12-18% depending on the backend RTTs.*

CCS CONCEPTS

• **Networks**; • **Theory of computation** → *Caching and paging algorithms*;

KEYWORDS

Caching, Delayed hits, Belatedly

ACM Reference Format:
Nirav Atre, Justine Sherry, Weina Wang, and Daniel S. Berger. 2020. Caching with Delayed Hits. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, NY, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3387514.3405883

1 INTRODUCTION

Caches are key components of the computer systems toolkit: they reduce bandwidth consumption to a bottlenecked backing store, they improve throughput for memory-intensive services, and they reduce read delays for latency-sensitive applications. Consequently, caches appear across seemingly every class of computer system: *e.g.*, in microprocessors [27], in distributed file systems [51], in CDN proxies [12, 21], and in software switches [47].

In this paper, we focus on a surprisingly overlooked aspect of caching and latency. Caching models and simulators assume that there are exactly two possible outcomes when an object is requested: a low-latency 'hit', or a higher latency 'miss.' In reality, there is a third potential outcome: a **delayed hit** [25, 56]. Delayed hits occur in high-throughput systems when multiple requests for the same object occur before the object is fetched from the backing store.

Our group first encountered delayed hits on an FPGA-based software switch, with incoming packets triggering access to a flow context stored in either an SRAM-based cache (5 *ns* reads), or a DRAM-based global backing store (100 *ns* reads). When a flow's packet results in a cache miss, it triggers the 100 *ns* fetch operation. At high throughput, a second packet of the same flow arrives before 100 *ns* have passed. This packet requests the same object, and waits for it to return from the fetch initiated by the first packet. While the second packet does not have to wait the full 100 *ns* for the object to arrive, it also does not experience a 5 *ns* 'hit' either. Per traditional caching literature, the request corresponding to the second packet would be counted as a hit. In reality, this second packet experiences a slower, 'delayed hit'.

We demonstrate throughout this paper that the traditional caching objective of hit-rate maximization and the related goal of latency minimization are not equivalent problems when some hits are delayed. We argue that therefore we need new algorithms for latency-sensitive caching systems.

One way to understand fundamental trade-offs in caching design is by studying an offline-optimal algorithm. The classic such algorithm is called Belady's algorithm [7]. Unlike real caching systems, offline-optimal algorithms assume an oracle with perfect knowledge of future requests. Offline algorithms can provide guidance and bounds for practical algorithms, *e.g.*, if the offline-optimal algorithm achieves a k% hit-rate, then any online algorithm will achieve *at most* k%. In the past, understanding which objects an offline algorithm chooses to cache or evict has often guided the design of practical systems [9, 29, 55]. Our approach to understanding delayed hits similarly uses lessons from the offline setting to guide our design of a practical online system.

Limitations of existing algorithms: We begin in §2 by showing that Belady's algorithm, the optimal offline approach for hit-rate maximization, does not guarantee minimal latency in the presence of delayed hits. We then measure the gap between hit-rate and latency-oriented regimes on cache traces including a 10 Gbps link and a latency-sensitive CDN. We find that latency evaluations of practical caching algorithms (*e.g.*, LRU [64]) based on hit-rate alone underestimate true latencies by 14-63% in switch caches and 22-36% in CDNs.

Optimal, latency-minimal caching: Having demonstrated that existing caching algorithms fail to minimize latency, we turn to the design of new algorithms that are aware of delayed hits. In §3, we design a new offline caching algorithm, BELATEDLY,[1] which computes empirically tight bounds on the minimum latency in polynomial time. Using BELATEDLY, we quantify the gap between Belady's algorithm – and thus the hit-rate maximization strategy – and true latency-optimality. We find that Belady's latencies are 0.1-38% worse

[1] Available at https://github.com/cmu-snap/Delayed-Hits

SIGCOMM '20, August 10–14, 2020, Virtual Event, NY, USA

ACM ISBN 978-1-4503-7955-7/20/08.
https://doi.org/10.1145/3387514.3405883

than BELATEDLY's latency upper-bounds for packet switches and 1.8-17% worse than BELATEDLY for CDNs.

Low-latency online caching: We use BELATEDLY as our guide for the design of a practical online caching strategy, Minimum-AggregateDelay (MAD), in §4. Specifically, we derive a simple ranking function from BELATEDLY by modeling an object's future 'aggregate delay'. This ranking function empirically approximates BELATEDLY's decisions. We then design a practical aggregate delay heuristic which can be used to make traditional caching algorithms aware of delayed hits. We implement a prototype of MAD within a CDN caching node. In experiments with backends deployed in the US, Europe, and East Asia, we observe average latency reductions of 12% to 18% with a memory overhead of under 3%. We use simulations to explore a wider range of scenarios and find that MAD improves latencies over traditional algorithms by 15-43% for packet switches, 10-60% for CDNs, and 5-40% for distributed storage systems. Most strikingly, for caches with extremely high latencies to the backing store, MAD can provide better average latencies than the latency provided by Belady's algorithm.

Why now? Why hasn't anyone noticed before that delayed hits play an important role in cache latencies? Delayed hits are noted in passing in several places in the literature [25, 56], and anyone who has ever *implemented* a cache has had to consider delayed hits as well [1, 8, 35, 43, 53, 58].

We conjecture that the problem has only recently become perceptible from a performance perspective due to an evolving ratio between system throughputs and latencies. If throughput is low relative to latency, it may only be possible for 1-2 requests to arrive during a fetch. However, if throughput is higher relative to latency, one would expect more requests during a fetch. We refer to the ratio between the object fetch time and mean request inter-arrival time as Z, and we show in §3.2 that as Z grows, the gap between Belady and BELATEDLY widens.

In recent years, Z has grown across a wide range of systems. For example, DRAM latencies are only marginally improving, while newer memory technologies (*e.g.* High Bandwidth Memory, or HBM) boast order-of-magnitude improvements in bandwidth over current DDR standards [37]. Similarly, the latency between a CDN forward proxy and a central data center is defined by wide-area latencies; meanwhile, throughputs are rapidly growing, *e.g.*, with network links moving from 10Gbps to 100Gbps and 400Gbps [24]. The fundamental problem is that latencies are edging marginally closer and closer to limits imposed by the speed of light, while throughputs keep growing unhindered. Hence, we believe that the impact of delayed hits on latency-minimizing caching systems will grow with time.

2 THE PROBLEM WITH DELAYED HITS

A basic delayed hit scenario is illustrated in Figure 1. When a request arrives for an object and the object is not stored in the cache ①, the cache triggers a request to retrieve this object from a backing store ②. The retrieval takes some non-zero amount of time, L seconds, and the average inter-request arrival time is R seconds. For simplicity we say that R seconds is one *timestep*, and that the amount of time to fetch the object is $Z = \frac{L}{R}$ timesteps. After the object is requested, but before Z timesteps have occurred, a new request arrives ③. This request must wait some non-zero, but $< Z$ amount of time for the object to arrive as well ④.

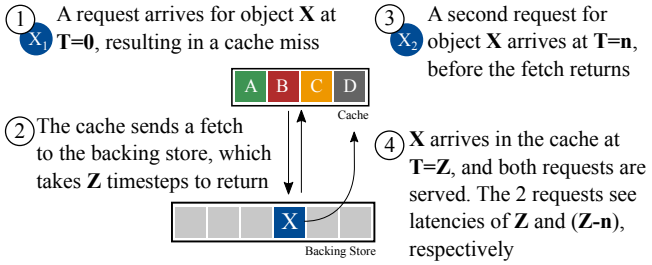


Figure 1: Two requests for object X arrive within Z timesteps of each other. The first request results in a miss, the second request is a 'delayed hit.'

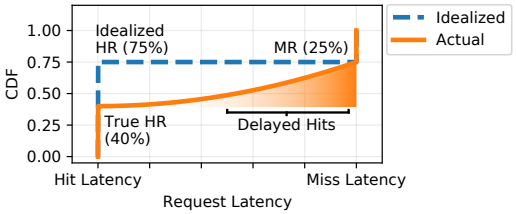


Figure 2: An example CDF of request latencies. Delayed hits account for the gap between the *true* hit-rate (HR) and the miss-rate (MR).

To concretize this notion, consider a cache where $Z = 10$. At timestep $T = 3$, a request for object A arrives, resulting in a cache miss; this triggers a fetch to the backing store for object A which will complete at time $T = 13$, hence the first request will be served after a total latency of 10 timesteps. If additional requests for A arrive at $T = 5$ and $T = 11$, then they too will complete at $T = 13$, and will experience latencies of 8 and 2, respectively.[2]

Traditional caching models ignore the contribution of delayed hits, which, as we show in the following sections, can be significant in systems with high latency to the backing store. Figure 2 depicts the physical interpretation of delayed hits, and the relationship between the *true* hit-rate, the *idealized* hit-rate, and the miss-rate.

2.1 Classic Caching Algorithms

Delayed hits subvert expectations of traditional caching algorithms when it comes to latency. A *caching algorithm* is an algorithm to decide, given a cache and an incoming stream of object requests, when and which objects to store in the cache, and when and which objects to evict. The caching algorithm produces a *cache schedule*: a series of decisions about admissions and evictions for a given set of cache parameters and a given sequence of object requests. A caching algorithm aims to meet a particular objective, *e.g.*, maximizing hit-rate. *Offline ('optimal') algorithms* know of all requests in the future, and can therefore generate a theoretically optimal schedule with regard to the objective. *Online ('practical') algorithms* are aware of past object requests, but not future requests.

Classical caching algorithms are designed with the objective of *maximizing hit-rate*, treating 'true' hits and delayed hits as one category [27, 54]. Measuring the hit-rate (*HR*) allows cache designers to evaluate numerous properties one might wish to extract from a caching algorithm. For example, if a cache is deployed to reduce

[2]Note that for the purposes of our modeling, we assume that processing time for each request is 0 – that is, as soon as the data arrives, all requests are served instantly. In many systems this is not true, and each request must be processed serially, e.g., reading, modifying, and writing updates to the cached object. Non-zero processing times therefore introduce an additional queueing delay which further increases the latency due to delayed hits.

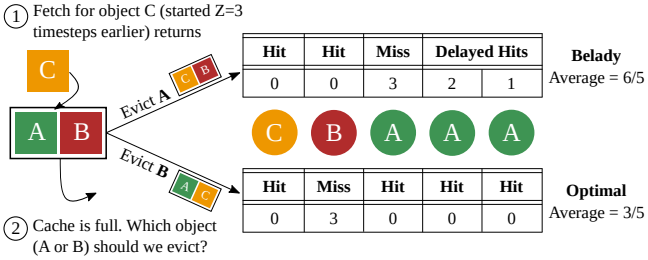


Figure 3: For this trace for a cache of size 2 and a Z of 3, Belady's algorithm chooses a latency-suboptimal schedule.

bandwidth consumption to a backing store (*e.g.* a forward proxy in a bandwidth-limited network), then the miss rate, $MR = (1 - HR)$ is proportional to the bandwidth consumption on the path to the backing store. Other caches are deployed to minimize latency. When assuming delayed hits do not exist (and that backing store latencies are uniform [23, 30, 39]), the average latency is equal to:

$$HR \times \text{hit latency} + MR \times \text{miss latency} \quad (1)$$

In the presence of delayed hits, the latency estimates derived from traditional hit-rate based models *underestimate true latency*. Some so-called 'hits' will in practice experience latencies closer to the high latency of a miss than the low latency of a true hit. As a consequence of this gap, traditional algorithms fail to minimize latency, which we demonstrate in offline simulations (§2.3) and real experiments (§5).

2.2 Belady is Not Latency-Minimal

The classical Belady's algorithm [7] is provably optimal at *both* maximizing hit rates *and minimizing latency* in the basic setting where all objects are the same size [10], and the backing store latency is both uniform and less than the request inter-arrival time. The algorithm itself is simple: when choosing which object to evict from a cache, evict the object whose next request is the farthest in the future.

However, Belady's algorithm is *not* latency minimal when delayed hits are present, as illustrated in Figure 3. In the example, the cache ($Size = 2$, $Z = 3$) currently contains objects A and B, and a fetch for object C (initiated $Z = 3$ timesteps earlier) has just completed. Now, the cache must evict either object A or B to accommodate C. Since B is accessed earlier than A, Belady's algorithm would choose to evict A. However, in our example, we see that there is a *burst* of requests to A, resulting in a series of 'delayed hits' with several requests to A experiencing higher latencies. A caching algorithm that evicts B instead of A experiences a single miss corresponding to B, but all of the subsequent requests to A would have been true hits, resulting in a lower average latency.

	Algorithm Description
LRU	Recency-based heuristic. Evicts the least-recently-used item from the cache [64].
LFU	Frequency-based heuristic. Evicts the least-frequently-used item seen since the beginning of time [20].
ARC	Balances frequency and recency [41].
LHD	Learns hit and lifetime distributions, evicts the object with the lowest *hit density* [5].
Belady	Offline-optimal algorithm for maximizing hit-rate ignoring delayed hits [7]. Requires an oracle of future requests.

Table 1: Overview of traditional caching algorithms.

Trace	Use Case	Latency	Z
CDN	Intra-datacenter proxy [63]	$1ms$	1K
	Forward proxy, nearby datacenter [34]	$10ms$	10K
IRT=1µs	Forward proxy, remote datacenter [34]	$200ms$	200K
Network	Single cache line DRAM lookup [27]	$100ns$	<1
	Traversing a DRAM datastructure [27]	$500ns$	<1
	RDMA Access in GEM-switch [33]	$5\mu s$	2
IRT=3µs	IDS with reverse DNS lookups [46]	$200ms$	67K
Storage	1MB SSD Disk Read [19]	$50\mu s$	2
	Hard Disk Seek & Read [19]	$3ms$	100
IRT=30µs	Cross Datacenter Filesystem Read [19, 62]	$150ms$	5K

Table 2: Average inter-request times (IRT), typical latencies, and Z values for a range of caching use cases.

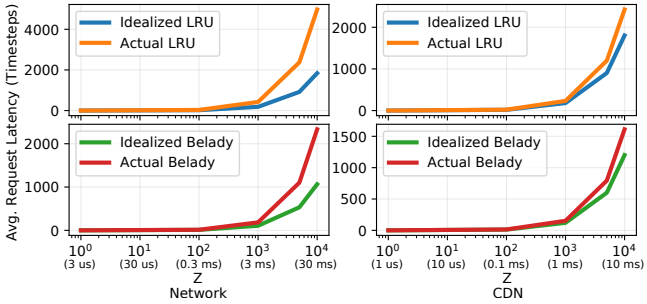


Figure 4: Average latency estimates of *Idealized* (not accounting for delayed hits) and *Actual* (accounting for delayed hits) versions of two standard caching algorithms.

2.3 Delayed Hits and Practical Algorithms

In addition to leading Belady to sub-optimal caching schedules, we also observe that delayed hits can mislead operators managing caching systems in practice. In this section, we simulate four classes of caching systems and observe that delayed hits can inflate latencies beyond what operators might expect from analyzing hit rates; in fact, delayed hits might even lead operators to choose the wrong caching algorithm to deploy for minimal latency.

Experimental Setup: We implement a cache simulator[1] which models delayed hits for a range of caching algorithms listed in Table 1. We rely on datasets from three classes of caching systems: a large content distribution network [11], the CAIDA Equinix 10G Packet dataset [59], and a networked file system at Microsoft [28]. For each trace, we simulate a set of caching scenarios, each with a different backing store latency normalized to Z, the average number of requests arriving during a single object fetch. To provide some context for what Z values one might find in practical systems, we describe a few examples in Table 2.

