USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Latency-Aware Caching with Delayed Hits: From Bursty Traffic to Pipeline Architectures

Nadav Keren, Gil Einziger, and Gabriel Scalosub,
Ben Gurion University of The Negev

https://www.usenix.org/conference/nsdi26/presentation/keren

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

Latency-Aware Caching with Delayed Hits: From Bursty Traffic to Pipeline Architectures

Nadav Keren Gil Einziger Gabriel Scalosub
Ben Gurion University of the Negev

Abstract

Modern computing systems rely on caching to reduce access latency and optimize resource utilization. However, in heterogeneous storage and cloud environments, non-uniform access latencies across storage tiers, network locations, and intermediary caches undermine traditional caching. Moreover, modern cache algorithms that attempt to capture multiple access patterns, recency, frequency, and burstiness, often become complex and difficult to maintain.

As a key contribution, we propose an adaptive caching architecture that treats caching strategies as a pipeline of simple, orthogonal policies, each focused on a distinct access bias. This modular design is easier to expand, debug, and integrate, and it self-adjusts the memory resources allocated to each stage to optimize overall workload performance. New heuristics can be introduced dynamically without disrupting existing behaviors.

In addition, in latency-aware caching, one often encounters the phenomenon of delayed hits, where items not yet available in the cache are requested repeatedly. We introduce the Least Bursty Used (LBU) heuristic, which retains items exhibiting high burstiness even when they are neither recent nor frequent, thereby mitigating delayed hits that degrade request latency. We embed LBU within our pipeline and derive the Recency–Frequency–Burstiness (RFB) policy, which balances resources among recency, frequency, and burstiness. Evaluations on thirteen real-world storage traces from IBM, Twitter and Meta using latencies drawn from real-life deployments show that RFB reduces average request latency by 10% compared to the best state-of-the-art alternative, while maintaining consistent performance, with a low standard deviation across bursty and non-bursty workloads.

1 Introduction

Modern applications increasingly rely on remote computation, distributed storage, and continuous streaming. Examples include Content Delivery Networks (CDNs) serving video and music, large-scale machine learning pipelines training on geographically dispersed datasets, and cloud or edge platforms composed of interdependent services. This reliance amplifies the cost of latency: items are often retrieved from distant sources, and access times are affected by network delays, congestion, and storage medium variability.

Caching is the standard remedy, reducing average access time by storing frequently requested items in faster, closer tiers. The classical cache model focuses on maximizing the hit ratio, defined as the fraction of requests served from the cache. Heuristics exploiting recency (e.g., LRU) and frequency (e.g., LFU) [20, 33] have been studied extensively under this model.

While simple and intuitive, the hit-ratio model is also simplistic. When access costs vary across heterogeneous storage tiers, network distances, or intermediary caches, maximizing hit ratio alone does not minimize latency. Two caches with identical hit ratios may yield very different response times if one evicts expensive-to-fetch objects in favor of cheaper ones. Such nonuniform costs arise in diverse settings, including search engines [7], distributed storage [12,24,26,40,41,47,50], mobile applications [37], DNS resolution [32], and web services [21].

To address these limitations, the community has proposed cost-aware caching models [11, 21]. These models associate each miss with a cost reflecting latency, monetary expenditure, or energy consumption, and optimize cache management accordingly [25, 38]. Algorithms such as GDS, GDSF, and GDS-LC [14, 21, 28, 38] exploit such metrics but rely on fixed heuristics and assume instantaneous admission.

When access latency is non-negligible, requests may arrive for items that are in transit to the cache but not yet available, resulting in a *delayed hit*. From the system's perspective, delayed hits behave like regular hits since they do not trigger additional fetches. From the user's perspective, their latency falls between a miss and a hit: shorter than a full fetch but longer than an immediate response. Explicitly modeling such intermediate states allows cache policies to better capture user-perceived performance and reduce access times in bursty workloads [5, 58].

Despite these advances, cache policies are still typically presented as monolithic designs. Policies are devised to address various workload characteristics, but do so within a single design, which leads to complex algorithms that are difficult to develop, deploy, and maintain.

We propose an alternative: a modular pipeline architecture composed of simple, narrowly focused cache policies. Each component policy targets a specific behavioral dimension, and the pipeline dynamically adjusts their relative sizes during execution to optimize performance. To demonstrate the approach, we extend the delayed-hits model by combining well-known recency and frequency heuristics with a new burst-oriented policy of our own design (LBU). The resulting adaptive pipeline achieves state-of-the-art performance while remaining modular and easy to manage.

Contributions. This paper makes five contributions to the study of latency-aware caching: (i) We design *Least Bursty Used (LBU)*, a cache policy that prioritizes items exhibiting bursty access patterns, (ii) we formalize the concept of a *pipeline cache architecture*, which composes multiple cache policies into a unified framework capable of addressing diverse workload characteristics, (iii) we introduce a simulation-based adaptive pipeline algorithm that dynamically allocates resources across the policies in the pipeline according to the observed performance, (iv) we propose *RFB*, a state-of-the-art adaptive policy based on our pipeline framework, specifically designed to address workloads driven by recency, frequency, and burstiness, and (v) we evaluate RFB against 12 SoTa policies across 13 real-world traces, demonstrating an average improvement of 10%.

1.1 System Model

We now describe our system model, inspired by [5, 44, 58], where our ultimate goal is to minimize the *average request latency* (to be formally defined in the sequel). Given a universe of uniform-size items U that are available in some (remote) *data source*, and a (local) cache π capable of storing at most M items, we consider a sequence σ of n requests arriving at the cache, $\sigma = r_1, r_2, \ldots, r_n$, where each such request r is characterized by a pair (i_r, t_r), where $i_r \in U$ is an item, and t_r is the issue time of the request. We will often omit the subscript r when it is clear from the context. We let σ_t denote the set of requests that have arrived by time t, and we assume without loss of generality that there is a well defined order on requests arriving at the same time t, i.e., we may refer to any request $r = (i,t)$ as the *unique* request arriving at time t, where we refer to requests arriving prior to t as requests that appear in σ before r.

For every request $r = (i,t)$, we define the fetch latency of retrieving i at time t, denoted $L_{i,t}$, as the time required to retrieve item i from its data source into the cache, for a retrieval request issued at time t. We assume $L_{i,t}$ is drawn from an item-specific latency distribution, which is generally unknown, and may well differ across items originating from different sources. For simplicity, we sometimes omit the subscript t, when the time is clear from the context, and simply refer to this time as L_i.[1]

For any time t, we let π^t denote the overall state of cache π at time t, and let C_t denote the set of items stored in π^t. In particular, for any request $r = (i,t)$, C_t denotes the items stored in the cache upon the arrival of r at time t. We now turn to describe the concepts required for formally defining the notion of *delayed hits*. At any time t, the cache maintains a set of *fetch requests* $F_t \subseteq \sigma_t$, and a set of *pending requests* $P_t \subseteq \sigma_t$. We sometimes refer to F_t and P_t as the *fetch requests buffer* (or F-buffer) and *pending requests buffer* (or P-buffer), respectively. We note that these buffers should merely maintain the metadata of the requests, and not the content of the items associated with these requests. We formally define these sets in the sequel. Furthermore, for every request $r = (i,t)$, we will also denote by $F_{<t}$ and $P_{<t}$ the sets of fetch requests and pending requests immediately before the arrival of request r at time t.

For any set of requests R, we let $I(R) = \{i \in U | \exists r = (i,t) \in R\}$, i.e., the set of items for which there is a corresponding request in R.

Intuitively, fetch requests in F_t are requests that initiate fetching an item from the data source, whereas pending requests in P_t are *additional* requests for items with an outstanding fetch request, i.e., there exists a fetch request in F_t for the same item. F_t and P_t maintain the *state* of fetched and pending requests, respectively.

Fetch requests. For every request $r = (i,t)$, if $i \notin C_t \cup I(F_{<t})$, then we say that r is a *fetch request*, and we add r to F_t. Such a request represents a request for an item that is currently not in the cache, nor is it in the process of being fetched from the data source. In particular, note that at any time t', and for any item i, there can be at most one request for item i in $F_{t'}$ (since such a request is added to F_t only if i is not in $I(F_{<t})$). Intuitively, any item may have at most one outstanding fetch request at any given time. For any time t, and request $r = (i,t)$, if r is not the first request for i in σ, we let $f_t(i)$ denote the last time before time t where a fetch request for i was issued, and refer to $f_t(i)$ as the *fetch time* of i with respect to time t.

Pending requests. For every request $r = (i,t)$ for which $i \in I(F_t) \setminus C_t$, we say r is a *pending request*, and we add r to P_t. Note that since $i \in I(F_t)$, there is a request $(i, f_t(i)) \in F_t$. Request r represents a request for an item that is currently being fetched from data source (where its fetch time is $f_t(i) \leq t$). We note that the sets of fetch and pending requests are always defined *implicitly*, certainly in hindsight, regardless of whether a caching algorithm maintains them explicitly; For any algorithm, and any time t, the identity of fetched

[1] Prior work (e.g., [21]) has shown that, for sufficiently long workloads and realistic latency distributions, using the most recent fetch latency provides an effective estimate for latency-aware cache calculations. In later sections we will use this estimation for L_i.

requests, and pending requests, as derived from the actual decisions made by the algorithm, is uniquely determined and well defined.

Clearing requests from the F-buffer and the P-buffer. For any fetch request $r = (i,t)$, we say that item i is successfully *retrieved*, or *made available*, from the data source at time $t' = t + L_i$. We recall that a fetch request actually triggers a retrieval of item i from the data source, when the item is not available in the cache at time t, and no such retrieval has been issued just prior to time t. Such a fetch request $r = (i,t)$ is thus cleared from the F-buffer at time $t' = t + L_i$. We now turn to describe how pending requests are cleared from the P-buffer. Any pending request $r = (i,t)$ has its unique corresponding fetch request $r' = (i,t') \in F_t$ for $t' = f_t(i)$. r is thus cleared from the P-buffer at time $t' + L_{i,t'}$, i.e., at the same time where the fetch request r' is cleared from the F-buffer (i.e., as soon as the item is made available in the cache).

For every request $r = (i,t) \in \sigma$, we wish to define its *request latency*, $c(r) = c(i,t)$, i.e., the latency between the time the request arrives, and the earliest time in which the item is available in the cache. We distinguish between three cases:

1. If $i \in C_t$ then the request is considered to be a *cache hit*. The request latency of such a request is therefore $c(i,t) = 0$.
2. If $i \notin C_t \cup I(F_{<t})$, i.e., the item is not in the cache and there is no outstanding fetch request for item i, the request is considered to be a *cache miss*. In such a case, r is added to F_t, and the fetch time of i is set to t. The request latency of such a request is the item's fetch latency, i.e., $c(i,t) = L_i$.
3. If $i \in I(F_t) \setminus C_t$, i.e., the item is not available in the cache, but there is an outstanding fetch request for the item, then the request is a pending request and is considered to be a *delayed hit*. In such a case, r is added to P_t, and the request latency of r is $c(i,t) = f_t(i) + L_{i,f_t(i)} - t$. This captures the *residual request latency* for request r, where its corresponding fetch request was issued at time $f_t(i) \le t$.

We will focus on designing online caching solutions that minimize the *average request latency* (sometimes referred to as the ARL) of the sequence, i.e.,

$$\tilde{c}(\sigma) = \frac{1}{n} \sum_{r \in \sigma} c(r). \tag{1}$$

Denote by $\sigma[t_1,t_2]$ all the requests $(i,t) \in \sigma$ such that $t_1 \le t \le t_2$, and let $\sigma_i[t_1,t_2]$ denote all the requests for item i in $\sigma[t_1,t_2]$. For any request $r = (i,t)$, we define its *accumulated request latency* as the sum of request latencies of all requests to i that arrive within the time interval $[t, t+L_i]$. Formally,

$$s(i,t) = \sum_{(i,t') \in \sigma_i[t,t+L_i]} c(i,t') = \sum_{(i,t') \in \sigma_i[t,t+L_i]} \left(t + L_i - t'\right). \tag{2}$$

We note that the accumulated request latency of $r = (i,t)$ can be viewed as the *potential* overall cost (over all affected requests) of *not caching* item i at time t, where $r = (i,t)$ ends up being a fetch request. In case $r = (i,t)$ is an *actual* fetch request (i.e., r is added to F_t at time t), then the accumulated request latency $s(i,t)$ accounts for the request latency of all requests for i arriving during the interval $[t, t+L_i]$, i.e., until the item is retrieved from the data source. Out of these requests, the first request $r = (i,t)$ is a fetch request, and all other requests are pending requests, which can be viewed as implicitly associated with the fetch request r. Since every request that is not a cache hit is either a fetch request, or a pending request implicitly associated with a single fetch request, it follows that by taking $F(\sigma) = \bigcup_t F_t$ as the set of all fetch requests over the entire arrival sequence, we obtain

$$\tilde{c}(\sigma) = \frac{1}{n} \sum_{(i,t) \in F(\sigma)} s(i,t). \tag{3}$$

This view essentially considers the request latency of each pending request as "piggybacked" on its corresponding fetch request.

The cache employs some *eviction policy* for determining the *victim* item, which should be evicted from the cache whenever a fetched item has been retrieved and is available for insertion to the cache at time t and the cache is full. We make no assumption on the eviction policy used by the cache.

Figure 1 illustrates the cache model, showing how a size 2 cache processes requests under an LRU eviction policy.