Delayed hits increase latencies in practice when Z is large. If delayed hits happened infrequently, the gap between the predicted latency derived from hit-rates (1) and true latencies would be marginal. But, in Figure 4, we show how the average latency reported by a simulator that models delayed hits differs from one that does not. In our simulations, we scale up the latency to the backing store; on the X-axis we plot Z, the ratio of the backing store latency to average request inter-arrivals. We see that for both traces, as the latency to the backing store increases (and hence Z), so does the difference between the simulated latency with delayed hits and the predicted latency assuming no delayed hits. Referring back to Table 2, this means that

latencies are noticeably worse than expected for Forward but not Reverse proxies, and for IDS DNS lookups but not DRAM accesses.

Evaluating caching policies on hit-rate alone can lead to selecting the wrong algorithm. The gap between a hit-rate derived estimate of latency and the true latency varies by trace and by algorithm. This means that comparisons of caching policies – even using real, not simulated systems – based on hit-rate measurements and Eq. (1) rather than true measurements of latency may lead to incorrect conclusions about which caching algorithm is 'better' for the system under consideration. Figure 5 depicts pairwise comparisons between four online caching algorithms applied to different application scenarios. *X*s denote situations where choosing an algorithm on the basis of hit-rate alone would result in a worse average latency. We find that in more than one-third of comparisons, not incorporating delayed hits into the system evaluation would lead one to make suboptimal decisions about the 'right' caching algorithm, which would lead to higher average latency in practice.

2.4 Minimizing Latency is Challenging

We have seen that optimizing for hit rate alone is insufficient to guarantee minimal latency. So, which caching schedule minimizes latency when there are delayed hits? Answering this question is more challenging than one might think.

To illustrate the challenge presented by delayed hits, we present an example where the right decision highly depends on Z. Intriguingly, we find that, as Z increases, the right schedule can change entirely. The example consists of the following sequence of requests to objects A and B, which is repeated indefinitely. Requests in yellow (indicated x) denote empty time slots.

We assume a cache of size 1 which either caches A or B. We consider four different Z values corresponding to the following fetch delays (L): $1ms$, $5ms$, $17ms$, and $22ms$ (assuming $R = 1ms$). For each Z value, we calculate the latency achieved by three algorithms: a) caching the bursty flow (A), b) caching the paced flow (B), and c) LRU. A green box denotes the lowest latency for each value of Z.

Algorithm	Z=**1**	Z=**5**	Z=**17**	Z=**22**
Cache Bursty, A	0.5ms	1.9ms	4.3ms	6.0ms
Cache Paced, B	0.5ms	1.5ms	7.5ms	5.5ms
LRU	0.2ms	2.2ms	5.9ms	6.6ms

We find that, while LRU is latency-optimal for $Z = 1$, the paced algorithm is optimal for $Z = 5$. For $Z = 17$, the bursty algorithm becomes latency-minimizing (albeit not optimal), and for $Z = 22$, the paced algorithm is latency-optimal once again. The difference in latencies is significant (between 1.1× and 2.5×) even for this simple example. We conclude that any traditional algorithm, which ignores delayed hits and thus considers only the sequence of requests, cannot expect to achieve good latencies. In fact, even an educated guess, *e.g.* preferring bursty flows – which suffer especially under delayed hits – does not consistently lead to the right strategy.

To further complicate matters, parallel work in our group [40] shows that the latency objective for the delayed hits caching problem is *not* antimonotone.[3] Consequently, a caching algorithm that *improves* average latency under delayed hits might actually *lower* the true hit-rate. In fact, it might even *increase* the miss-rate (i.e. inflate the number of requests sent to the backing store). This finding confirms our intuition that optimizing for latency is a fundamentally different problem than optimizing for hit- or miss-rates. It also has implications for bandwidth consumption of latency-minimizing caching algorithms, which we discuss further in §5.4.

3 LATENCY OFFLINE OPTIMAL

Belady, the offline hit-rate maximizing caching algorithm, fails to minimize latency in the presence of delayed hits, and neither do the heuristic algorithms in §2.4. In this section, we find the answer to the latency-minimization question by reducing it to a Minimum-Cost Multi-Commodity Flow (MCMCF) problem. We present BELATEDLY, an offline caching algorithm we designed to minimize latency given delayed hits. With BELATEDLY, we can measure the gap between Belady and true latency-optimality. Furthermore, BELATEDLY generates a latency-optimal cache schedule which we will later use to guide the design of a practical, online algorithm (MAD).

A latency-minimizing cache schedule minimizes the mean latency of all requests, where latency = 0 upon a true cache hit, latency $\in (0, Z)$ upon a delayed hit, and latency = Z upon a miss. In §3.1, we show that the latency-minimization problem is equivalent to an MCMCF problem.

However, computing integer solutions to MCMCF problems is known to be NP-Complete, and naively implementing the algorithm involves a significant number of decision variables. To make the problem tractable enough to compute over our empirical datasets, we apply two optimizations: (1) we 'prune' and 'merge' states in the MCMCF graph using a priori insights about caching, and (2) we configure our MCMCF solver (Gurobi [45]) to solve for a 'fractional' solution, which can be found in polynomial time, and then perform randomized integer rounding [10, 50] to recover a valid caching schedule. Due to space limitations, we defer the details of these optimizations to Appendix §A.3, and summarize their impact on BELATEDLY's performance in §A.4. The BELATEDLY pipeline is illustrated in Figure 6.

3.1 Network Flow Formulation

We first describe our MCMCF formulation. Due to space limitations, some formal definitions are deferred to §A.1; we provide a proof of equivalence between latency minimizing caching and BELATEDLY in §A.2.

Overview: MCMCF is a classic network flow problem and a generalization of Min-Cost Flow (MCF) [2]. Min-Cost Flow involves a set of *sources* and *sinks* embedded in a larger graph; every edge in the graph has a *capacity* representing the maximum amount of flow which may traverse that edge. A solution to MCF must route *flow* from the sources to the sinks without exceeding any individual edge capacity. Furthermore, each edge is also associated with a *cost*. The ultimate goal of Min-Cost Flow is to route flow across the edges *such that the total cost of all traversed edges is minimized*. MCMCF adds an additional twist to the problem: flows are associated with a *commodity*, and edges may have different costs for different commodities.

[3] For a request sequence of size T, we can encode a cache schedule as a *hit vector* of boolean values, $b \in \{0,1\}^T$, where $b_i = 1$ if the i'th request experienced a *true* hit, and $b_i = 0$ otherwise (i.e. delayed hit, or miss). Then, we can define a latency function, $l : \{0,1\}^T \to \mathbb{R}$, such that $l(b)$ represents the total latency for schedule b. We say that l is antimonotone if, for every pair of schedules $b, b' \in \{0,1\}^T$, where $b'_i \geq b_i \forall i$, it holds that $l(b') \leq l(b)$. Perhaps surprisingly, [40] shows that this is *not* the case, implying that it is sometimes preferable to forgo a true cache hit in order to achieve lower latency.

	LRU	LFU	ARC	LHD
LRU		✗	✓	✓
LFU			✗	✓
ARC				✗

(a) Network (CAIDA ORD, $z = 2K$)

	LRU	LFU	ARC	LHD
LRU		✗	✗	✓
LFU			✗	✓
ARC				✓

(b) Network (CAIDA NYC, $z = 2K$)

	LRU	LFU	ARC	LHD
LRU		✓	✓	✓
LFU			✗	✗
ARC				✓

(c) CDN ($z = 100K$)

Figure 5: Pairwise comparisons between online policies.

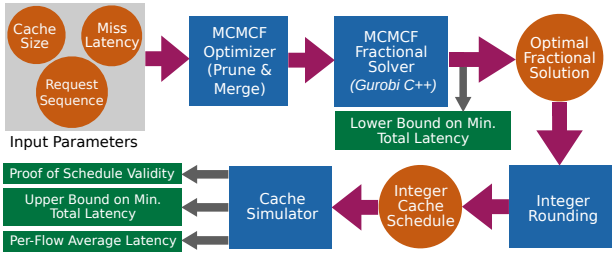
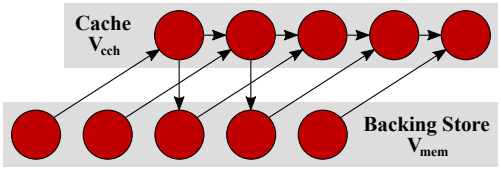


Figure 6: The BELATEDLY pipeline for computing bounds on latency-optimal cache schedule using MCMCF reduction.

Our reduction from minimum latency caching to MCMCF constructs a commodity for each object requested from the cache. Vertices in the graph represent either that the object is in the cache, or that it is in the backing store; edges between vertices represent the object entering the cache, remaining in the cache, or being evicted from the cache. Weights along edges represent the latency cost of misses and delayed hits. By minimizing the weights of traversed edges, MCMCF equivalently computes a cache schedule with a minimal latency cost.

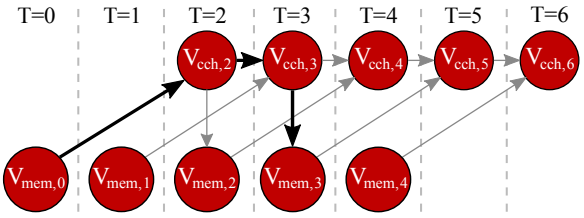
A key component in this formulation is the *costs* we assign to edges in the flow network, which reflect the *true* latency costs of misses. Our main finding is that the right costs to assign are 'aggregate delays'. Specifically, the aggregate delay of a miss is the total delay of the miss and all the delayed hits within a time window of Z of the miss (see Eq. (2) for the mathematical definition). *This notion of aggregate delay influences the design of our online algorithms, discussed in §4.*

Construction of the Flow Network: BELATEDLY operates on a *flow network*, a directed graph consisting of a set of vertices and edges. In our formulation, the vertex set, V, consists of two types of vertices, which we draw as two rows. The bottom and top rows represent the backing store and the cache, respectively. We refer to the set of 'backing store' nodes as V_{mem}, and the set of 'cache' nodes as V_{cch}.



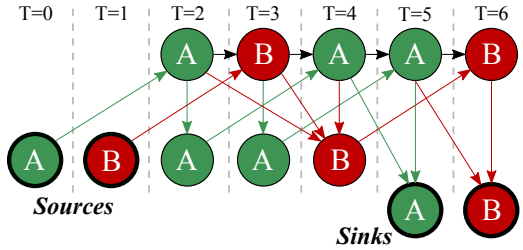
Note that the rows are slightly *offset*. This is because we index each row by time, and have vertices for each timestep. For the bottom row, we have one vertex for each timestep $T = 0, 1, \ldots, N-1$. We denote these vertices as $V_{\text{mem},T}, T = 0,1,\ldots,N-1$. For the top row, we duplicate the vertices in the bottom row, but shift them to the right by Z timesteps as shown in the figure below. We denote the vertices in the top row by $V_{\text{cch},T}, T = Z, 1+Z, \ldots, N-1+Z$. In the figure below, $Z = 2$.

Flow moving along an edge represents an object moving in and out of the cache. In the following figure, an object is requested at time $T = 0$, arrives in cache at time $T = 2$, and is evicted at time $T = 3$.



Since there are multiple objects, we view each object as a commodity and index them by $i \in [M]$, where M is the number of objects and $[M] = \{1,2,\ldots,M\}$. We also say $\sigma(x)$ is the object requested at time x. We have 1 unit of demand for each object. The source vertex for each object, i, is the vertex V_{mem,T_i} where T_i is the first timestep at which i is requested. We also add a sink vertex for each object i in the bottom row, denoted by $V^{(i)}_{\text{sink}}$.

At a high level, each node in the bottom layer represents the time of request to exactly one object; we construct an edge from $V_{\text{mem},t}$ to $V_{\text{cch},t+Z}$ to allow the flow for that object to move from the backing store to the cache. In the top layer, each node $V_{\text{cch},t+Z}$ represents the request from time t being served. Objects may stay in the cache by following edges from some $V_{\text{cch},n}$ to the next $V_{\text{cch},n+1}$ – all nodes in $V_{\text{cch},}$ have an edge to the subsequent cache node. To leave the cache, an object follows an edge from some $V_{\text{cch},n}$ to some $V_{\text{mem},x}$ for x, the next time ($\geq n$) the same object is requested – hence all nodes in $V_{\text{cch},}$ have M edges back to $V_{\text{mem},}$ nodes, one for each object that *could* be evicted at this point. If there is no further request to an object, the edge points to the *sink* node for that object rather than $V_{\text{mem},x}$. We illustrate the request sequence $\{A, B, A, A, B\}$ for objects A and B:

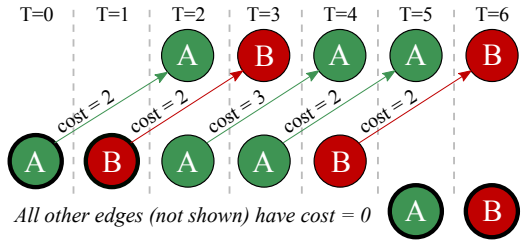


Looking at the above figure, it is obvious that some edges will never be taken (*e.g.* $V_{\text{cch},2}$ has an edge to $V_{\text{mem},4}$ despite the fact that it is impossible for flow for object B to have reached $V_{\text{cch},2}$. We discuss pruning superfluous edges and merging nodes to improve performance in §A.3.

The last features to add to our construction are *capacities* and *costs* along edges to ensure that each object's flow obeys a valid caching schedule that minimizes latency. For example, we want to prevent all objects simply following the edges $(V_{\text{cch},n}, V_{\text{cch},n+1})$ for the entire duration and exceeding the cache capacity. No more than *capacity* flows may traverse an edge, and our solver will try to minimize the *total cost* of routing flow across these edges. We assign capacity and cost to edges as follows:

- $(V_{\text{cch},n}, V_{\text{cch},n+1})$ edges (which represent staying in the cache) are assigned capacity C, and the cost of routing flow across them is 0. This models the fact that staying in the cache does not increase latency, but the cache can only hold C objects at the same time.
- $(V_{\text{cch},n}, V_{\text{mem},x})$ edges (which represent evicting an object whose next request is at $T = x$) are assigned capacity 1, and the cost is ∞ for all commodities *except* $\sigma(x)$ (the object requested at time x), for which the cost is 0. This prevents objects from exiting the cache along edges for a different object. Intuitively, the action of eviction itself does not incur a latency cost. But it forces the object out of the cache so the next request for the object and the corresponding delayed hits will experience non-zero latencies.
- $(V_{\text{mem},T}, V_{\text{cch},T+Z})$ are the edges that represent bringing an object into the cache, which happens when there is a miss. It is here that we encode delayed hit latency into the cost. The capacity of $(V_{\text{mem},T}, V_{\text{cch},T+Z})$ is 1, and the cost is ∞ for all objects other than $\sigma(T)$. The cost of routing $\sigma(T)$ along $(V_{\text{mem},T}, V_{\text{cch},T+Z})$ is the *aggregate delay* for requests of object $\sigma(T)$ while the data is being fetched; i.e., it is the *total latency* for the miss plus all requests that arrive during the delayed hits. The miss experiences a latency of Z, and a delayed hit that arrives t timesteps after the miss experiences a latency of $Z-t$. Therefore, the cost is:

$$Z + \sum_{t=1}^{Z-1} \mathbb{1}_{\{\sigma(T+t)=\sigma(T)\}} \cdot (Z-t). \tag{2}$$



In the above figure, the cost for all edges is 2 (the latency Z to the backing store) except for the edge $(V_{\text{mem},2}, V_{\text{cch},4})$. Because A is also requested at $T = 3$, it will be queued and later served by the request being fetched; as such, we need to account for *both* the cost of serving the request at $T = 4$ (which is 2) and the request at $T = 3$ (which is 1).

Routing Flows: The MCMCF problem is to find routes for the objects such that the total routing cost is minimized. Specifically, the routes are represented by flow variables, where each flow variable represents whether an object/commodity is routed along an edge or not. Here flow variables need to satisfy link capacity constraints and flow conservation constraints, which will guarantee that the flow variables can be converted to a valid cache schedule.