The first request is for item A arriving at time t_0. The request is added to F_t at t_0, and we start fetching the item from the data source. The item is retrieved and made available at the cache at time $t_0 + L_A$, implying that at that time item A is added to C_t, the fetch request (A,t_0) is served by the cache, and the request is removed from F_t. The next request, which arrives at time t_1, is for item B, which is not in the cache. The request is added to F_t at t_1, and we start fetching item B from the data source. A subsequent request for item B arrives at time t_2, where B is not yet available at the cache. Since $B \in I(F_{t_2})$, this request (B,t_2) is, therefore, a pending request and it is added to P_t at time t_2. The next request to arrive is a request for item C arriving at time t_3, where C is not in the cache. The request is added to F_t at t_3, and we start fetching item C from the data source. As the next request at time t_4, also for item C, arrives before C is available at the cache, this request is added to P_t at time t_4. At time $t_1 + L_B$ item B has been successfully retrieved from the data source and is added to the cache. The two requests (B,t_1) (the fetch request in F_t) and (B,t_2) (the pending request in P_t) for item B are thus served and removed from F_t, and P_t, respectively. The cache stores at this point items A and B. At time t_5 an additional request for item C is issued and added to P_t, since item C is not yet available in the cache. Finally, at time $t_3 + L_C$ item C is successfully retrieved. Since the cache size is 2, and an LRU policy is employed, item A is evicted to allow adding

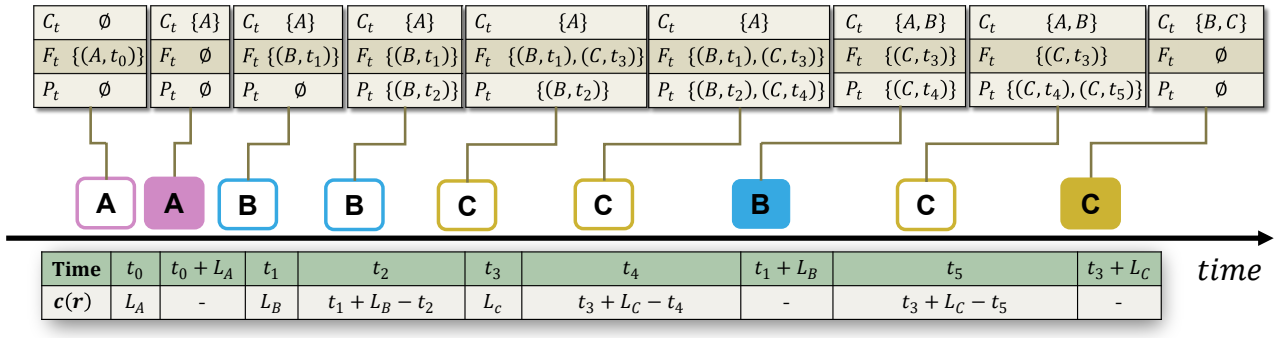


Figure 1: Illustration of an arrival sequence within our system model with a cache of size 2, including the timestamp of each request and the perceived request latency. A filled square represents the arrival of an item to the cache.

item C to the cache. At this point, the three requests for item C (one in F_t, and two in P_t) are served, and removed from their corresponding sets. Figure 1 also illustrates the request latency $c(r)$ incurred by each of the requests.

1.2 Related Work

Cache algorithms are heuristic-based methods that attempt to anticipate and adapt to workload patterns. Each workload favors a different heuristic, making the best-performing cache algorithm workload-specific. Moreover, even within the same workload, the optimal heuristic may change over time depending on period-specific variations. The seminal Belady's Clairvoyant policy [9] provides an upper bound on the maximal attainable hit ratio. This optimal policy looks ahead into the future and evicts the item whose next access occurs the furthest in time. Despite the diversity of cache algorithms, most rely on a few simple, yet effective, heuristics. The most common is the Recency heuristic, which assumes that recently accessed items are more likely to be accessed again soon. A widely used implementation of this heuristic is the Least Recently Used (LRU) policy [27].

The Frequency heuristic differs from Recency by considering the distribution of accesses over time rather than individual access events. This approach assumes that frequently accessed items in the recent past are likely to remain popular in the near future, and is the basis of many implementations [3, 4, 33, 52] including recent developments such as S3FIFO [62] and SIEVE [64].

In particular, TinyLFU [20] is a popular frequency-based caching policy. TinyLFU employs a compact oracle based on a sketching algorithm [15,23] to estimate the access frequency of each item and admits only items whose frequency exceeds that of the eviction candidate.

Another noteworthy heuristic is reuse distance, which considers the interval between subsequent accesses to the same item. Prominent examples employing this approach include HiFi [1], LIRS [31, 36], LHD [8] and FRD [49].

Recent advancements in caching integrate machine learning algorithms in multiple approaches, from learning the future access patterns in the workload [17,54,57] to dynamically adjustable scoring functions [2, 13, 51]. The most notable of these is the LRB policy [54] which tries to mimic the performance of Belady's policy. These are interesting approaches that come at a computational cost, demanding specialized hardware, abundance of information and as a side effect, make the decision making process less interpretable.

As systems grow in complexity, cache algorithms also become more convoluted, and harder to implement and modify. To address this shortcoming, many adaptive caches are designed, balancing different heuristics to better match workload characteristics; To the best of our knowledge, all suggested adaptive algorithms try to balance at most two heuristics, and are limited to tuning a single parameter, such as cache sizing [16, 20, 45], or modifying a scoring function [11]. Notable examples of such dynamic policies include Adaptive Replacement Cache (ARC) [45], its recent improvement SHARC [16], and adaptive TinyLFU [18]. These algorithms function by combining two caches that work together, with adaptivity achieved through dynamic cache resizing.

Our work extends cost-aware cache algorithms [7, 30, 38, 39, 48, 53, 63] by *incorporating the possibility* of a delayed hit when accessing an item, instead of creating a new ad-hoc approach as offered in other works [5]. Notable cost-aware algorithms include GDWheel [38], CAMP [25], Hyperbolic caching [11] and Cost-Aware Window TinyLFU [21]. These algorithms incorporate the recency and frequency heuristics while taking into account variable access times. The latter also offers an adaptive model for fine-tuning the cache sizes, while not taking delayed-hits into account.

The proposed Adaptive CA-W-TinyLFU algorithm can be viewed as a two-stage pipeline of recency and frequency, incorporating a hill-climber mechanism that dynamically balances their sizes. In addition, the authors introduce a latency-aware variant of LRU, called Cost and Recency Aware (CRA).

Moreover, since items are of variable size, two main approaches were adopted: (i) cluster items of similar size using

a technique such as slabbing [8, 61, 62, 64], and then apply a unit-sized cache policy (as in this work), or (ii) design cache policies that directly accommodate large variability in object sizes [10, 14, 19] .

1.3 Preliminaries and Latency Aware Policies

To conclude our introduction, we provide details on the two policies used as example building blocks in our proposed solutions. The CRA policy [47] extends the LRU policy for latency-aware scenarios. CRA utilizes multiple LRU-like lists, each storing items whose fetch latencies fall within a specific range, determined by the latencies observed so far. When an eviction is required, CRA considers both the time since the last query for an item and its latency to compute a score, which is used to select the victim item for eviction.

The access-time-aware TinyLFU [47] leverages two CRA policies in conjunction, effectively forming a Segmented-LRU (SLRU) policy [34]. Under this approach, an item's score is defined as the product of its latency and frequency. When an eviction decision is made, the item with the lowest score is chosen as the victim.

In our design and evaluation, we utilize these policies as our Recency-aware and Frequency-aware policies. For convenience, we refer to CRA as LRU* and the access-time-aware TinyLFU as LFU*. Notably, in our proposed solution, we do not use the vanilla (access-time-oblivious) LRU or LFU.

2 The Least-Bursty-Used (LBU) Policy

In this section, we introduce our new cache eviction policy, referred to as the *Least-Bursty-Used (LBU)* policy, which is latency-aware and specifically designed to retain items accessed in bursts. That is, LBU targets items that are accessed successively within relatively short time intervals. As we demonstrate, in certain workloads, failing to cache such bursty items can lead to a significant increase in average request latency. We discuss how burst-aware caching differs from traditional paradigms such as LRU and LFU, highlighting key logical distinctions. To illustrate these differences, we provide a synthetic example comparing the performance of LBU against LRU and LFU in workloads with distinct access patterns. Next, we introduce a metric designed to quantify the impact of burstiness on average request latency. Finally, we describe the design and implementation of our LBU cache, explaining how it effectively captures and leverages bursty access patterns to improve latency.

2.1 A Call for Burstiness Awareness

At first glance, mechanisms designed for bursty traffic may appear similar to those that favor recency-biased access patterns. However, the key distinction lies in the length of the time interval over which requests for an item arrive. LRU always admits a new item upon request, making it most effective when the cache completes the admission process before the next request for that item occurs. However, in the presence of bursty traffic, subsequent requests for an item often arrive before the item has been fully retrieved from the data source, leading to multiple delayed hits (as defined in our system model). These delayed hits substantially increase the average request latency, negating the benefits of temporal locality.

Thus, despite the strong temporal locality exhibited by bursty workloads, an LRU-based policy may fail to leverage this locality effectively to minimize request latency. This limitation arises because bursts may be too short-lived or scattered arbitrarily throughout the workload, preventing LRU from sustaining cached items long enough to serve subsequent requests efficiently.

In contrast, an LFU-based policy might favor items for which requests exhibit a bursty arrival pattern. This depends, of course, on the *overall* frequency of the requests for such an item. An LFU-based policy will often fail to distinguish between an item whose access pattern is dense within a short time frame (i.e., bursty), and an item whose access pattern has the same density but is distributed over a longer time frame. In particular, a classic LFU approach focuses on the *benefits* of having an item in the cache but is oblivious to the tolls incurred when an item is missing from the cache. The reason for this is that LFU treats cache misses as isolated events, which can be easily mitigated by simply (re)admitting the item to the cache. Thus, subsequent requests for the item would result in a cache hit, giving the impression that cache misses have minimal impact.

However, in latency-aware settings, this approach fails to capture the effect of not having an item i in the cache at time t on potentially multiple requests for the item during the interval $[t, t+L_i]$.

We note that for many, if not most, workloads, relying solely on the burstiness of items as a primary criterion for cache management may not be advisable. However, incorporating burstiness awareness can lead to performance improvements in specific cases, as we demonstrate in the following section. Moreover, integrating burstiness awareness into more general, *adaptive* architectures presents an opportunity to achieve significant performance gains over state-of-the-art designs. Our adaptive pipeline cache design, described in Section 3.1, explicitly incorporates this capability. As we later demonstrate in Section 4, our solution leverages burstiness awareness to exploit optimization opportunities beyond those captured by existing approaches.

As an illustration of the need for a new metric in cache policy design, we conduct experiments on a carefully crafted synthetic workload, which is an amalgamation of four types of items: (i) many items exhibiting a *recency bias*, which receive approximately 30 requests, uniformly distributed over ~ 20 seconds, and are never requested again, (ii) a few items exhibiting a *frequency bias*, which arrive once every ~ 1

Table 1: Performance of LRU*, LFU*, and LBU on varying synthetic traffic patterns.

Traffic bias	LRU*	LFU*	LBU	Burst Score
Recency	**65.61**	390.06	606.73	1296.13
Frequency	459.78	**79.29**	999.94	1295.53
Burstiness	499.59	501.70	**6.63**	**505000**

second throughout the sequence, (iii) a few items exhibiting a *burstiness bias*, which arrive in bursts of ~ 1000 requests within ~ 1 second, with each burst occurring once every $\sim$ 1000 seconds, and (iv) many items that are *one-hit wonders*, i.e., requested only once throughout the sequence. These serve as noise and cannot be effectively cached by any policy. All requests have a fetch latency of 1 second.

Table 1 compares the average request latency for LRU*, LFU*, and our proposed LBU policy (defined in Section 2.3) across different workload biases.[2] The table presents the average request latency of each policy when evaluated on each type of item in the sequence, along with the average burst score, which we define in Section 2.2.

As shown, the LRU* policy excels in handling recency-biased items, the LFU* policy performs well for frequency-biased items, and the LBU policy is highly effective in managing burstiness-biased items. However, each policy performs poorly on the other item types. This highlights that burstiness-biased items are not well-handled by common caching policies within a latency-aware framework, reinforcing the need for a specialized approach.

In the next section, we formally define a metric for measuring "burstiness" and demonstrate how this metric can be translated into a burst-oriented cache policy.

2.2 Defining a Burstiness Score

In this section, we focus on computing a *burstiness score*, denoted β_i, for each item i. For LRU-based or LFU-based policies, computing the "score" of an item i at any time t is well-defined. This typically involves tracking the number of distinct items accessed since the last access to i, or counting how frequently item i has been accessed up to time t. It is also common to apply various heuristics in the computation of the score in order improve computational efficiency and reduce resource overhead.

A natural candidate for the burstiness score of an item i at time t is its accumulated request latency, denoted $s(i,t)$. This metric quantifies the latency cost incurred if item i is not present in the cache at time t, and a fetch request for i is issued at that moment. However, by definition, this value can only be computed in hindsight, at time $t+L_i$. We note that although the accumulated request latency was originally defined for individual requests $r=(i,t)$, it can still be considered for item i at any time t, even if no actual request for i arrives at t. Nevertheless, we estimate the burstiness score of i at time t by leveraging the accumulated request latencies of past requests for i, even when i was already present in the cache at the time those requests were issued.