Equivalence to Latency-Minimizing Caching:

Theorem 1. *BELATEDLY's underlying MCMCF problem (§A.2) is equivalent to the latency minimization problem (§A.1).*

The detailed proof of Theorem 1 can be found in §A.2. Both the MCMCF problem and the latency minimization problems are optimization problems. To show that these are equivalent,[4] we first show in Lemma 1 that the feasible set of flow variables is "equivalent" to the feasible set of caching schedules (i.e. from any feasible cache schedule, we can define a set of flow variables that are also feasible for the MCMCF problem, and vice versa).

Lemma 1. *Given a sequence of object requests, there is a bijection between the set of feasible flow variables and the set of feasible cache schedules.*

Once we have this bijection, we can show that the objective functions of these two problems are the same. With equivalent feasible sets and objective functions, the MCMCF problem and the latency minimization problem are thus equivalent.

[4] At this juncture, one might ask: why bother with MCMCF instead of solving the latency minimization ILP directly? The answer is three-fold. First, the LP is convoluted and quite unintuitive (in fact, we discovered the MCMCF formulation first!). Second, it is the network flow formulation that allows us to implement the optimizations described in §A.3; without these, even modest LP instances of the problem are too compute- or memory-intensive for today's solvers. Finally, formulating the problem as an MCMCF naturally leads to the notion of aggregate delay; as we show in §4, this is a key component of our online algorithm.

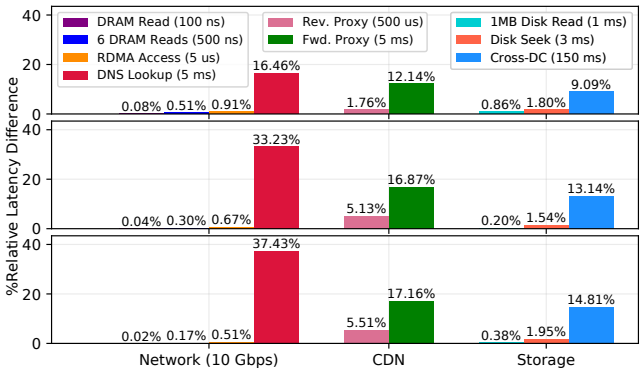


Figure 7: Latency gap between Belady and BELATEDLY for different application scenarios (Network, CDN, Storage) today. Top to bottom: 1%, 5%, and 10% cache sizes.

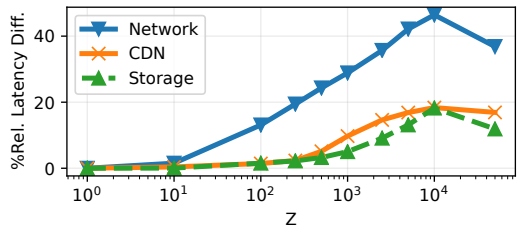


Figure 8: %Relative latency difference between Belady and BELATEDLY versus z. Cache size, $c = 5\%$.

3.2 Delayed Hits and Empirical Latencies

We now evaluate BELATEDLY's latency estimates relative to Belady for a range of application scenarios.

BELATEDLY provides significantly better average latency than Belady for today's highest-latency systems. In Figure 7, we plot Belady's percent error relative to the optimal upper-bound provided by BELATEDLY.[5] For the highest latencies – referring to Table 2, those with Z values in the thousands – Belady deviates from the optimal by 9-37%. However, for more modest latencies to the backing store, BELATEDLY does not have noticeably lower latencies than Belady. Even in the original FPGA-based switching scenario which caused us to detect delayed hits, the gap between Belady and BELATEDLY is less than 1%.

[5] $\frac{(\text{Belady} - \text{BELATEDLY})}{\text{BELATEDLY}} \times 100\%$

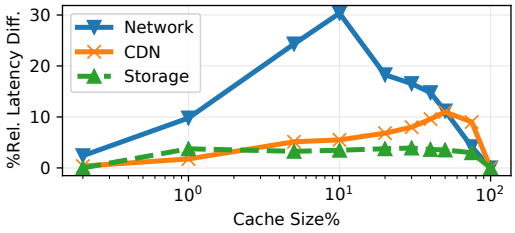


Figure 9: %Relative latency difference between Belady and BELATEDLY versus cache size (expressed as a percentage of the maximum number of concurrent flows). Using $Z = 500$.

Z is correlated with an increasing gap between Belady and BELATEDLY. In Figure 8 we see that for all three datasets, Belady performs progressively worse with respect to true latency optimality as Z increases – until Z moves past $10K$. The growth correlation follows intuition: as Z grows, there are more chances for delayed hits to occur, and hence more opportunities for Belady to err. We find that narrowing of the gap between Belady and BELATEDLY beyond $Z = 10K$ is an artifact of our simulation duration; since Z is large relative to the size of the trace ($250K$ requests), it also exceeds the duration of most flows. As a result, most requests experience 'forced' cache misses, raising the latency baseline and giving BELATEDLY fewer opportunities to make meaningful caching decisions.

The gap between Belady and BELATEDLY varies with cache size. In Figure 9, we see that the latency difference first rises, then falls as the cache size increases. When the cache is extremely small, neither BELATEDLY nor Belady's caching decisions have significant impact on latency (since most requests experience cache misses, the average latency is close to the full latency of a cache miss); similarly, as the cache capacity becomes very large, both strategies can afford to simply cache all or almost all objects (the extreme case being a cache large enough to fit all concurrent flows or active objects). In between, however, all three datasets 'peak' at different points. In particular, the Network trace has a sharp spike at 10%, while the CDN and Storage traces have more gradual curves.

BELATEDLY's caching decisions are correlated with the burstiness of requests. The Goh-Barabasi Score [26] is a statistical measure of 'burstiness' in a sequence of events. A score of '1' reflects many arrivals in a short period of time (a 'train') followed by longer periods with no requests. A score of '-1' represents a perfectly paced stream of arrivals with one request every fixed number of timesteps. In Figure 10, we see that bursty traffic (with a high Goh-Barabasi score) incurs a lower percent latency relative to Belady. This suggests that burstiness may be a worthwhile candidate for consideration in the design of *online* algorithms that optimize for latency in the context of delayed hits. It is this observation that guides us in the development of our online strategy, and we discuss it in more detail in the following section.

4 APPROXIMATING BELATEDLY

BELATEDLY provides two principal lessons for the design of improved low-latency caching algorithms. First, BELATEDLY demonstrates that the opportunity for latency improvement is high: the gap between latency-optimal and hit-rate optimal can be as much as 45%. Second, BELATEDLY provides us with a caching schedule that *achieves* optimal latency for a given trace and Z value.

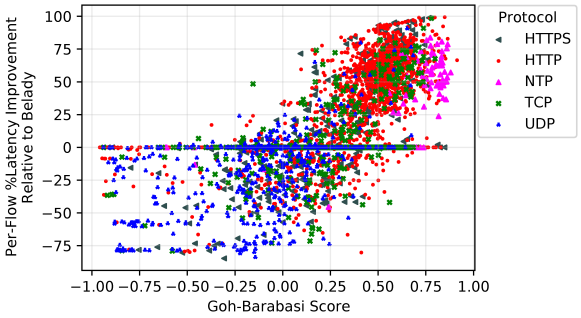


Figure 10: Relative latency improvement *vs* burstiness (for Network traffic). Bursty flows suffer less under BELATEDLY.

Unfortunately, BELATEDLY is *slow* – taking up to 8 hours to compute an optimal schedule for a trace with 250,000 requests – and *requires knowledge of the future*. Both of these properties mean that BELATEDLY itself cannot serve as a caching algorithm for practical systems.

In this section, we learn from BELATEDLY's optimal schedule how to achieve better latencies in practical implementations. In §4.1 we first explore heuristics in the offline setting. In this setting, we still assume an oracle with perfect knowledge of future requests, but we target a computationally tractable algorithm. In §4.2 we then move to a fully online setting where the algorithm both needs to be efficient *and* operate without knowledge of future requests.

4.1 Offline Approximations: Belady-AD

We seek a heuristic *ranking function* which quickly tells us the priority of an object for our goal to minimize latency. In practice, almost all caching algorithms use some ranking function, *e.g.*, LRU – an online algorithm – prioritizes objects by how recently they were last used. Belady – an offline algorithm – is the inverse and ranks objects by how *soon* they will be used in the future. These ranking functions prioritize hit rate whereas we seek a ranking that minimizes latency.

To derive a ranking function, we look to BELATEDLY. While we cannot simply emulate BELATEDLY's behavior (unfortunately, flow algorithms like BELATEDLY don't reveal *how* they make decisions), we can search for easily measurable metrics correlated with BELATEDLY's caching decisions. As we discussed in §3.2, BELATEDLY prioritizes caching bursty objects, i.e. those objects with a high Goh-Barabasi score [26]. We experimented with ranking functions based on this score. While these functions had excellent runtime performance (the Goh-Barabasi score is a function of mean and variance, both of which can be measured cheaply with online algorithms), they delivered poor latency results. Therefore, burstiness on its own is not a good ranking function, which confirms the intuition we derived in §2.4

Instead, we turn to another metric that is directly associated with the latency cost of bursty flows: *aggregate delay*, which is computed in Eq. (2). To compute the rank of an object, we assume that the object's next access in the future is a miss. Its aggregate delay is the sum of the delay due to the miss and any delayed hits which occur during the next Z timesteps while the object would be fetched. Intuitively, an object with a higher delay cost – with a burst of requests during that Z window – increases average latency more than an object with a lower delay cost, and hence should be prioritized.

Nevertheless, aggregate delay by itself is still not an effective ranking function. Consider the ranking of two objects A and B in a cache where $Z = 3$ as shown in Figure 11. A has an aggregate delay of 6 and will not be accessed for another 100 timesteps. B has an aggregate

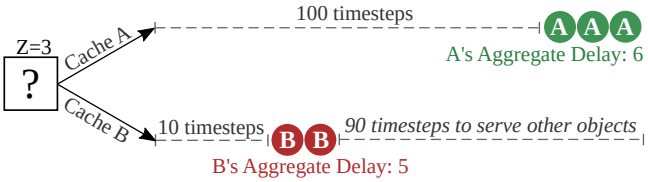


Figure 11: Ranking objects *solely* based on aggregate delay may lead to poor utilization of cache space.

delay of 5 and will be accessed only 10 timesteps in the future. Should the rank function prefer A or B? Assuming we keep the cached object until its next access, keeping A utilizes one cache line – which cannot be used for other objects – for a very long interval. On average, each timestep we keep A in the cache will 'save' an average of $\frac{6}{100}$ units of delay. On the other hand, for each timestep we keep B in the cache, we save an average of $\frac{5}{10}$ units of delay, with the opportunity to cache other objects in the remaining 90 timesteps. Hence, B appears to be – on average – a more efficient use of cache space.[6]

Following this intuition, our offline ranking function, BELADY-AD, computes two values for each object. $AggDelay(x)$ is the aggregate delay for the next access to object x, and $TTNA(x)$ is the number of timesteps until the next access to x.[7] The rank is then:

$$Rank(x) = \frac{AggDelay(x)}{TTNA(x)} \tag{3}$$

We find that, across all Z values, the average request latency provided by BELADY-AD is within 0.1-12% of BELATEDLY. In Figure 12, we show the average latency for BELADY-AD and BELATEDLY (normalized against the performance of Belady's algorithm) for a range of Z values for the CAIDA Chicago network trace; BELADY-AD closely trails BELATEDLY, although the gap between the two widens as Z grows. Furthermore, BELADY-AD runs several orders of magnitude faster than BELATEDLY, computing a cache schedule in under 3 seconds for a trace containing 250,000 requests, where BELATEDLY would take up to 8 hours.

4.2 Online Algorithm: MAD

Finally, we turn to the true online setting, where we both need to use simple heuristics to rank objects and do not have knowledge of the future. Fortunately, we can use the past to make predictions about the future. Just as LRU uses recency as a ranking function – the 'inverse' of Belady's algorithm – we need to 'flip' our measures of $AggDelay(x)$ and $TTNA(x)$ to use data from past requests rather than future ones.

[6]This intuition does not necessarily lead to optimal decisions! For example, if we were to prefer B and evict A, but in the 90 timesteps after B *no other requests arrived* then it would have been better to prefer A.

[7]Note that Belady's algorithm uses the ranking function $\frac{1}{TTNA(x)}$ alone.

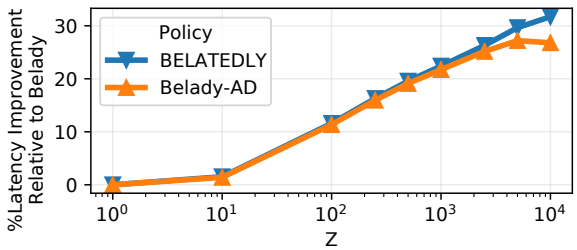


Figure 12: BELADY-AD closely trails BELATEDLY.

Luckily, we already have a large literature of estimators for $TTNA(x)$, as almost all algorithms are essentially predictors of the next access to an object. Recall that Belady's algorithm ranks objects by $TTNA(x)$ alone, and is optimal in the absence of delayed hits. Hit-rate optimizing algorithms aim to operate as close to Belady as possible [55], and so the closer their ranking function is to $\frac{1}{TTNA(x)}$, the better they perform. Hence, in §5 we experiment with using LRU [64], ARC [41], and LHD [5] as estimators of $TTNA(x)$.

This leaves us with estimating $AggDelay(x)$. Recall that we measure Aggregate Delay by assuming that the next request to object x will be a miss, and computing the sum of delays for the miss to x and any subsequent delayed hits for x. We 'flip' this by assuming that all *past* requests to x were misses and then calculating the average aggregate delay per miss; we illustrate this in Algorithm 1. We find that this approximates the true $AggDelay(x)$ well, *e.g.* with a Pearson Correlation Coefficient of 0.7 for the network trace.

Finally, to create MAD, we combine the code[8] from Algorithm 1 with a known estimator for $TTNA(x)$. We can now compute the rank using Eq. (3).

Algorithm 1 Estimating AggregateDelay

```
1:  struct ObjectMetadata
2:      NumWindows = 0
3:      CumulativeDelay = 0
4:      WindowStartIdx = −∞
5:
6:  function EstimateAggregateDelay(X: ObjectMetadata)
7:      return X.CumulativeDelay / X.NumWindows
8:  end function
9:
10: function OnAccess(TimeIdx, X: ObjectMetadata)
11:     // Time since start of the previous miss window
12:     TSSW = (TimeIdx - X.WindowStartIdx)
13:
14:     if TSSW ≥ Z then
15:         // This access commences a new miss window
16:         X.NumWindows += 1
17:         X.CumulativeDelay += Z
18:         X.WindowStartIdx = TimeIdx
19:     else
20:         // This access is part of the previous miss window
21:         X.CumulativeDelay += (Z - TSSW)
22:     end if
23: end function
```

We note that parallel work [40] in our group has shown that any deterministic online algorithm for the delayed hits problem has a competitive ratio[9] of $\Omega(kZ)$, where k is the size of the cache. Despite falling in that category, our empirical evaluations show that MAD yields considerable latency improvements over traditional caching algorithms, and its simplicity lends itself well to implementation. We leave to future work to find a randomized caching strategy which improves upon MAD's worst-case performance.

[8]For the sake of brevity, the provided pseudocode assumes discrete timesteps and prior knowledge of Z. Both of these assumptions are easily dispensable.

[9]The *competitive ratio* of an online algorithm, α, is the *worst-case ratio* between the costs of the solution computed by α to that of the optimal, offline solution for the same problem instance. Knowledge of a caching algorithm's competitive ratio allows us to impose bounds on its worst-case performance (i.e. for the most pessimal workload) [57].

5 EVALUATION

We evaluate the effectiveness and the overhead of MAD in a CDN caching system. We then use our simulator from §2.3 to explore a wider range of applications and parameters.