There are a few caveats to consider when using accumulated request latency as a measure of burstiness. First and foremost, workload patterns may change significantly over time, meaning that items previously identified as "bursty" may become less bursty, and vice versa. However, we note that this issue is not unique to burstiness estimation, as similar challenges also affect other caching heuristics, such as LFU.

Second, the accumulated request latency can be highly sensitive to the timing of an item's first appearance in the workload. For instance, a single isolated request for an item i might result in a very low burstiness score. However, this same item i might soon be followed by multiple requests within a short time interval. If burstiness is estimated solely based on the initial isolated request, it may underestimate the true bursty nature of i. In contrast, an estimate made on a subsequent request for i might yield a much higher burstiness score, which more accurately captures the actual burstiness pattern of i.

Because accumulated request latency serves as the foundation for our burstiness score, this score should be re-evaluated periodically, even when the item is still in the cache. Failing to update the burstiness score in a timely manner could lead to significant degradation in average request latency. Ideally, one would evaluate (and update) the burstiness score β_i for item i at every time t when a request $r=(i,t)$ arrives. However, doing so could introduce significant overhead, particularly in terms of memory usage required to compute all potential scores. In what follows, we introduce the Least-Bursty-Used (LBU) policy and present an efficient method for estimating the burstiness score of distinct items.

2.3 The LBU Policy

For every request $r=(i,t)$ for item i, our LBU policy calculates the accumulated request latency associated with a *constant* number of requests, denoted by ℓ, arriving during $[t,t+L_i]$ (in a sliding-window manner). This ensures that no more than ℓ values are maintained for every item i. The burstiness score β_i is then set to be the maximum over these ℓ values. This score serves as an estimate of the exact burstiness score (as discussed in Sec. 2.2). The fact that β_i is estimated along a sliding window whenever a new request for i arrives, ensures that the estimation follows the burstiness dynamics of the various items being handled. The parameter ℓ serves to balance the tradeoff between the accuracy of the estimation (which improves as ℓ is larger) and the overheads incurred by computing the estimation. We employ an aging mechanism to ensure that rare outliers do not remain in the cache for

[2] We note that in our illustrating example, all items have the same fetch latency, which implies that LRU* and LFU* behave the same as LRU and LFU, respectively.

prolonged periods of time. Whenever the cache is full, and a new item is offered for admittance (after it has been retrieved from the data source), the eviction policy mandated by the LBU policy picks the item with the smallest burstiness score (over all items currently in the LBU cache, along with the newly arrived item) as the victim designated for eviction.

A vanilla implementation of LBU can be performed using a heap, with time complexity of $O(\log M)$ per operation. However, in our implementation of the LBU policy, we attain a worst-case time complexity of $O(\log M)$ for evictions, and $O(1)$ for testing a new candidate, at the cost of a slight relaxation of ensuring that the item with the minimum burstiness score is indeed chosen as the victim. This includes periodic self-adjustments of the data structures at constant-time amortized cost. The formal description of our estimated burstiness score and the full implementation details can be found in Appendix A.

3 A Pipeline Caching Architecture

This section presents our pipeline method for designing complex cache policies from simple independent *cache blocks* that are pipelined sequentially. In this setup, the victim of a cache block is the input to the next cache block in the pipeline.

This concept has been explored on a limited scale in prior work, such as in [18, 20], which can be interpreted as a pipelined design of LRU and TinyLFU. Our work formalizes and generalizes this pipelined design pattern. Each pipeline exposes a spectrum of policies that vary by the sizing of the pipelined blocks.

One of the primary motivations behind our general design is to incorporate an LBU cache into an already very successful pipeline that combines recency and frequency. As previously mentioned, we do not expect LBU to be a standalone policy. It does not attempt to get a high hit ratio but to store in its cache bursty items which LRU and LFU based policies might miss. By adding LBU to a pipeline, we avoid designing a rigid, ad-hoc cache architecture that tries to do it all in one complex algorithm.

We assume the total memory allowance for the entire cache architecture to be M. At any time t, we consider the cache C_t as being partitioned into a sequence of m cache blocks, denoted $B_1^t, \ldots, B_m^t$. The size of each cache block B_j^t at time t is given by $\left|B_j^t\right| = \alpha_j^t M$, where $\alpha_j^t \in [0,1]$ and $\sum_j \alpha_j^t = 1$.[3] Each cache block B_j^t operates as a separate caching mechanism, which can be managed using different cache policies, such as LRU, LFU, or LBU. We now proceed to describe the inner workings of our Pipeline Cache architecture, for which the pseudocode is provided in Algorithm 1.

The pipeline handles two types of events: (i) *incoming request events* $r = (i,t) \in \sigma$, which are requests originating at the client and arriving at the cache, and (ii) *retrieved item events* of item i at time t, where t is the time at which the item arrives at the cache from the data source. We note that for each such retrieved item event of i at t, there exists a single fetch request $r = (i, f_t(i)) \in F_t$.

We first describe the behavior of the pipeline cache for incoming request events. Consider the arrival of a new request $r = (i,t)$. The policy first checks whether item i is available in C_t, i.e., whether it is stored in one of the blocks $B_1^t, \ldots, B_m^t$. If the item is available in some block, the request is served from this block, and its metadata is updated (e.g., recency, frequency, burstiness, and possibly other metadata, as we discuss in the sequel). We recall that as part of our latency-aware model, incoming requests for which the requested item is not available in the cache (i.e., it is not stored in any of the cache blocks) are inserted into either the fetch requests buffer (in which case a fetch request is issued to the backend data source) or the pending requests buffer, as described in our system model. These buffers are maintained by the pipeline architecture, rather than by the individual cache blocks.

We now turn to describe the behavior of the pipeline cache for the second type of events, retrieved item events. These events occur when an item, previously being fetched from the data source due to the arrival of an incoming fetch request, arrives at the cache and becomes available. First, any outstanding fetch request for i, as well as any additional pending requests for i, are served. This is followed by an update to the relevant item's metadata and the removal of the request from the corresponding request buffer (F_t or P_t). Once all outstanding requests for i are served, the cache blocks apply their logic in sequence.

Let $i_0 = i$ denote the item that was thus retrieved at time t. In what follows, for every $j = 1, \ldots, m$, block B_j^t would be offered a *virtual request* (i_{j-1}, t) for admittance to B_j^t; This is performed by the *admit* call in line 20 in Algorithm 1, which either returns $i_j = null$ in case there is available memory in block B_j^t and item i is retained in block B_j^t, or the *victim* item i_j destined for exclusion from block B_j^t (as prescribed by the policy employed at B_j^t). In the former case, no further action is needed (as no item should be evicted from the pipeline). In the latter case, the victim i_j, is chosen out of the set of items currently available for block B_j^t, along with the new item being offered for admittance, i_{j-1}. This victim item i_j is then offered to B_{j+1} (if $j < m$), or dropped (if $j = m$). The above behavior is formally defined in Algorithm 1 and illustrated in Figure 2. It is important to notice that items can be offered to cache blocks only in the *predefined* order of the pipeline and that no item is contained in more than a single block at any given time. The order of the blocks in the pipeline may well, and indeed often does, affect the behavior and performance of the overall pipeline cache architecture.

[3] We make a slight abuse of notation and let B_j^t refer to block j at time t, while $\left|B_j^t\right|$ denotes the amount of memory allocated to block j at time t.

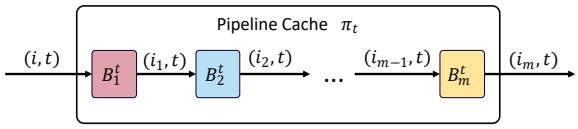


Figure 2: An overview of a flow of an item within the pipeline

Algorithm 1 HandleEvent(π^t, $e=(i,t)$)

Require: pipeline π^t, event $e=$(item i, time t)
1: **if** e is an incoming request event **then**
2: apply HandleIncoming($r=(i,t)$) in π^t
3: **else** ▷ e is a retrieved item event
4: apply HandleRetrieved(i,t) in π^t

5: **procedure** HandleIncoming(request $r=(i,t)$)
6: **if** $i \in C_t$ **then**
7: serve r, and update metadata
8: **else if** $i \notin I(F_t)$ **then** ▷ (i,t) is a fetch request
9: $F_t \leftarrow F_t \cup \{(i,t)\}$
10: fetch item i from data store
11: **else** ▷ $i \in I(F_t)$, i.e., (i,t) is a pending request
12: $P_t \leftarrow P_t \cup \{(i,t)\}$
13: **end procedure**

14: **procedure** HandleRetrieved(item i, retrieval time t)
15: **for** every request $r=(i,t') \in F_t \cup P_t$ **do**
16: serve r, and update metadata
17: remove r from $F_t \cup P_t$
18: $i_0 = i$
19: **for** $j=1,\ldots,m$ **do**
20: $i_j \leftarrow \text{admit}(B_j^t,(i_{j-1},t))$ ▷ extract B_j^t's victim i_j
21: **if** i_j is *null* **then** ▷ B_j^t was not full
22: **return**
23: drop i_m
24: **end procedure**

3.1 An Adaptive Pipeline Architecture

In this section, we introduce a *self-adjusting*, *adaptive* pipeline cache, which dynamically (re)sizes the blocks within the pipeline while maintaining a constant overall cache size. For practicality, we assume that the partitioning of the available memory is performed in a *coarse* manner, i.e., in *quanta* of non-negligible size Q. We further refer to κ as the number of possible quanta in a cache of size M, such that: $\kappa \cdot Q = M$. For any $j=1,\ldots,m$, and any time t, we let q_j^t be the size of block B_j^t, i.e., the number of quanta allocated to B_j^t at time t.

Our dynamic scheme enables a variety of cache behaviors, allowing it to adapt to the underlying workload and approximate the best *overall policy*. We note that the exact semantic characterization of the resulting overall policy cannot be explicitly determined. At the core of our adaptive pipeline is a continuous search for improved sizing of the pipeline blocks. We note that this allocation process is a moving target, as workload variations also modify the required allocation. While sizing-based adaptivity is commonly employed, it has traditionally been implemented in an ad-hoc manner between only two competing options [18, 45]. In what follows, we present our design for multiple (i.e., possibly more than two) pipeline blocks, where in Section 4 we discuss the performance of a concrete example of a 3-block pipeline.

The pseudocode for our adaptive architecture is provided in Algorithm 2. We make use of a fixed-step *hill-climbing* algorithm that *simulates* multiple possible *adaptation steps* and *applies* the most promising one among the tested options. Each simulated reallocation in our algorithm is referred to as a *ghost cache*, or *oracle*. For every such ghost cache, only items' metadata is utilized, without modifying the actual cache contents. The configuration-change decisions are performed periodically, every pre-defined number of requests relative to the cache size, and we refer to the time between two consecutive configuration-change decisions as a *simulation interval*. to ensure statistically significant resource reallocations rather than random fluctuations. The chosen adaptation step is applied at the end of each such period, following an *adaptation event*. In our design, such an event coincides with the arrival of a request, once every $10 \cdot M$ requests.

Our hill-climber only considers adaptation steps that reduce the size of one pipeline block by a single quantum (i.e., a fixed amount Q) and increase the size of another block by the same fixed amount. This approach restricts the number of considered options while still exploring a broad range of possible configurations.

Specifically, given a pipeline cache with m blocks, there are $2 \cdot \binom{m}{2} = m(m-1)$ possible ghost caches to consider. For example, in a three-block pipeline (as discussed in depth in Section 4), there are six possible adaptation steps to consider at each adjustment opportunity (see Table 2).

We note that the limited reallocation of a fixed amount of resources in each adaptation step is intentional, as our goal is not to find a local minimum, but rather to avoid allocations that result in unnecessary performance degradation. As we demonstrate in the sequel, the resulting performance remains competitive even if the optimal allocation is not attained; merely being near a good allocation is sufficient.

When performing an adaptation step at time t, we do not evict items from the cache. Instead, we relocate items from the reduced-size block B_j^t to the expanded block $B_{j'}^t$. This is achieved by virtually "removing" the lowest-priority items from B_j^t (according to its eviction policy) and virtually "adding" them to the newly available space now allocated to $B_{j'}^t$. The priority of these items in $B_{j'}^t$ is determined by the eviction policy of the target block. We emphasize that this process only involves updating metadata within the pipeline, avoiding any costly data copying and preventing unnecessary evictions from the cache. This reconfiguration of the main

pipeline cache π^t is captured by the applying the configuration of the best ghost (in case its performance is indeed better than the performance attained by the main pipeline cache), captured by the "assignment" in line 7 of Algorithm 2. The adaptation step ends by resetting all the ghost caches to the new configuration of the main pipeline cache π^t, and then resizing the blocks in each ghost cache according to the simulation it is destined to perform (lines 9-10 in Algorithm 2). Even though Algorithm 2 is formulated with ARL as its optimization objective, it can be easily adapted to target other objectives.

Given a starting configuration, we simulate all adjacent configurations obtained by moving one chunk between policies. At each decision point, we compare the results of these simulations and switch to the best-performing configuration, or retain the current one if it remains superior.

Our approach is similar to prior work [18, 21], which also relies on decision intervals and adaptive resizing. The key differences are: (i) previous approaches are limited to two policies and resize one only at the expense of the other, and (ii) the previous work do not use simulations. Instead, it performs an adaptation step (e.g., increase LRU and decrease LFU) and repeat the step if performance improves; if performance declines, they revert the change. This makes the previous work vulnerable to missteps when workload fluctuations, rather than configuration changes, affect performance. By contrast, our method avoids this issue by simulating all candidate configurations within the same interval, ensuring adaptation decisions are based on consistent evidence.