5.1 Experimental setup

Prototype. We emulate a CDN deployment with clients and backends in geographically different locations. For rapid prototyping, we implement our own asynchronous caching system in 1500 lines of C++ code, using Boost.ASIO [4, 52]. Our architecture uses sharding and a single thread per cache shard [5, 12, 22]. An overview of the system architecture is depicted in Figure 13.

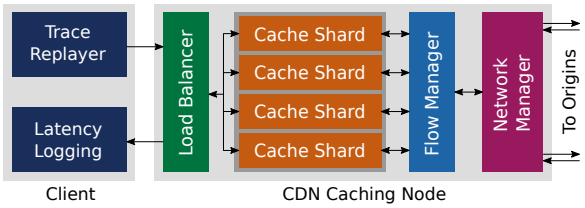


Figure 13: Architecture of our experimental prototype.

The client sends requests as 16B object IDs to the *Load Balancer*, which forwards it to the *Cache Shard* corresponding to the object ID. The shard's thread performs a cache look-up. If the object is cached, the request is resolved immediately by relaying a response back to the client (a *true hit*). Else, the request is forwarded to the *Flow Manager*, which maintains queues of outstanding requests separately for each unique object ID.[10] On receiving a request, if the object ID is not mapped to an existing queue, the Flow Manager allocates a new queue for the object and forwards the request to the *Network Manager* (a *miss*). Else, the request is simply inserted at the tail of the queue (a *delayed hit*). The Network Manager use a pool of threads with long-running TCP connections to the backing stores. These threads perform the actual fetch operation and relay the response to the Flow Manager. The Flow Manager buffers the response, flushes the request queue for the corresponding object ID, and issues a write request to the appropriate cache shard. The cache is updated (based on the specified caching policy), and the responses are sent to resolve all queued client requests.

To achieve low latency and high concurrency, the system components communicate using lock-free, single-producer single-consumer queues. The system is capable of sustaining a throughput of 1.2M requests/sec using 12 threads on an x86 server with 16GB of DRAM.

Cache configuration and policies. We use a 64-way set-associative cache, with the total cache size set to 5% of the maximum number of active concurrent objects (e.g, 67k cache entries overall for the CDN trace from §2.3). For the purpose of our experiments, we fix the object size to $1KB$. We implement two policies: *LRU*, and *LRU-MAD*, which combines LRU's $TTNA(x)$ estimator and our $AggDelay(x)$ estimator.

Traces. We use a busy period from the CDN trace from §2.3 which contains 243M requests, 7.7M unique object IDs, a maximum of 1.3M active concurrent objects, and an average inter-request time of 1 μs.

Setup. To emulate different Z values, we set up backing stores (using GCP VMs) in three different locations around the world: The U.S. West Coast (Los Angeles), Western Europe (the Netherlands), and East Asia (Singapore). For simplicity, we refer to these as Origin A, with an RTT of *68ms* ($Z = 68k$), Origin B, with an RTT of *103ms* ($Z = 103k$), and Origin C, with an RTT of *226ms* ($Z = 226k$), respectively.[11] We deploy our CDN caching node on a server at CMU in Pittsburgh.

[10] We use separate request queues to avoid head-of-line blocking.

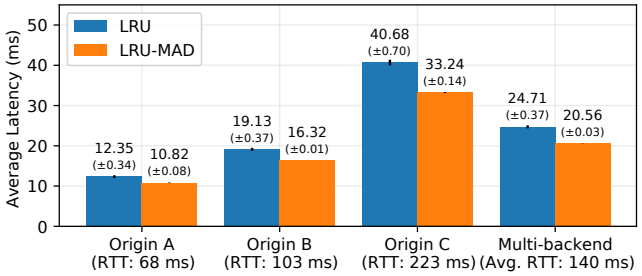


Figure 14: Prototype results for different origin locations.

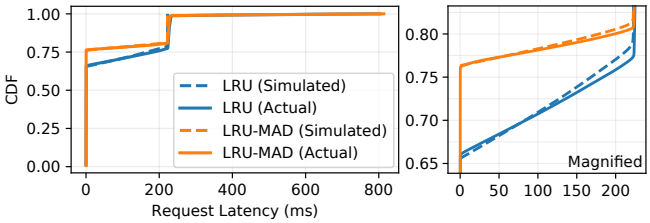


Figure 15: CDF of latencies in simulation versus real experiments.

5.2 Prototype Evaluation on CDN Trace

What latency improvements does LRU-MAD provide for our wide area cache? To answer this question, we consider each of the three backing stores independently, and measure the *average request latency* provided by the two caching policies for the given workload. Figure 14 shows the average latencies achieved using LRU-MAD versus LRU. Overall, using LRU-MAD, we see a 12.4%, 14.7%, and 18.3% reduction in average latency for Origins A, B, and C, respectively. As expected, LRU-MAD's benefit increases with Z.

Does the MAD caching strategy still work if multiple, non-uniform backing store latencies[12] are involved? This differs significantly from our offline formulation which only considered uniform latencies (i.e., a single Z value). We find that MAD indeed works well in the multi-backend scenario. Figure 14 shows a 16.8% reduction in average latency for this case. This result suggests that maintaining per-object estimates of the backing store latency (instead of a single, global average) is an important feature of the online strategy, since it gives MAD a higher degree of freedom in computing ranks.

What are the overheads of using MAD? We discuss two kinds of overheads associated with MAD: *memory* and *request latency*. We evaluate the memory overhead of two different implementations of MAD. Both implementations maintain 4 counters per object. Our *strawman* implementation faithfully implements MAD by persisting these counters for both cached and uncached objects. However, in a long-running caching system, this would require an unbounded amount of memory. Our *efficient* implementation only stores the

[11] We remark that, although the backing store latencies are known a priori, we do *not* explicitly provide this information to MAD; instead, MAD automatically computes per-object estimates of backing store latencies at run-time.

[12] We map each object ID to a randomly-generated *origin location*, which places a third of object IDs on each origin server. The distribution of *requests* is: 29% to Origin A, 39% to Origin B, and 32% to Origin C.

counters for currently cached objects. Fortunately, we find that the average latency provided by the efficient implementation never diverges from the strawman by more than 6% over the entire range of Z values, across all traces. In fact, all results presented so far have been using the efficient implementation. Our counters are 8B; so, the overall overhead is 32B per cached object, which is comparable to existing key value stores [22]. Our efficient implementation thus has a memory overhead of just over 3% for small 1KB objects and under 0.003% for objects in the MB range (e.g., video caching [42]).

We compare MAD's request latency to LRU, where eviction is a constant-time operation (the entry to evict is always at the head of a linked list). Evictions in MAD require computing the *rank(X)* function from §4.2 over all objects in the corresponding cache set. While each computation is cheap, its complexity scales linearly with the set-associativity of the cache in our naive implementation. This leads to several microseconds of overhead, which is orders of magnitude lower than the latency of the backing store. We remark that this small overhead can be further reduced using existing techniques.[13]

How accurately do our simulations reflect results in the wide area? We use simulated results in §2.3 and in the following evaluation sections. While our simulator models the effects of delayed hits, it makes several simplifications. For example, it assumes that arrivals neatly fall into discrete time slots, that cache management operations are instantaneous, and that network latencies are deterministic. We validate these simulation results by comparing the latency distribution (CDF) measured with our prototype to simulations based on averaged estimates of Z for Origin B (results for other origins are the same). Figure 15 shows that the simulated latencies indeed closely match the empirical measurements.

5.3 Simulation Results: Systems

Our prototype experiments focus on the CDN setting with a small set of backing latencies and a single algorithm. We now return to our delayed hits aware simulator to test three MAD variants (LRU-MAD, LHD-MAD, and ARC-MAD) in the context of CDNs, network traces, and storage traces.

How does MAD help CDNs with other base algorithms and a wider range of latencies? Figure 16 illustrates the performance gains from combining $AggDelay(x)$ with LRU, LHD, and ARC. The y-axis measures the relative improvement in latency between LRU-MAD and LRU, LHD-MAD and LHD, and ARC-MAD and ARC. MAD *always* performs better than the baseline algorithm, suggesting that there is no downside, from a latency minimization perspective, to adopting MAD– regardless of what ranking algorithm was used initially. As with our LRU prototype, we see gains of 5-20% when latencies are in the 10's of milliseconds.

We also see that as Z reaches some extreme values – 1M or even 10M – the gains from MAD increase dramatically. Today, these examples are only useful for an imaginary web user with a CDN cache on the moon. However, they *may* serve as an estimate for the impact of delayed hits on future workloads. Recall that Z does not represent latency itself, but the *ratio* between latency to the backing store and request inter-arrival time (§2). Hence, as link and request rates grow by 10×, a Z value of 1M would only represent a 100ms latency for the CDN. Nonetheless, these extreme values remain flawed estimators –

[13]Large-scale production systems achieve constant-time evictions using sampling techniques [5], which can be immediately applied to an implementation of MAD.

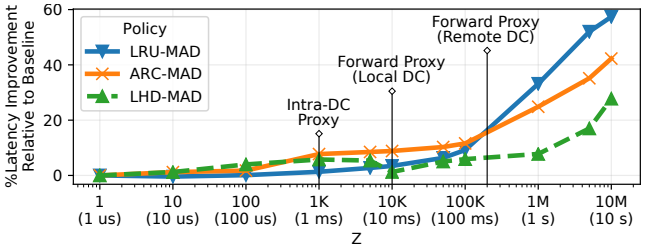


Figure 16: MAD simulations for the CDN Trace.

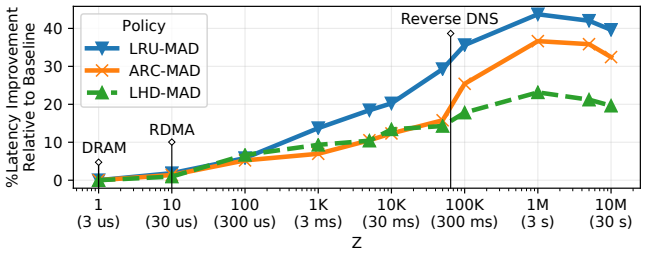


Figure 17: MAD simulations for the Network Trace.

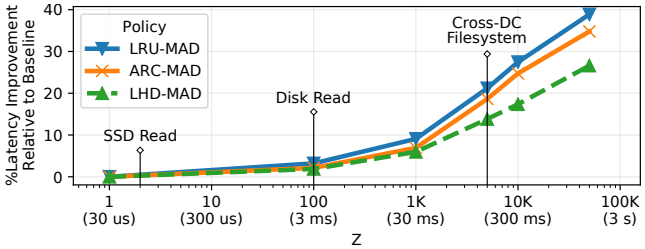


Figure 18: MAD simulations for the Storage Trace.

we expect that request arrival rates, their burstiness, and the number of requested objects may all change in this time; these datapoints are hence little more than an educated guess towards the future.

Can MAD help network switch caches? As discussed in §1, we first observed delayed hits in a programmable switch. Hence, we were surprised to see the *lowest* gains with regard to practical caching scenarios (recall Table 2). The 100ns DRAM latency we worried about had a $Z < 1$ given our 10Gbps network trace and our simulation suggests essentially no performance gains for this scenario from using MAD. The only application where we would expect to see any gains is an IDS with a reverse-DNS lookup, which we would expect to run in the 10s or 100s of milliseconds; the simulation here predicts latency gains of 10-35%. Nonetheless, most IDSes which perform such lookups are not inline, and hence we would not expect to see these latency gains passed on to Internet users whose traffic is intercepted by the IDS.

Looking to the future and very high Z values, we see a tapering off trend which we do *not* observe in the CDN scenario. As discussed in our BELATEDLY results, this tapering off in the network setting is due to flows beginning and ending during the entire Z window; we do not see this trend in the CDN or storage scenarios because objects are much longer lived than a few milliseconds or even seconds. The simulations are hence flawed for network traffic in this regard – in practice, a switch would 'hold' the first SYN packet until its flow context were fetched and subsequent packets would not arrive at the switch until the SYN completed. We leave to future work a more

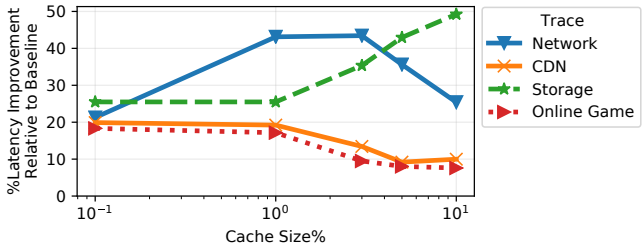


Figure 19: Relative latency difference between LRU-MAD and LRU as a function of the cache size. Using $Z = 100K$.

accurate model of network traffic and Z values where the arrival time of packets is *dependent* on the time it takes to serve the first packet.

Can MAD help distributed storage? Our storage trace has similar results to the CDN result; we see that in the millisecond range we achieve gains between 3-30% from adopting MAD, representing improvements for wide-area or cross-datacenter storage systems. However, when deployed intra-datacenter where network latencies are in the microseconds and system latencies in the low milliseconds, we would expect much more minimal gains of zero to a few percent.

Summary. Overall, our experiments suggest that the systems that would benefit most from MAD toady are CDNs and distributed storage systems with high latencies to the backing store. While switch workloads tend to be more bursty (resulting in higher gains for MAD even at relatively low Z values), few scenarios involve this latency being passed on to end-users.

We note that there are several interesting properties of real systems that are not captured here. For instance, while MAD may only shave off a few *ms* worth of latency on each individual request, some tasks, such as loading web pages, involve *chains* of serialized requests (*e.g.* due to recursive dependencies in HTML or CSS elements [44]); consequently, the overall impact (*e.g.* on page load time) may be more significant. Similarly, fetching large objects from the backing store may require multiple RTTs, exacerbating the effect of delayed hits. Additionally, certain objects *must* be periodically purged from the cache due to TTL expiration (*e.g.* cached DNS entries), introducing an additional layer of complexity in the design of online algorithms. We leave a more detailed investigation of these effects to future work.

5.4 Simulation Results: Analysis

We now present findings that are not tied to any particular system.

Impact of cache sizing: We evaluate how cache size impacts MAD's improvements over traditional caching algorithms. Recall that we measure the cache size as a fraction of the peak number of concurrently-active objects.[14] We calculate the latency improvement of MAD relative to LRU for all four scenarios while keeping Z fixed at $Z = 100k$.

Figure 19 shows the results for cache sizes between 0.1% and 10%. We find that MAD's improvement is around 20% for small caches ($< 1\%$) in the CDN and online gaming scenarios. In the networking scenario, MAD's improvement is between 20% and 43% (we fix Z to demonstrate the effect of cache sizing, but we note that the chosen value is higher than one would expect to see in a networked setting).

[14]Note that our cache size definition is motivated by networking applications where flow state only needs to be tracked for 'active' flows. Caching papers on CDNs and storage systems typically express the cache size as a fraction of the working set [5, 12, 41], which is orders of magnitude larger. The cache size numbers shown in our graphs thus might look comparably large but they are based on a different denominator.

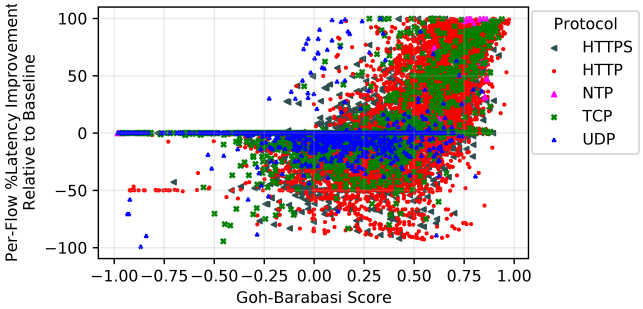


Figure 20: Like BELATEDLY, MAD prioritizes bursty objects.