In the *full ghost* caches, we simulate the altered cache configuration on *all incoming requests*. However, such a simulation can become prohibitively expensive as the number of pipeline blocks m increases. To see that, consider that for each request, many ghost caches require a state update. Thus, we employ a sampling method inspired by MiniSim [56], which considers only a fraction of the incoming traffic. We refer to oracles implemented with this sampling approach as *sampled ghost* caches, and we provide further details of the implementation in the following. Sampling drastically reduces overheads, both in terms of memory required for maintaining ghost caches and the processing overhead associated with updating their states. As we demonstrate in Section 4, the implementation costs of our improved design are comparable to, and at times superior to state-of-the-art caching architectures.

We now turn to describe our simulation using *sampled ghost* caches, where these sample ghost caches have a significantly smaller footprint in terms of memory and computation, when compared to the full ghost caches. The simulation takes into account a *sample* of the sequence of incoming requests, which captures all requests that are associated with some *subset of items*. This is achieved by using a single random hash function which identifies the (random, but fixed) set of items. The underlying idea of this sampling approach is to extrapolate the behavior of a full-size cache handling all requests

Algorithm 2 PipelineReconfigure(event $e = (\text{item } i, \text{time } t)$)

Require: main pipeline cache π^t, ghost pipeline caches G^t_j, $j = 1, \ldots, m(m-1)$

1: apply HandleEvent(π^t, e)
2: **for** $j = 1, \ldots, m(m-1)$ **do**
3: apply HandleEvent(G^t_j, e) ▷ update state of ghost
4: **if** e is an adaptation event **then** ▷ once every $10 \cdot M$ requests
5: $j^* \leftarrow \arg\min_j \{\text{ARL}(G^t_j)\}$ ▷ best ghost cache
6: **if** $\text{ARL}(G^t_{j^*}) < \text{ARL}(\pi^t)$ **then** ▷ ghost $G^t_{j^*}$ is better
7: $\pi^t \leftarrow \text{Configuration}(G^t_{j^*})$
8: **for** $j = 1, \ldots, m(m-1)$ **do**
9: $G^t_j \leftarrow \pi^t$ ▷ reset ghost cache to π^t's content
10: reallocate resources in G^t_j
11: **else**
12: **for** $j = 1, \ldots, m(m-1)$ **do**
13: Reset G^t_j stats.

from the behavior of a smaller cache that processes only the sampled sequence of requests. Intuitively, if we sample with a probability $0 < R < 1$, henceforth referred to as the *sampling rate*, then we can estimate the performance of a ghost cache of size M by observing the performance of a ghost cache of size $R \cdot M$. By using hash-based sampling rather than uniform sampling, we ensure that all occurrences of sampled items appear in the sampled stream, thus retaining all information about those items, including their burstiness and frequency. We compute the average request latency for the sampled subsequence and use it as an estimate of the average request latency for the entire request stream.

We note that using uniform sampling would result in inaccurate statistics, particularly in terms of frequency and burstiness, as not all requests corresponding to an item would be sampled. Our hash-based sampling approach, while utilizing a smaller simulated cache, is inspired by MiniSim [56]. However, our use case is novel, as the authors of MiniSim focused on mitigating performance cliffs, whereas our work aims to enhance adaptive caching algorithms. [4]

We note that the sampling approach improves both the memory and computation overheads of simulating the ghost caches. Observing the ghost caches, when no sampling is applied, we simulate $m \cdot (m-1)$ full ghost pipelines, one for each configuration change possible. When using sampled ghost caches, we simulate $m \cdot (m-1) + 1$ ghost ghost caches (one more ghost cache simulation is required for the current cache configuration). Each of these caches is of size $R \cdot M$, and, on average, only an R fraction of the requests are handled in the simulation. This implies that the overhead in the sampled

[4] We note that employing our sampled ghosts requires a slight modification of Algorithm 2: One needs to also simulate π^t using the sampled sub-universe in order to ensure a sound reset of the ghosts in line 9.

Table 2: Example of the six ghost caches maintained by FGHC in a pipeline with three blocks. Each ghost cache differs from the main cache by reducing one block by one quantum and increasing another block by one quantum.

Cache name:	Block 0	Block 1	Block 2
Main Cache	0	0	0
Ghost cache 1	+1	-1	0
Ghost cache 2	-1	+1	0
Ghost cache 3	+1	0	-1
Ghost cache 4	-1	0	+1
Ghost cache 5	0	+1	-1
Ghost cache 6	0	-1	+1

pipeline is no more than an

$$\frac{R\cdot(m(m-1)+1)}{m(m-1)} = R\cdot\left(1+\frac{1}{m(m-1)}\right)$$

fraction of the overhead required for the full ghost pipeline.

As an illustration, consider a sampling rate of $R=\frac{1}{8}$ to simulate a pipeline cache with $m=3$ blocks. We need to simulate a total of $m(m-1)=6$ blocks (as shown in Table 2), resulting in overall metadata overheads of $\frac{6}{8}$, which implies that we increase our cache's metadata for ghost entries by 75% for adaptivity. As calculated above, the operational overhead is merely 15% of the overhead required for using the full ghost simulation, and generally, we perform 0.875 additional operations in computation for metadata updates per request. While such overheads are not negligible, they remain comparable to those of similar algorithms such as FOMO [42] (100% ghost entries), ARC [45] (100% ghost entries) and LIRS [31] (200% ghost entries). Furthermore, these overheads are significantly more manageable than those incurred by full simulations, which require six metadata updates per item (which can be done in parallel to the actual cache management), and a 600% ghost entry overhead.

4 Performance Evaluation

This section introduces an empirical study where we evaluate the performance of our proposed designs on various workloads against leading alternatives. In our study we explore the design choices in our proposed architecture, as well as provide insight into the real-time adaptivity of our solutions.

The datasets: We use real traces from IBM's Object-Storage solution [22] with varying lengths, along with additional traces from Meta's KV Storage [59] (those who contain request timestamps) and Twitter's memcached [60] which are available online [29, 46]. Due to space constraints, we present our evaluation results for a subset of the available traces, which exemplify workloads with *diverse characteristics*. The diversity is further exemplified by the fact that the optimal static RFB configurations across the various traces exhibit distinct allocation of resources to the constituent blocks, as shown in Table 3.

Table 3: Survey of the traces used, including the *best static RFB* pipeline configurations (column RFB), the *best static RF* pipeline configurations (column RF), and the improvement of RFB over RF (column Diff). For both RFB and RF, the optimal configurations are denoted as (x,y,z) and (x,y), respectively, where x quanta are allocated to the recency block, y to the frequency block, and z (for RFB) to the burstiness block. The total number of quanta is 16.