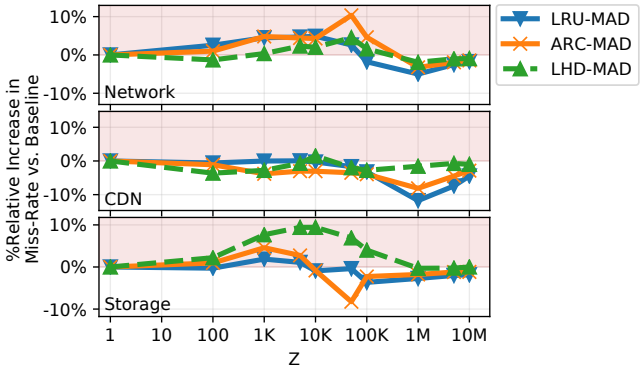


Figure 21: Percent relative change in miss-rate between MAD and various baseline caching algorithms for Network, CDN, and Storage.

Finally, we see that MAD's improvement is highest in the storage scenario, with a 26% to 50% lower latency than LRU.

MAD prioritizes bursty objects, just like BELATEDLY. We described the intuition behind MAD as prioritizing bursty objects, just like BELATEDLY. Nonetheless, we use aggregate delay rather than true burstiness (Goh-Barabasi score) and we weigh aggregate delay against time to next access. Hence it is worth asking – does our intuition about burstiness indeed map on to why MAD is doing well? Figure 20 shows per-object latency gain (or loss) between LRU and LRU-MAD's caching schedule for the Network trace. Much like Figure 10 illustrating BELATEDLY's correlation with burstiness, MAD prioritizes bursty objects as well.

Impact on cache miss-rate. As described in §2.4, latency-minimizing algorithms might in fact *increase* the overall miss-rate. Hence, we quantify the impact of MAD – an algorithm designed to minimize latency – on the overall cache miss-rate (which in turn affects the bandwidth consumption on the link to the backing store). Figure 21 depicts the relative change in miss-rates[15] between MAD and our three baseline algorithms as a function of Z. Regions where MAD *increases* the miss-rate (i.e. performs worse than the baseline) are highlighted in *red*. We find that, across all Z values and choice of baseline algorithms, MAD increases miss-rates by at most 10%[16] for the Network and Storage settings (+1.84% and +1.43% on average), but almost always reduces miss-rates in the CDN setting (−1.89% on average). We conclude that, depending on the workload, there is a tradeoff between optimizing for latency and bandwidth.

[15] $\frac{MR_{(\text{MAD})} - MR_{(\text{Baseline})}}{MR_{(\text{Baseline})}} \times 100\%$

[16]Note that this value represents a *relative* increase in miss-rate compared to the baseline. In our experiments, the *absolute* difference in miss-rates never exceeds 1%.

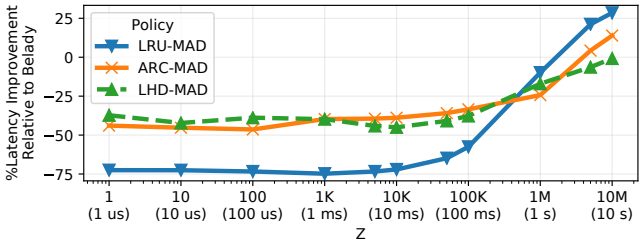


Figure 22: At extremely high latencies, MAD outperforms Belady's algorithm for *CDN*.

MAD can out-perform Belady's algorithm. We were surprised to notice that MAD can out-perform Belady's algorithm. Figure 22 illustrates LRU-MAD, LHD-MAD, and ARC-MAD in the CDN setting, now normalized to the latency achieved by Belady's algorithm rather than their baseline online algorithms.

6 LIMITATIONS AND OPEN QUESTIONS

This paper opens up a broad range of theoretical and practical questions and we are only able to answer some of them.

Our model of caches (§2.2) is very simple and there are many attributes of practical systems that it does not capture; richer and more complex scenarios hence merit additional investigation in both the online and offline setting. For example, our theoretical model does not account for variable backing store latency (although our evaluation does measure this setting), nor does it account for differing object sizes. Both our theory and simulator assume that, once the data fetch delay has passed, all outstanding delayed hits are immediately processed and released, although many systems may instead operate over each response sequentially leading to additional queuing at the cache. Finally, in the online setting, prefetching algorithms may also merit a second look with respect to latency and delayed hits.

Another nagging concern of ours is that we have have yet to prove the hardness of the delayed hits optimization problem. While all indicators point towards a hard problem, a formal proof remains open.

Finally, while the online algorithm we propose in this paper seems to perform well empirically, we now know that it has a poor competitive ratio [40]. Consequently, we don't expect MAD to be the final word on latency-minimizing caching in the presence of delayed hits; indeed, we believe randomized algorithms will yield better results.

7 RELATED WORK

Caching algorithms have received a significant amount of research attention, but the aspect of delayed hits is largely disregarded in the literature. We are not aware of any prior work proposing an analytical model for the delayed hits problem, or designing algorithms targeting delayed hits. Most existing caching algorithms focus on maximizing hit ratios, with significant advances in recent work [5, 12, 13, 29, 38, 55] and excellent surveys of older work [48, 60]. There are two groups of prior work that look at maximizing metrics other than hit ratios.

(1) **Cost-aware online caching algorithms.** This group of algorithms [15, 30–32, 36, 65, 66] seeks to minimize the average cost of misses, where an object's cost models differences in retrieval latencies or computation costs. In this setting, if an object is cached, its next request does not contribute to the overall average cost, but no other requests are affected. This is different from the delayed hits settings where a single caching decision may affect many future requests (to the same object). By assuming independence, cost-aware caching assumes that misses are retrieved before another request to the same object arrives.

(2) **Weighted, general, and other offline caching theory.** This group of algorithms [3, 6, 10, 14, 16–18, 61] considers offline caching problems beyond Belady. Weighted caching is like cost-aware caching, but using offline knowledge [17]. Caching for variable object sizes optimizes hit ratios, but considers objects that require a different number of bits to be stored in cache [3, 10]. General caching generalizes both by considering both weighted and variably-sized objects at once [14, 18]. In general, these problems are NP-hard, except for weighted caching which can also be approximated using a flow formulation.

The architecture community has a rich literature on *implementing* non-blocking caches to handle multiple outstanding misses [1, 8, 35, 43, 53, 58] – a prerequisite for the occurrence of delayed hits. In addition, [49] considers the effect of *correlated* cache misses (different from delayed hits, but in a similar vein) on Memory Level Parallelism (MLP) performance in processors. Finally, we are aware of two prior works [25, 56] which observe improved accuracy when accounting for delayed hits in simulations of processor caches.

8 CONCLUSION

As we look forward to continuous increases in bandwidth and throughput (e.g., in networks, memory, new storage technologies, and CPU-interconnects), access latencies will become larger and larger relative to request inter-arrivals, increasing the likelihood of delayed hits. Indeed, we believe that the problem of delayed hits will surface in almost any caching scenario sooner or later.

Our work constitutes a first step in recognizing and possibly mitigating the increased latencies created by this fundamental trend. Nonetheless, as we discuss in §6, there remain many open questions about incorporating delayed hits into practical caching schemes. We look forward to future work in engaging with delayed hits as we extend the theoretical literature and observe the importance of delayed hits become more apparent in practical systems.

Ethics: This paper raises no ethical concerns.

9 ACKNOWLEDGEMENTS

We thank our shepherd, Anirudh Sivaraman and the anonymous reviewers for their insightful comments. We also thank Jalani Williams, Peter Manohar, and Sai Sandeep Pallerla for helpful discussions regarding the underlying theory, and Nathan Beckmann for his feedback and help with implementing LHD. We are also grateful to the Parallel Data Lab (PDL) at CMU for providing compute resources to us. This work was funded by NSF Grants 1700521 and 2007733, and supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

[1] K. Aasaraai and A. Moshovos. An efficient non-blocking data cache for soft processors. In *2010 International Conference on Reconfigurable Computing and FPGAs*, pages 19–24, 2010.
[2] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice hall, 1993.
[3] Susanne Albers, Sanjeev Arora, and Sanjeev Khanna. Page replacement for general caching problems. In *SODA*, pages 31–40, 1999.
[4] Wisnu Anggoro and John Torjo. *Boost. Asio C++ Network Programming*. Packt Publishing Ltd, 2015.

[5] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. Lhd: Improving hit rate by maximizing hit density. In *USENIX NSDI.*, pages 1–14, 2018.

[6] Nathan Beckmann, Phillip B Gibbons, Bernhard Haeupler, and Charles McGuffey. Writeback-aware caching. In *Symposium on Algorithmic Principles of Computer Systems*, pages 1–15. SIAM, 2020.

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

[8] Samson Belayneh and David R. Kaeli. A discussion on non-blocking/lockup-free caches. *SIGARCH Comput. Archit. News*, 24(3):18–25, June 1996.

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

[10] Daniel S. Berger, Nathan Beckmann, and Mor Harchol-Balter. Practical bounds on optimal caching with variable object sizes. *ACM POMACS*, 2(2):32, 2018.

[11] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. Robinhood: Tail latency aware caching–dynamic reallocation from cache-rich to cache-poor. In *USENIX OSDI*, pages 195–212, 2018.

[12] Daniel S. Berger, Ramesh Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a cdn. In *USENIX NSDI*, pages 483–498, March 2017.

[13] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. Hyperbolic caching: Flexible caching for web applications. In *USENIX ATC*, pages 499–511, 2017.

[14] Niv Buchbinder, Joseph Seffi Naor, et al. The design of competitive online algorithms via a primal–dual approach. *Foundations and Trends in Theoretical Computer Science*, 3(2–3):93–263, 2009.

[15] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Usenix symposium on internet technologies and systems*, volume 12, pages 193–206, 1997.

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

[17] Marek Chrobak, H Karloof, Tom Payne, and S Vishwnathan. New ressults on server problems. *SIAM Journal on Discrete Mathematics*, 4(2):172–181, 1991.

[18] Marek Chrobak, Gerhard J Woeginger, Kazuhisa Makino, and Haifeng Xu. Caching is hard—even in the fault model. *Algorithmica*, 63(4):781–794, 2012.

[19] Jeff Dean and R. Colin Scott. Numbers every programmer should know. https://people.eecs.berkeley.edu/ rcs/research/interactive_latency.html.

[20] John Dilley and Martin Arlitt. Improving proxy cache performance: Analysis of three replacement policies. *IEEE Internet Computing*, 3(6):44–50, 1999.

[21] John Dilley, Bruce Maggs, Jay Parikh, Harald Prokop, Ramesh Sitaraman, and Bill Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[22] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *USENIX NSDI*, pages 371–384, 2013.

[23] A. Feldmann, R. Caceres, F. Douglis, G. Glass, and M. Rabinovich. Performance of web proxy caching in heterogeneous bandwidth environments. In *IEEE INFOCOM*, volume 1, pages 107–116 vol.1, 1999.

[24] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *USENIX NSDI*, pages 51–66, 2018.

[25] Davy Genbrugge and Lieven Eeckhout. Memory data flow modeling in statistical simulation for the efficient exploration of microprocessor design spaces. *IEEE Transactions on Computers*, 57(1):41–54, 2007.

[26] K.-I. Goh and A.-L. Barabási. Burstiness and memory in complex systems. *EPL (Europhysics Letters)*, 81(4):48002, jan 2008.

[27] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 4 edition, 2011.

[28] SNIA IOTTA. Microsoft production server traces, 2011.

[29] Akanksha Jain and Calvin Lin. Back to the future: leveraging belady's algorithm for improved cache replacement. In *ACM/IEEE ISCA*, pages 78–89, 2016.

[30] Jaeheon Jeong and Michel Dubois. Cache replacement algorithms with nonuniform miss costs. *IEEE Transactions on Computers*, 55(4):353–365, 2006.

[31] Shudong Jin and Azer Bestavros. Popularity-aware greedy dual-size web proxy caching algorithms. In *IEEE ICDCS*, pages 254–261, 2000.

[32] Shudong Jin and Azer Bestavros. Greedydual⋆ web caching algorithm: exploiting the two sources of temporal locality in web request streams. *Computer Communications*, 24(2):174–183, 2001.

[33] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. Generic External Memory for Switch Data Planes. In *ACM HotNets*, pages 1–7, 2018.

[34] Rupa Krishnan, Harsha V Madhyastha, Sridhar Srinivasan, Sushant Jain, Arvind Krishnamurthy, Thomas Anderson, and Jie Gao. Moving beyond end-to-end path information to optimize cdn performance. In *ACM IMC*, pages 190–201, 2009.

[35] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ACM/IEEE ISCA*, page 81–87, Washington, DC, USA, 1981. IEEE Computer Society Press.

[36] Conglong Li and Alan L Cox. Gd-wheel: a cost-aware replacement policy for key-value stores. In *EUROSYS*, pages 1–15, 2015.

[37] Shang Li, Dhiraj Reddy, and Bruce Jacob. A performance & power comparison of modern high-speed dram architectures. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '18, page 341–353, New York, NY, USA, 2018. Association for Computing Machinery.

[38] Suoheng Li, Jie Xu, Mihaela van der Schaar, and Weiping Li. Popularity-driven content caching. In *IEEE INFOCOM*, pages 1–9, 2016.

[39] Shuang Liang, Ke Chen, Song Jiang, and Xiaodong Zhang. Cost-aware caching algorithms for distributed storage servers. In Andrzej Pelc, editor, *Distributed Computing*, pages 373–387, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[40] Peter Manohar and Jalani Williams. Lower Bounds for Caching with Delayed Hits. arXiv:2006.00376 [cs.DS], 2020.

[41] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *USENIX FAST*, pages 115–130, 2003.

[42] Matthew K Mukerjee, David Naylor, Junchen Jiang, Dongsu Han, Srinivasan Seshan, and Hui Zhang. Practical, real-time centralized control for CDNM-based live video delivery. In *ACM SIGCOMM*, pages 311–324, 2015.

[43] A. Musa, Y. Sato, T. Soga, R. Egawa, H. Takizawa, K. Okabe, and H. Kobayashi. Effects of mshr and prefetch mechanisms on an on-chip cache of the vector architecture. In *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 335–342, 2008.

[44] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *USENIX NSDI*, March 2016.

[45] Gurobi Optimization. Inc.,"gurobi optimizer reference manual," 2015, 2014.

[46] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23-24):2435–2463, 1999.

[47] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *USENIX NSDI*, pages 117–130, 2015.

[48] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Computing Surveys*, 35(4):374–398, 2003.

[49] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for mlp-aware cache replacement. In *ACM/IEEE ISCA*, pages 167–178, 2006.

[50] Prabhakar Raghavan and Clark D Tompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, 1987.

[51] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *USENIX OSDI*, pages 401–417, 2016.

[52] Boris Schäling. *The boost C++ libraries*. Boris Schäling, 2011.

[53] James Edward Sicolo. A multiported nonblocking cache for a superscalar uniprocessor. Master's thesis, University of Illinois at Urbana-Champaign, 1992.

[54] Abraham Silberschatz, Greg Gagne, and Peter B Galvin. *Operating system concepts*. Wiley, 2018.

[55] Zhenyu Song, Daniel S. Berger, and Lloyd Wyatt LI, Kai. Learning relaxed belady for content distribution network caching. In *USENIX NSDI*, 2020.

[56] Edward S Tam. *Improving cache performance via active management*. PhD thesis, University of Michigan, 1999.

[57] Eric Torng. A unified analysis of paging and caching. *Annual Symposium on Foundations of Computer Science - Proceedings*, 08 1995.

[58] J. Tuck, L. Ceze, and J. Torrellas. Scalable cache miss handling for high memory-level parallelism. In *ACM/IEEE MICRO*, pages 409–422, 2006.

[59] Colby Walsworth, Emile Aben, K Claffy, and D Andersen. The caida anonymized 2019 internet traces, 2019.

[60] Jia Wang. A survey of web caching schemes for the internet. *ACM SIGCOMM Computer Communication Review*, 29(5):36–46, 1999.

[61] Justin Wang, Benjamin Berg, Daniel S Berger, and Siddhartha Sen. Maximizing page-level cache hit ratios in largeweb services. *ACM SIGMETRICS Performance Evaluation Review*, 46(2):91–92, 2019.

[62] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *USENIX OSDI*, pages 307–320, 2006.

[63] Wikipedia. Reverse proxy. https://en.wikipedia.org/wiki/Reverse_proxy.

[64] Maurice V Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions Electronic Computers*, 14(2):270–271, 1965.

[65] Neal E Young. On-line caching as cache size varies. In *ACM SODA*, 1991.

[66] Neal E Young. On-line file caching. *Algorithmica*, 33(3):371–383, 2002.

A APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A.1 Latency Minimization Problem

In this section we give a formal definition of the latency minimization problem for caching with delayed hits.

Recall that we consider a cache of size C and M objects indexed by $i \in [M]$. We are given a sequence object requests, where $\sigma(T)$ denotes the object requested at timestep T with $T = 0,1,\ldots,N$.

We use the following quantities to describe the state of the system at the beginning of each timestep T. For each object i, let

$$x_0^{(i)}(T) = \mathbb{1}_{\{\text{object } i \text{ is in the cache at } T\}}, \tag{4}$$

$$x_\tau^{(i)}(T) = \mathbb{1}_{\{\text{object } i \text{ was requested at } T-(Z+1-\tau) \text{ and the request has not been resolved}\}} \quad \tau = 1,\ldots,Z. \tag{5}$$

Here, when an object i is requested but cannot be resolved immediately, we say that we put it in a queue. So (5) describes the state of the queue for i.

We specify a cache schedule using the following decision variables. Let $a_i(T)$ be defined by

$$a_i(T) = \begin{cases} 1 & \text{if object } i \text{ is admitted to cache at } T, \\ -1 & \text{if object } i \text{ is evicted from cache at } T, \\ 0 & \text{if no action is taken on object } i \text{ at } T. \end{cases} \tag{6}$$

To make sure $a_i(T)$ with $i \in [M], T = 0,1,\ldots,N$ form a valid cache schedule, we enforce the following constraints for each object $i \in [M]$ and timestep $T = 0,1,\ldots,N$:

- An object can be admitted only when its data arrives:
$$\mathbb{1}_{\{a_i(T)=1\}} \le x_1^{(i)}(T). \tag{7}$$
- An object can be evicted only when it is already in the cache:
$$\mathbb{1}_{\{a_i(T)=-1\}} \le x_0^{(i)}(T) \tag{8}$$
- The schedule should guarantee that the number of objects in the cache is no larger than the cache size C:
$$\sum_{i\in[M]} x_0^{(i)}(T) \le C. \tag{9}$$
Although it seems that this is a constraint on the state, it is in fact a constraint on the cache schedule since the state at the current timestep is determined by the past decisions. This will become clear after we describe the relation between the state and the schedule next.

With the notation above, we can write out how the system state evolves over time as follows:

- The data that just arrived resolves the requests for the same object in the queue, and other requests move forward in queue:
$$x_\tau^{(i)}(T+1) = x_{\tau+1}^{(i)}(T)\cdot(1-x_1^{(i)}(T)), \quad i \in [M], \tau = 1,\ldots,Z-1, T = 0,1,\ldots,N-2. \tag{10}$$
- The admission or eviction of an object changes the state in the cache:
$$x_0^{(i)}(T+1) = x_0^{(i)}(T) + a_i(T), \quad i \in [M], T = 0,1,\ldots,N-2. \tag{11}$$
- The new request comes in and is added to the queue if the requested object is not in the cache:
$$x_Z^{(i)}(T+1) = \mathbb{1}_{\{\sigma(T)=i\}}\cdot(1-x_0^{(i)}(T+1)), \quad i \in [M], T = 0,1,\ldots,N. \tag{12}$$

It can be proven that the state that evolves according to the dynamics above satisfies that for any $T = 0,1,\ldots,N-2$,

$$x_0^{(i)}(T) + \mathbb{1}_{\{\sum_{\tau=1}^{Z} x_\tau^{(i)}(T)>0\}} \le 1. \tag{13}$$

This inequality states the fact that if object i is in the cache, then there will not be requests for i in the queue, and if there are requests of object i in the queue, then i is not in the cache.

At timestep T, object $\sigma(T)$ is requested. If it is not in the cache nor requested during the past Z timesteps, it will trigger a sequence of delayed hits when $\sigma(T)$ is requested again during the next Z timesteps. Therefore, the total latency can be written as:

$$\sum_{T=0}^{N-2} x_Z^{(\sigma(T))}(T+1)\cdot\prod_{\tau=1}^{Z-1}\left(1-x_\tau^{(\sigma(T))}(T+1)\right)\cdot\sum_{t=0}^{Z-1}\mathbb{1}_{\{\sigma(T+t)=\sigma(T)\}}\cdot(Z-t). \tag{14}$$

Then the *latency minimization problem* is to find the cache schedule subject to the constraints (7)–(9) such that the resulting states minimize the total latency in (14).

A.2 Proof of Theorem 1

We first give some notation for the flow variables and state the MCMCF problem with the notation. We define a flow variable for each object on each edge, which takes values from $\{0,1\}$ and represents the fraction of flow for the object routed along that edge. In particular, we define the flow variables below:

$$f_{\text{mem}}^{(i)}(T):\ \text{object } i \text{ along edge } (V_{\text{mem},T}, V_{\text{cch},T+Z}),\ T = 0,1,\ldots,N-1-Z;$$

$$f_{\text{cch}}^{(i)}(T):\ \text{object } i \text{ along edge } (V_{\text{cch},T}, V_{\text{cch},T+1}),\ T = Z,1+Z,\ldots,N-2+Z;$$

$$f_{\text{evict}}^{(i)}(T):\ \text{object } i \text{ along edge } (V_{\text{cch},T}, V_{\text{next},i}^{(T)}),\ T = Z,1+Z,\ldots,N-1+Z.$$

Note that $f_{\text{mem}}^{(i)}(T)$ is always 0 if $i \neq \sigma(T)$ due to the infinite cost. Similarly, the flow variable for object j along edge $(V_{\text{cch},T}, V_{\text{next},i}^{(T)})$ with $j \neq i$ is also always 0. Here our formulation is a so-called 'single-path routing' formulation, i.e., the flow variables are either 0 or 1 and they together represent a path for each object. Additionally, for convenience, for each vertex $V_{\text{mem},T}$, we use $\mathcal{P}^{(j)}(V_{\text{mem},T})$ to denote the set of vertices in the top row that have outgoing edges to $V_{\text{mem},T}$ associated with object j. Our goal is to minimize the following objective function:

$$\sum_{T=0}^{N-1} c^{(\sigma(T))}(V_{\text{mem},T}, V_{\text{cch},T+Z})\cdot f_{\text{mem}}^{(\sigma(T))}(T), \tag{15}$$

where $c^{(\sigma(T))}(V_{\text{mem},T}, V_{\text{cch},T+Z})$ is the latency cost in (2). The minimization problem is subject to the following constraints for each object i:

- **Link capacity:**
$$f_{\text{mem}}^{(\sigma(T))}(T) \le 1, T = 0,1,\ldots,N-1, \tag{16}$$
$$\sum_{i\in[M]} f_{\text{cch}}^{(i)}(T) \le C, T = Z,1+Z,\ldots,N-2+Z, \tag{17}$$
$$f_{\text{evict}}^{(i)}(T) \le 1, T = Z,1+Z,\ldots,N-1+Z. \tag{18}$$

Here (17) models the constraint that we can have at most C objects in the cache. The constraints (16) and (18) are automatically satisfied.

- **Flow conservation:**

$$f^{(i)}_{\text{mem}}(T_i)=1, \text{ where } V_{\text{mem},T_i} \text{ is the source of } i, \tag{19}$$

$$\text{total incoming flow to } V^{(i)}_{\text{sink}} \text{ is 1}, \tag{20}$$

$$\sum_{t:\, V_{\text{cch},t}\in\mathcal{P}^{(i)}(V_{\text{mem},T})} f^{(i)}_{\text{evict}}(t)=f^{(i)}_{\text{mem}}(T), i=\sigma(T), T>T_i, \tag{21}$$

$$f^{(i)}_{\text{cch}}(T-1)+f^{(i)}_{\text{mem}}(T-Z)=f^{(i)}_{\text{cch}}(T)+f^{(i)}_{\text{evict}}(T), \quad T=Z,1+Z,\ldots,N-1+Z. \tag{22}$$

Here the constraints (19) and (20) at sources and sinks are straightforward. The constraint (21) is a flow conservation constraint at vertex $V_{\text{mem},T}$. It implies that if object i was evicted from the cache before T and has not been requested since, then its data will be fetched from the backing store to the cache when it is requested at T. The constraint (22) is a flow conservation constraint at vertex $V_{\text{cch},T}$. Let us set $f^{(i)}_{\text{cch}}(Z-1)=f^{(i)}_{\text{cch}}(N-1+Z)=0$ so (22) is valid at $T=Z$ and $T=N-1+Z$. This constraint guarantees the obvious requirement that an object is either in the cache or not in the cache.

Both the MCMCF problem and the latency minimization problems are optimization problems. To show the equivalence of these two problems, we first show in Lemma 1 below that the feasible set of flow variables is 'equivalent' to the feasible set of caching schedules. In particular, from any feasible cache schedule, we can define a set of flow variables that are also feasible for the MCMCF problem; conversely, given any feasible set of assignments to flow variables, we can define a feasible cache schedule. Once we have this bijection, we can show that the objective functions of these two problems are the same. With equivalent feasible sets and objective functions, the MCMCF problem and the latency minimization problem are thus equivalent.

Lemma. *Given a sequence of object requests, there is a bijection between the set of feasible flow variables and the set of feasible cache schedules.*

Proof of Lemma 1. We first prove that any feasible cache schedule defines a set of feasible flow variables. Let $a_i(T), i\in[M], T=0,1,\ldots,N$ be a feasible cache schedule. We show that the flow variables defined below are feasible:

$$f^{(i)}_{\text{mem}}(T)=x^{(i)}_Z(T+1)\cdot\prod_{\tau=1}^{Z-1}\left(1-x^{(i)}_\tau(T+1)\right), \tag{23}$$

$$f^{(i)}_{\text{cch}}(T)=x^{(i)}_0(T)\cdot\mathbb{1}_{\{a_i(T)=0\}}+x^{(i)}_1(T)\cdot\mathbb{1}_{\{a_i(T)=1\}}, \tag{24}$$

$$f^{(i)}_{\text{evict}}(T)=x^{(i)}_0(T)\cdot\mathbb{1}_{\{a_i(T)=-1\}}+x^{(i)}_1(T)\cdot\mathbb{1}_{\{a_i(T)=0\}}. \tag{25}$$

Let us first consider the capacity constraints (16)–(18). It is easy to check that the constraints (16) and (18) are satisfied. Now we check the constraint (17). By the definition of $f^{(i)}_{\text{cch}}(T)$ in (24),

$$\sum_{i\in[M]} f^{(i)}_{\text{cch}}(T) = \sum_{i\in[M]}\left(x^{(i)}_0(T)\cdot\mathbb{1}_{\{a_i(T)=0\}}+x^{(i)}_1(T)\cdot\mathbb{1}_{\{a_i(T)=1\}}\right) \tag{26}$$

When $a_i(T)=-1$, the summand in (26) is 0; when $a_i(T)=0$, the summand equals $x^{(i)}_0(T)=x^{(i)}_0(T+1)$; otherwise, when $a_i(T)=1$ and $x^{(i)}_1(T)=1$, object i will be admitted to the cache so $x^{(i)}_0(T+1)=1$. Combining these cases, we have that the summand in (26) is always no larger than $x^{(i)}_0(T+1)$. Thus,

$$\sum_{i\in[M]} f^{(i)}_{\text{cch}}(T)\le\sum_{i\in[M]} x^{(i)}_0(T+1)\le C, \tag{27}$$

and the constraint (17) is satisfied.

Next let us consider the flow conservation constraints (19)–(22). It is easy to check that the constraints (19) and (20) are satisfied.

For the constraint (21), let $i=\sigma(T)$. Let $t^*=\max\{t:\, t<T, \sigma(t)=i\}$. Then one can check that $\{t:\, V_{\text{cch},t}\in\mathcal{P}^{(i)}(V_{\text{mem},T})\}=\{t^*+1, t^*+2,\ldots,T\}$. So it suffices to show that

$$\sum_{t=t^*+1}^{T}\left(x^{(i)}_0(t)\cdot\mathbb{1}_{\{a_i(t)=-1\}}+x^{(i)}_1(t)\cdot\mathbb{1}_{\{a_i(t)=0\}}\right) = x^{(i)}_Z(T+1)\cdot\prod_{\tau=1}^{Z-1}\left(1-x^{(i)}_\tau(T+1)\right). \tag{28}$$

First, consider the case where $x^{(i)}_1(t)=1$ for some $t^*<t\le T$. Then we must have $t\le t^*+Z$ since there is no request for object i after t^* and before T. This arrival at t will resolve all the requests for i in the queue (if there exist any). We observe that $x^{(i)}_0(u)=0$ for $t^*<u\le t$ by (13) and $x^{(i)}_{t+1-u}(u)=x^{(i)}_1(t)=1$. Also $x^{(i)}_1(u)=0$ for $t^*<u<t$ since otherwise it would have resolved the request and thus results in no data arrival at t. If $a_i(t)=0$, then the data is not admitted to the cache. Also there is no request for object i on or after t (before T). So $x^{(i)}_1(u)=x^{(i)}_0(u)=0$ for $t<u\le T$. Then when the request for i comes in at T, it sees nothing in the cache nor the queue. So by the dynamics in (12), we have $x^{(i)}_Z(T+1)=1$. Therefore, the right-hand-side (RHS) of (28) is equal to 1, which is equal to the left-hand-side (LHS). For the case that $a_i(t)=1$, the data is admitted to the cache at t. There can be at most one eviction after t and no later than T (two evictions require data arrival in between). If there is no eviction, then the LHS is 0. The RHS is also 0 since the request for i at T will not be put in the queue and thus $x^{(i)}_Z(T+1)=0$. If there is an eviction at some u with $t<u\le T$, then all the summands except $x^{(i)}_0(u)\cdot\mathbb{1}_{a_i(u)=-1}$ on the LHS are 0. So the LHS is equal to 1. The RHS is also equal to 1 since the request for i at T sees nothing in the cache nor the queue. In summary, (28) holds when $x^{(i)}_1(t)=1$ for some $t^*<t\le T$.

Next, consider the case where $x^{(i)}_1(t)=0$ for all t with $t^*<t\le T$. In this case there is no data arrival for object i during the whole time period. Then again there can be at most one eviction. Suppose there is no eviction for all t with $t^*<t\le T$. Then the LHS of (28) is 0. In this case, object i is either in the cache for all timestep t with $t^*<t\le T$ or it is not in the cache for all t with $t^*<t\le T$. If it is in the cache all the time, then the RHS is also 0 since $x^{(i)}_Z(T+1)=0$. If it is always not in the cache, then $T<t^*+Z$ since the request at t^* is put in the queue and arrive at t^*+Z, but we have assumed that $x^{(i)}_1(t)=0$ for all t with $t^*<t\le T$. However, $T<t^*+Z$ implies that $x^{(i)}_{t^*+Z-T}(T+1)=x^{(i)}_Z(t^*+1)=1$, which implies that the RHS of (28) is 0. Therefore, for the case of no eviction, LHS and RHS are equal. Suppose there is an eviction at some t with $t^*<t\le T$. Then the LHS of (28) is equal to 1. Since we have assumed that $x^{(i)}_1(t)=0$ for all t with $t^*<t\le T$, object i cannot reenter the cache after the eviction. So $x^{(i)}_\tau(T)=0$ for $0\le\tau\le Z$. Then the request for i at T will be added to the queue, so the RHS of (28) is equal to 1. Therefore, LHS and RHS are also equal in this case.

Combining the arguments above, we have shown that the flow conservation constraint (21) is satisfied.

Now let us check the constraint (22), i.e., we want to show that

$$\begin{aligned}
&\underbrace{x_0^{(i)}(T-1)\cdot\mathbb{1}_{\{a_i(T-1)=0\}}}_{\text{Term (L1)}}+\underbrace{x_1^{(i)}(T-1)\cdot\mathbb{1}_{\{a_i(T-1)=1\}}}_{\text{Term (L2)}}\\
&\underbrace{+x_Z^{(i)}(T+1-Z)\cdot\prod_{\tau=1}^{Z-1}\left(1-x_\tau^{(i)}(T+1-Z)\right)}_{\text{Term (L3)}}\\
=&\underbrace{x_0^{(i)}(T)\cdot\mathbb{1}_{\{a_i(T)=0\}}}_{\text{Term (R1)}}+\underbrace{x_1^{(i)}(T)\cdot\mathbb{1}_{\{a_i(T)=1\}}}_{\text{Term (R2)}}\\
&\underbrace{+x_0^{(i)}(T)\cdot\mathbb{1}_{\{a_i(T)=-1\}}}_{\text{Term (R3)}}+\underbrace{x_1^{(i)}(T)\cdot\mathbb{1}_{\{a_i(T)=0\}}}_{\text{Term (R4)}}.
\end{aligned}\tag{29}$$

We start by discussing different cases of Term (L3). Suppose (L3) = 1. Then

$$x_Z^{(i)}(T+1-Z)=1, x_\tau^{(i)}(T+1-Z)=0, \tau=1,2,\ldots,Z-1. \tag{30}$$

In this case, $x_0^{(i)}(T+1-Z)=0$ by (13) and object i will not arrive until timestep T. So $x_0^{(i)}(t)=0$ for $t=T+1-Z,T+2-Z,\ldots,T$, and $x_1^{(i)}(T)=1$. Then (L1) = (L2) = (R1) = (R3) = 0 and (R2) + (R4) = 1. Therefore (29) holds.

Now suppose (L3) = 0. Then either $x_\tau^{(i)}(T+1-Z)=1$ for some $1\le\tau\le Z-1$ or $x_\tau^{(i)}(T+1-Z)=0$ for all $1\le\tau\le Z-1$ and $x_Z^{(i)}(T+1-Z)=0$.

- Suppose it is the former case. Then let t^* be the earliest time with $T+1-Z\le t^*\le T-1$ such that $x_1^{(i)}(t^*)=1$. In fact, since all the requests in queue will be resolved when the data arrives, t^* is also the only time between $T-Z+1$ and T such that $x_1^{(i)}(t^*)=1$. So (R2) = (R4) = 0. Also, $x_0^{(i)}(t)=0$ for all t with $T-Z\le t\le t^*$. If $t^*=T-1$, then (L1) = 0 and (L2) = (R1) + (R3). If $t^*<T-1$, then (L2) = 0. Since $x_0^{(i)}(T)=x_0^{(i)}(T-1)+a_i(T-1)$, we have (L1) = (R1) + (R3). So (29) holds.
- Suppose it is the latter case, i.e., $x_\tau^{(i)}(T+1-Z)=0$ for all $1\le\tau\le Z-1$ and $x_Z^{(i)}(T+1-Z)=0$. Then $x_1^{(i)}(t)=0$ for $T+1-Z\le t\le T$. So (L2) = (R2) = (R4) = 0. Similar to the former case, it can be shown that (L1) = (R1) + (R3).

Combining the arguments above, we have shown that (29) always holds and thus the flow conservation constraint (22) is satisfied.

Now we prove the other direction of the lemma, i.e., we prove that any feasible set of flow variables define a feasible cache schedule. Let $f_{\text{mem}}^{(i)}(T), f_{\text{cch}}^{(i)}(T), f_{\text{evict}}^{(i)}(T)$ be a set of feasible flow variables. We show that the cache schedule defined below is feasible. For each timestep $T\ge Z$,

$$a_i(T)=\begin{cases}1 & \text{when } f_{\text{mem}}^{(i)}(T-Z)=1,\ f_{\text{cch}}^{(i)}(T-1)=0,\\ & \quad\text{and } f_{\text{evict}}^{(i)}(T)=0,\\ -1 & \text{when } f_{\text{cch}}^{(i)}(T-1)=1 \text{ and } f_{\text{evict}}^{(i)}(T)=1,\\ 0 & \text{otherwise.}\end{cases}\tag{31}$$

For T with $0\le T<Z$, let $a_i(T)=0$, which is always feasible. Let $x_\tau^{(i)}(T)$ with $i\in[M], \tau=0,\ldots,Z$ be the state of the system as defined in (4) and (5) under this cache schedule in (31) . To show that this schedule is feasible, we first prove the following claims.

Claim 1. *For any object i and any timestep $T\ge Z$,*

$$x_0^{(i)}(T)=f_{cch}^{(i)}(T-1). \tag{32}$$

Claim 2. *For any object i and any $T\ge 0$,*

$$f_{mem}^{(i)}(T)=x_Z^{(i)}(T+1)\cdot\prod_{\tau=1}^{Z-1}\left(1-x_\tau^{(i)}(T+1)\right). \tag{33}$$

We note that in Claim 2,

$$\begin{aligned}
&x_Z^{(i)}(T+1)\cdot\prod_{\tau=1}^{Z-1}\left(1-x_\tau^{(i)}(T+1)\right)=1\\
\Leftrightarrow\ &x_Z^{(i)}(T+1)=1, x_\tau^{(i)}(T+1)=0 \text{ for all } \tau=0,1,\ldots,Z-1\\
\Leftrightarrow\ &x_1^{(i)}(T+Z)=1.
\end{aligned}$$

Therefore, it is equivalent to $f_{\text{mem}}^{(i)}(T)=x_1^{(i)}(T+Z)$.

We prove both claims by induction.

Proof of Claim 1. **Base case.** When $T=Z$, $x_0^{(i)}(T)=0$ for all i since we start from an empty cache and $a_i(u)=0$ for $0\le u<Z$. We have also defined $f_{\text{cch}}^{(i)}(Z-1)$ to be 0 as an custom. So $x_0^{(i)}(Z)=f_{\text{cch}}^{(i)}(Z-1)$.

Induction step. Assume that for some $T\ge Z$, $x_0^{(i)}(T)=f_{\text{cch}}^{(i)}(T-1)$. We want to show that $x_0^{(i)}(T+1)=f_{\text{cch}}^{(i)}(T)$. Note that by the system dynamics in (11), we have that $x_0^{(i)}(T+1)=x_0^{(i)}(T)+a_i(T)$.

We consider the different cases of $a_i(T)$.

- If $a_i(T)=1$, then by (31), $f_{\text{cch}}^{(i)}(T-1)=0$, $f_{\text{mem}}^{(i)}(T-Z)=1$ and $f_{\text{evict}}^{(i)}(T)=0$. By the flow conservation at $V_{\text{cch},T}$, we have that $f_{\text{cch}}^{(i)}(T)=1$. By the induction assumption, $x_0^{(i)}(T)=f_{\text{cch}}^{(i)}(T-1)$. Then $x_0^{(i)}(T+1)=x_0^{(i)}(T)+a_i(T)=1$. So $x_0^{(i)}(T+1)=f_{\text{cch}}^{(i)}(T)$.
- If $a_i(T)=-1$, then by (31), $f_{\text{cch}}^{(i)}(T-1)=1$ and $f_{\text{evict}}^{(i)}(T)=1$. Due to the unit demand of each object, it is not hard to show that the total incoming flow an object to a vertex is at most 1. Specifically, consider the vertex $V_{\text{cch},T}$. Then $f_{\text{cch}}^{(i)}(T-1)+f_{\text{mem}}^{(i)}(T-Z)\le 1$. So $f_{\text{mem}}^{(i)}(T-Z)=0$. By the flow conservation at $V_{\text{cch},T}$, $f_{\text{cch}}^{(i)}(T)=0$. Since $x_0^{(i)}(T+1)=x_0^{(i)}(T)+a_i(T)=0$, we have $x_0^{(i)}(T+1)=f_{\text{cch}}^{(T)}$.
- If $a_i(T)=0$, by (31), we have the following possibilities:

$$f_{\text{cch}}^{(i)}(T-1)=0, f_{\text{mem}}^{(i)}(T-Z)=1, \tag{34}$$
$$f_{\text{cch}}^{(i)}(T)=0, f_{\text{evict}}^{(i)}(T)=1; \tag{35}$$
$$\text{or } f_{\text{cch}}^{(i)}(T-1)=1, f_{\text{mem}}^{(i)}(T-Z)=0, \tag{36}$$
$$f_{\text{cch}}^{(i)}(T)=1, f_{\text{evict}}^{(i)}(T)=0; \tag{37}$$
$$\text{or } f_{\text{cch}}^{(i)}(T-1)=0, f_{\text{mem}}^{(i)}(T-Z)=0, \tag{38}$$
$$f_{\text{cch}}^{(i)}(T)=0, f_{\text{evict}}^{(i)}(T)=0. \tag{39}$$

For all the possibilities, $f_{\text{cch}}^{(i)}(T)=f_{\text{cch}}^{(i)}(T-1)$. Since $x_0^{(i)}(T+1)=f_{\text{cch}}^{(i)}(T-1)+a_i(T)$, we have $x_0^{(i)}(T+1)=f_{\text{cch}}^{(i)}(T)$.

This completes the proof of Claim 1.

Proof of Claim 2. **Base case.** When $T=0$, by flow conservation, $f_{\text{mem}}^{(i)}(0)=1$ if and only if $\sigma(0)=i$. Since we start from an empty cache and $a_i(u)=0$ for $0\le u<Z$, by the state dynamics (10)–(12), $x_\tau^{(i)}(1)=0$ for all $i\in[M]$ and $\tau=0,1,\ldots,Z$, and $x_Z^{(i)}(1)=1$ for $i=\sigma(0)$ and 0 for other objects. So (33) holds for $T=0$.

Induction step. Assume that for each timestep u with $0 \le u \le T$,

$$f^{(i)}_{\text{mem}}(u) = x^{(i)}_Z(u+1) \cdot \prod_{\tau=1}^{Z-1} \left(1 - x^{(i)}_\tau(u+1)\right). \tag{40}$$

We want to show

$$f^{(i)}_{\text{mem}}(T+1) = x^{(i)}_Z(T+2) \cdot \prod_{\tau=1}^{Z-1} \left(1 - x^{(i)}_\tau(T+2)\right). \tag{41}$$

First, it is not hard to see that

$$\begin{aligned} & x^{(i)}_Z(u+1) \cdot \prod_{\tau=1}^{Z-1} \left(1 - x^{(i)}_\tau(u+1)\right) = 1 \\ \Leftrightarrow\ & x^{(i)}_Z(u+1) = 1, x^{(i)}_\tau(u+1) = 0 \text{ for all } \tau = 0,1,\ldots,Z-1 \\ \Leftrightarrow\ & x^{(i)}_1(u+Z) = 1. \end{aligned}$$

Therefore, the induction assumption (40) is equivalent to $f^{(i)}_{\text{mem}}(u) = x^{(i)}_1(u+Z)$.

Observe that the RHS of (41) is equal to 1 if and only if $\sigma(T+1) = i$ and $x^{(i)}_\tau(T+2) = 0$ for all $\tau = 1,2,\ldots,Z-1$. Then (41) is trivially true for $i \neq \sigma(T+1)$. So it suffices to focus on the case where $i = \sigma(T+1)$.

If $V_{\text{mem},T+1}$ is a source vertex of object i, then $f^{(i)}_{\text{mem}}(T+1) = 1$. By flow conservation, $f^{(i)}_{\text{mem}}(u) = f^{(i)}_{\text{cch}}(u) = f^{(i)}_{\text{evict}}(u) = 0$ for $0 \le u \le T$. Then $a_i(u) = 0$ for all $0 \le u \le T$. So by the dynamics in the system, at $T+1$ the request will see that i is not in the cache and the queue for i is also empty. Then the RHS of (41) is equal to 1.

When $V_{\text{mem},T+1}$ is not a source vertex, let t^* be the last time object i was requested, i.e.,

$$t^* = \max\{t : t < T+1, \sigma(t) = i\}. \tag{42}$$

Suppose $f^{(i)}_{\text{mem}}(T+1) = 1$. Then by flow conservation at $V_{\text{mem},T+1}$, $f^{(i)}_{\text{evict}}(t) = 1$ for some t with $t^* < t \le T+1$. Let t' be the latest timestep with $t' \le t$ such that $f^{(i)}_{\text{mem}}(t'-Z) = 1$. By the induction assumption, $x^{(i)}_Z(t'-Z+1) = 1$ and $x^{(i)}_\tau(t'-Z+1) = 0$ for all $\tau = 1,2,\ldots,Z-1$. If $t' \le t-1$, then by flow conservation $f^{(i)}_{\text{cch}}(t'-1) = 0$ and $f^{(i)}_{\text{evict}}(t') = 0$. So $a_i(t') = 1$ and $x^{(i)}_0(t'+1) = 1$. Then this enforces $x^{(i)}_\tau(t'+1) = 0$ for all $\tau = 1,2,\ldots,Z$. For all u with $t' < u < t$, we can verify that $f^{(i)}_{\text{evict}}(u) = 0$. Then by the construction of the cache schedule, $a_i(u) = 0$. Therefore, the queue stays empty, i.e., $x^{(i)}_\tau(t) = 0$ for $\tau = 1,2,\ldots,Z$. At t, since $f^{(t-1)}_{\text{cch}} = 1$ and $f^{(t)}_{\text{evict}} = 1$, we have $a_i(t) = -1$, and thus $x^{(i)}_0(t+1) = 0$. For any u with $t < u \le T+1$, we can show that $f^{(i)}_{\text{mem}}(u) = f^{(i)}_{\text{cch}}(u) = f^{(i)}_{\text{evict}}(u) = 0$, so $a_i(u) = 0$. We also know that $\sigma(u-1) \neq i$. So the queue for i stays empty at $T+1$ and i is not in the cache at $T+1$. Combing these, we can see that $x^{(i)}_\tau(T+1) = 0$ for $\tau = 1,2,\ldots,Z-1$ and $x^{(i)}_Z(T+2) = 1$. So $f^{(i)}_{\text{mem}}(T+1) = \text{RHS}$. If $t' = t$, then we have $f^{(i)}_{\text{mem}}(t-Z) = f^{(i)}_{\text{evict}}(t) = 1$ and $f^{(i)}_{\text{cch}}(t-1) = f^{(i)}_{\text{cch}}(t) = 0$, and thus $a_i(t) = 0$. Using similar arguments as above, we can show that the queue for i stays empty and i is not in the cache at $T+1$. Then the RHS is 1 and thus $f^{(i)}_{\text{mem}}(T+1) = \text{RHS}$.

Now consider the case where $f^{(i)}_{\text{mem}}(T+1) = 0$. Then $f^{(i)}_{\text{evict}}(t) = 0$ for all t with $t^* < t \le T+1$. If $f^{(i)}_{\text{cch}}(T) = 1$, then by flow conservation, $f^{(i)}_{\text{cch}}(T+1) = 1$. Then by Claim 1, $x^{(i)}_0(T+2) = f^{(i)}_{\text{cch}}(T+1) = 1$. Then $x^{(i)}_Z(T+2) = 0$ and thus $f^{(i)}_{\text{mem}}(T+1) = \text{RHS}$. If $f^{(i)}_{\text{cch}}(T) = 0$, then again, by flow conservation, we have that $f^{(i)}_{\text{cch}}(t-1) = 0$ and $f^{(i)}_{\text{mem}}(t-Z) = 0$ for all t with $t^* < t \le T+1$. By Claim 1, $x^{(i)}_0(t^*+1) = f^{(i)}_{\text{cch}}(t^*) = 0$. If $f^{(i)}_{\text{mem}}(t^*) = 1$, then we must have $T+1-t^* < Z$. Therefore, $x^{(i)}_{t^*+Z-T-1}(T+2) = x^{(i)}_Z(t^*+1) = 1$ and thus the RHS of (41) is equal to 0. If $f^{(i)}_{\text{mem}}(t^*) = 0$, then $x^{(i)}_Z(t^*+1) = 0$ or $x^{(i)}_\tau(t^*+1) = 1$ for some $\tau = 1,2,\ldots,Z-1$. Since $x^{(i)}_0(t^*+1) = 0$, there must exists a $\tau = 1,2,\ldots,Z-1$ such that $x^{(i)}_\tau(t^*+1) = 1$. Let τ^* be the smallest τ such that $x^{(i)}_\tau(t^*+1) = 1$. Then $x^{(i)}_1(t^*+\tau^*) = 1$. Since $1 \le \tau^* \le Z-1$, we have $t^*+\tau^*-Z \le T$ and thus by the induction assumption $f^{(i)}_{\text{mem}}(t^*+\tau^*-Z) = x^{(i)}_1(t^*+\tau^*) = 1$. We must have $t^*+\tau^* > T+1$ since $f^{(i)}_{\text{mem}}(t-Z) = 0$ for all t with $t^* < t \le T+1$. Then $1 \le t^*+\tau^*-T-1 \le Z-1$ and $x^{(i)}_{t^*+\tau^*-T-1}(T+2) = x^{(i)}_1(t^*+\tau^*) = 1$. Thus $0 = f^{(i)}_{\text{mem}}(T+1) = \text{RHS}$.

This completes the proof of Claim 2.

From Claims 1 and 2, it is easy to see that

$$\mathbb{1}_{a_i(T)=1} \le f^{(i)}_{\text{mem}}(T-Z) = x^{(i)}_1(T+Z) \tag{43}$$

$$\mathbb{1}_{a_i(T)=-1} \le f^{(i)}_{\text{cch}}(T-1) = x^{(i)}_0(T) \tag{44}$$

$$\sum_{i \in [M]} x^{(i)}_0(T) \le C = \sum_{i \in [M]} f^{(i)}_{\text{cch}}(T-1) \le C. \tag{45}$$

This verifies the constraints (7)–(9) and proves that the cache schedule defined in (31) is feasible. □

Once we have Lemma 1, the only thing left is to show that the MCMCF problem and the latency minimization problem have the same objective function. This is easy to see once we compare the objective functions (15) and (14) and apply Claim 2 from the proof of Lemma 1.

A.3 Optimizations to Reduce Complexity

In this section, we provide implementation details of BELATEDLY for reducing complexity. Our overall approach is illustrated in Figure 6.

A.3.1 ***Pruning and Merging.***

While the MCMCF formulation is conceptually simple, a naive implementation of the algorithm has serious practical limitations. Observe that the number of flow variables in the MCMCF formulation is $O(N \cdot M)$. For a request sequence of size $N = 250{,}000$ containing $M = 20{,}000$ objects, the number of decision variables alone would be on the order of 10^{10}. Further, the total number of flow conservation constraints is $O(N \cdot M)$ (see (19)–(22)). In Gurobi, where decision variables are encoded as 64-bit floating-point values, and constraint expressions as vectors of 64-bit pointers to the relevant decision variables, simply encoding the model would require well over 400 GB of memory.

In this section, we describe two optimizations to the above formulation that allow us to significantly tighten the resource requirements (memory and execution time) for solving the MCMCF problem and to make it more tractable. Our goal is to be able to compute BELATEDLY on a 32-core x86 server with 128 GB of RAM, for request sequences containing $N \approx 250{,}000$ requests, $M \approx 50{,}000$ objects, and any combination of z and C.

Caching Intervals. Since the majority of decision variables stem from either $(V_{\text{cch},n}, V_{\text{cch},n+1})$ (cache-to-cache) or $(V_{\text{cch},n}, V_{\text{mem},x})$ (cache-to-memory) edges, we first attempt to reduce the number of elements in these sets. The key idea here is that, for each object, the request sequence can be partitioned into disjoint intervals (composed of one or more consecutive timesteps) where BELATEDLY

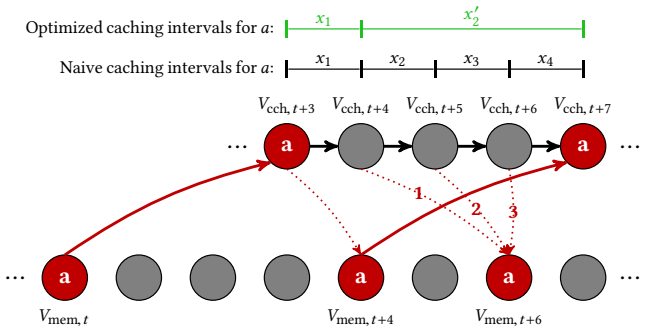


Figure 23: A fragment of a request sequence highlighting nodes and edges corresponding to object a (colored red), with $Z=3$.

is never incentivized to change its caching decision for that object; we call these *caching intervals*.

To concretize this notion, consider the subproblem depicted in Figure 23. Per the original MCMCF formulation, there are four distinct decision variables on edges between cache vertices corresponding to a (denoted by x_1, x_2, x_3, and x_4). Now, consider the possibility of routing flow along edges labeled **1**, **2**, and **3**. All three edges have the same capacity, cost-per-unit-flow, and destination node. Effectively, the latency cost incurred by evicting a using any of these edges is identical. However, observe that routing a's flow along edge **2** involves keeping a in the cache for one timestep longer than routing it along edge **1**. Similarly, routing a's flow along edge **3** involves keeping it in the cache for two additional timesteps. Since deferring the eviction consumes valuable cache space (but yields no tangible benefit in terms of latency cost), it is *strictly better* to evict a using edge **1** (at timestep $t+4$) than using edges **2** or **3**.

This simple observation gives us three major optimization opportunities. In particular, it enables us to:

- Eliminate the redundant edges **2** and **3** (along with the corresponding decision variables).
- Replace x_2, x_3, and x_4 with a single decision variable, x_2'. Since edges **2** and **3** no longer exist, any flow entering $cch_{(t+4)}$ must remain in the cache until $cch_{(t+7)}$; in other words, BELATEDLY's caching decision remains the same for the entire duration of the interval $[(t+4),(t+7))$.
- Eliminate flow conservation constraints involving object a for nodes $cch_{(t+5)}$ and $cch_{(t+6)}$. In the new representation, for each object, i, we only need flow conservation constraints for V_{cache} nodes corresponding to the end-points of i's caching intervals.

Lastly, this representation also allows us to bound the total number of caching intervals for any request sequence. Let n_i denote the number of requests to object i in a given request sequence of size N. Observe that an endpoint of object i's caching intervals is a V_{cch} node that either corresponds to i being admitted into the cache, i being evicted from it, or both. Since there are exactly n_i admission edges corresponding to object i, there must be *at least* n_i endpoints (or, equivalently, n_i-1 intervals) corresponding to i. Conversely, in the worst case, there are n_i-1 additional V_{cch} nodes which have eviction edges corresponding to object a. Thus, there may as many as $2n_i-1$ unique endpoints (or, equivalently, $2n_i-2$ caching intervals) corresponding to i. The total number of caching intervals (for all objects), K, can then be bounded as follows:

$$\sum_{i\in[M]}(n_i-1)\le K\le\sum_{i\in[M]}2(n_i-1)$$
$$\Rightarrow N-M\le K\le 2(N-M).$$

For a fragment of an empirical trace (CAIDA Chicago, 2014) containing $N=250{,}000$ packets and $M=37{,}725$ objects (unique flows), the total number of caching intervals is on the order of $400,000$. Compared to the naive formulation, this optimization reduces the number of decision variables from 18×10^9 to 10^6, and the number of model constraints from 9×10^9 to 10^6.

Optimizing Away Backing Store Nodes. Partitioning the global set of nodes into cache nodes and backing store nodes is a convenient abstraction since it allows us to reason about cache evictions and admissions independently of one another. Unfortunately, this representation also adds considerable overhead: excluding sink nodes, there are N backing store nodes, each of which contributes one decision variable on an edge $(V_{\text{mem},T},V_{\text{cch},T+Z})$, as well as one flow conservation constraint. However, observe that, in our MCMCF formulation, any flow entering a $V_{\text{mem},T}$ node *must* be routed to the corresponding cache node, $V_{\text{cch},T+Z}$. This leads us to our next optimization: replacing pairs of cache eviction and admission edges of the form $(V_{\text{cch},T},V_{\text{mem},x})$ and $(V_{\text{mem},x},V_{\text{cch},x+Z})$ with a single edge $(V_{\text{cch},T},V_{\text{cch},x+Z})$ with unit capacity and cost $c^{(i)}(V_{\text{cch},T},V_{\text{cch},x+Z})=c^{(i)}(V_{\text{mem},x},V_{\text{cch},x+Z})$ for object i.

As an example, consider the subproblem depicted in Figure 24. Here, $V_{\text{mem},t}$ has two in-edges, labelled **1** and **2**, and one out-edge, labeled **3**. Using the optimization strategy discussed above, we can coalesce edges **1** and **3** into a single edge, **1'**, with a capacity of 1 and a cost-per-unit-flow of $c^{(a)}=6$. Similarly, we can coalesce edges **2** and **3** into a single edge, **2'**. This effectively disconnects node $V_{\text{mem},t}$ from the remainder of the flow graph, and we can safely remove

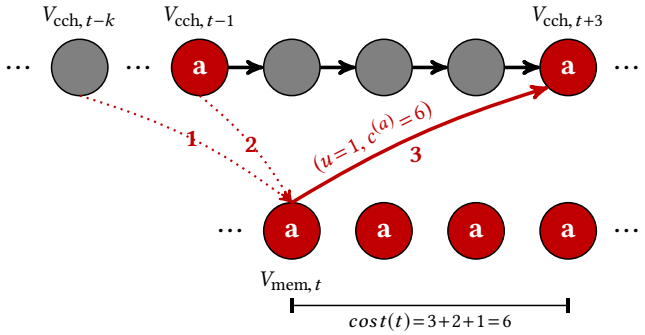


Figure 24: A fragment of a request sequence highlighting ingress and egress edges for node $V_{\text{mem},t}$, with $Z=3$.

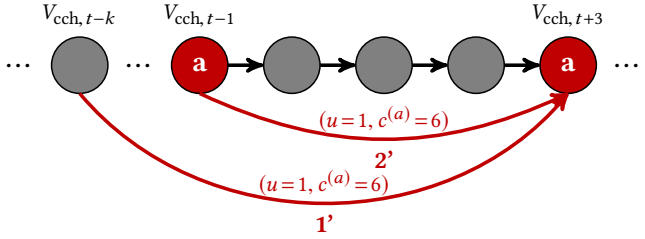


Figure 25: The optimized representation with backing store nodes removed.

it from V. A visual representation of the optimized flow graph is depicted is Figure 25. Overall, this optimization:

- Eliminates N decision variables corresponding to all N backing store to cache edges.
- Eliminates N flow conservation constraints corresponding to backing store nodes (*excluding* sink nodes).

For the aforementioned empirical trace, this optimization reduces the total number of decision variables and model constraints by another 25% (down to 750,000 each). Overall, the optimized MCMCF formulation (expressed in Gurobi C++ format) occupies under 25 GB of memory.[17]

*A.3.2 **Rounding to Approximate Integer Solutions.***

Recall that, since the integer version of MCMCF is NP-Complete, we instead opt to solve a fractional (or *relaxed*) version of the problem by removing the integrality constraints. However, this often results in solutions that do not map on to realistic caching strategies.[18] In this section, we describe our methodology for extracting an implementable caching schedule from a fractional solution.

A naive, yet intuitive, strategy is to simply round any non-zero fractions of evicted flows to 1, thereby always creating enough space in the cache for the next object to be admitted; unfortunately, this greedy rounding strategy does not generally work. It is easy to construct request sequences where evicting *too much flow* results in a violation of the cache capacity constraint several timesteps later. Further, attempting to satisfy the constraint by randomly evicting objects causes the upper-bound on the latency cost to diverge significantly from the true optimum. BELATEDLY addresses this problem in two ways:

(1) Instead of rounding *all* non-zero evicted flow fractions to 1, rounding is done with a probability corresponding to the fraction itself (a form of *randomized* rounding). In other words, if the fraction of flow for object i evicted at timestep T is $f^{(i)}_{evict}(T) \in [0,1]$, then we perform eviction with probability $f^{(i)}_{evict}(T)$. This ensures that, in expectation, the cache occupancy at any timestep is equal to the total flow routed along the corresponding edge in the MCMCF solution.

(2) While randomized rounding works well in theory, it does not *guarantee* that the cache capacity constraint is satisfied. In order to enforce this, we introduce the notion of *flow balance*. The idea is to track the *expected* amount of cached flow for each object (according to the fractional solution); then, at any timestep, if the cache occupancy would exceed the cache size, we evict the flow that is most *unbalanced* (deviates the most from its expected cached fraction). In practice, this is implemented using a priority queue.

A.4 BELATEDLY Performance Evaluation

Recall that we apply two optimizations to make the MCMCF problem described in §3.1 tractable: first, we prune and merge states in the flow graph to reduce the number of decision variables (§A.3.1); second, we solve a 'relaxed' version of the problem, followed by integer rounding (§A.3.2), to ensure that the algorithm terminates in polynomial time. In this section, we evaluate the benefits of these optimizations (using the naive MCMCF formulation as a baseline), as well the impact of rounding on BELATEDLY's latency upper-bound.

[17] This includes overheads incurred by Gurobi's internal data-structures; the raw model itself is significantly more compact.

[18] For instance, the optimal fractional solution may involve caching *half* an object, which is not particularly meaningful.

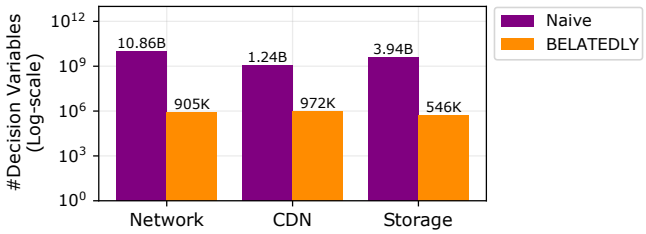


Figure 26: Number of decision variables in the naive MCMCF formulation versus BELATEDLY for different application scenarios.

Our optimizations to the original MCMCF formulation reduces BELATEDLY's memory and compute requirements by orders of magnitude. In Figure 26 we count the number of decision variables in the MCMCF formulation given our naive construction (§3.1) and our pruned version (§A.3.1). For all three application scenarios, the number of decision variables is reduced by three to four orders of magnitude.

Empirically, the formulation provides tight bounds. While solving a 'relaxed' version of the problem only gives us a lower-bound on the total latency (and not an implementable schedule), our randomized rounding strategy and flow balance heuristics work well in practice. For each application scenario, we perform 20 runs of BELATEDLY sweeping different Z values and cache sizes. Across all three scenarios, we see a median error of at most 0.05% and a maximum error of 1.71%. Table 3 lists the relative error between the upper- and lower-bounds of the solution generated by BELATEDLY.

	Mean Err.%	Median Err.%	Max. Err.%
Network	0.017	0.004	0.124
CDN	0.325	0.051	1.707
Storage	0.015	0.007	0.072

Table 3: Empirical bounds on BELATEDLY's error (calculated by comparing the integer upper-bound to the relaxed lower-bound).