Trace	Requests	Cache Size	RFB	RF	Diff
IBM10	13,193,532	512	(14, 2, 0)	(14, 2)	0%
IBM12	532,892	1,024	(1, 1, 14)	(1, 15)	19%
IBM24	33,370,453	512	(1, 14, 1)	(1, 15)	0.8%
IBM29	505,631	512	(16, 0, 0)	(16, 0)	0%
IBM31	31,580,019	65,536	(2, 14, 0)	(2, 14)	0%
IBM34	19,368,658	16,384	(13, 2, 1)	(14, 2)	0.2%
IBM45	28,277,611	4,096	(16, 0, 0)	(16, 0)	0%
Meta2	199,999,962	8,192	(0, 15, 1)	(0, 16)	1.1%
Meta4	199,999,942	8,192	(1, 14, 1)	(1, 15)	1.2%
Twi1	200,006,207	1,024	(0, 15, 1)	(0, 16)	0.2%
Twi3	38,080,953	1,024	(0, 16, 0)	(0, 16)	0%
Twi9	200,003,272	4,096	(0, 16, 0)	(0, 16)	0%
Twi28	200,000,000	4,096	(2, 9, 5)	(5, 11)	4.4%
Concat	85,263,358	512	(4, 12, 0)	(4, 12, 0)	0%

The traces include the issue time for each request, but are devoid of information regarding the items' fetch latency. We thus enrich the traces by associating each request with a latency drawn from real-world distributions reported for AWS distributed systems [40] and model every read as a full item retrieval. We note that [40] reports a range of latency profiles. We simulate a representative deployment configuration of three sites: Oregon, Ohio, and Ireland, where measured latencies from these sites to Victoria, Canada have ranges 100–140ms, 315–365ms, and 650–700ms, respectively. We assume items are distributed uniformly at random across sites. For cache misses, the latency is drawn from a normal distribution calibrated such that 90% of the samples fall within the specified ranges, following the methodology of [40]. The results appear in Table 4.

We note that similar distributions were reported in additional work (e.g., [26]).

Determining the Cache Size: For each trace, we target evaluating its performance using a cache size that avoids getting trivial results for the trace. First, the cache size should not be too large to avoid the case in which most items can fit within the cache. Second, the cache size should not be too small, so most requests will not be considered misses or delayed hits. In order to find a sizing that attains these two goals, we conducted experiments testing the performance of LRU* against various cache sizes (powers of 2) and picked the size corresponding to the Pareto front that minimizes the distance to the origin, thus balancing these above requirements.

Concrete Pipeline Policies We make use of our pipeline cache architecture and define a new policy called the *Recency-*

Table 4: Comparison of policy performance (ARL, in %) relative to the best static RFB pipeline across various traces (lower is better). The average performance of each policy over all native traces is shown in blue, and our proposed policies are shown in green, whereas the performance for the concatenation of traces (described in Section 4.2) is marked with yellow. For completeness, we also report LBU in isolation, although it is not intended as a standalone policy outside our pipeline. Best-performing policies (within one point of the optimum) are marked in gray. Policies are ordered by average performance, from worst to best.

Policy	IBM10	IBM12	IBM24	IBM29	IBM31	IBM34	IBM45	Meta2	Meta4	Twi1	Twi3	Twi9	Twi28	Average ± STD	concat
GDW [38]	6.1%	71.0%	51.3%	6.4%	19.7%	148.3%	84.5%	126.8%	130.7%	501.0%	126.0%	225.6%	110.4%	123.7% ± 129.6%	37.3%
LAC [58]	45.2%	138.0%	39.9%	33.8%	47.2%	185.3%	143.9%	96.7%	102.5%	163.2%	167.3%	120.4%	97.1%	106.2% ± 52.4%	36.5%
LRU* [47]	**0.2%**	76.5%	49.5%	0.0%	60.6%	1.0%	**0.0%**	57.3%	58.5%	318.9%	28.8%	97.5%	26.1%	59.6% ± 84.3%	32.0%
LBU	44.0%	24.0%	38.2%	32.5%	41.7%	65.6%	133.3%	36.6%	33.4%	19.2%	17.6%	18.6%	22.7%	40.6% ± 30.9%	32.7%
LFU* [47]	41.5%	77.1%	3.1%	33.0%	21.3%	115.0%	134.9%	**1.1%**	**3.0%**	**0.2%**	**0.0%**	**0.0%**	9.3%	33.8% ± 46.4%	8.0%
LHD [8]	1.3%	78.8%	35.2%	9.5%	33.8%	-3.1%	**0.7%**	42.9%	44.2%	101.3%	17.7%	23.1%	28.6%	31.9% ± 30.6%	20.8%
ACA TLFU [47]	37.8%	55.7%	1.2%	29.0%	**4.3%**	108.4%	126.3%	**0.2%**	**2.8%**	**0.7%**	**-0.1%**	**-0.1%**	11.4%	29.0% ± 43.1%	5.9%
ARC [45]	**0.3%**	25.4%	42.6%	0.1%	44.0%	**-14.7%**	**0.0%**	41.4%	43.6%	100.4%	10.5%	18.1%	14.8%	25.1% ± 29.9%	26.8%
FRD [49]	22.0%	31.8%	21.1%	18.7%	15.6%	-8.1%	34.6%	39.3%	41.1%	55.7%	9.5%	18.8%	13.8%	24.1% ± 16.3%	16.2%
SIEVE [64]	21.1%	25.1%	8.6%	**-11.2%**	13.2%	43.0%	54.0%	5.0%	6.9%	47.3%	30.7%	32.8%	20.1%	22.8% ± 18.7%	7.1%
HLA [11]	18.8%	50.7%	25.9%	10.6%	39.1%	21.8%	31.1%	4.2%	6.0%	20.0%	10.3%	9.3%	3.9%	19.3% ± 14.3%	18.6%
S3Fifo [62]	16.2%	22.7%	13.4%	**-12.0%**	**4.3%**	16.7%	33.4%	30.6%	25.7%	27.6%	4.3%	8.8%	11.6%	15.6% ± 12.6%	9.6%
LRB [54]	**0.4%**	48.1%	**-7.9%**	0.1%	25.2%	10.5%	**0.0%**	26.9%	35.2%	15.3%	3.2%	7.8%	25.3%	14.6% ± 16.5%	4.8%
RF 1	**0.1%**	35.3%	-0.5%	0.1%	12.7%	1.9%	**0.0%**	20.2%	16.8%	1.9%	**0.7%**	1.1%	6.2%	**7.4% ± 10.9%**	**-4.0%**
RFB 1/4	**0.1%**	16.8%	-1.7%	0.1%	11.7%	3.4%	**0.1%**	24.2%	15.0%	3.8%	1.1%	1.8%	**-0.5%**	**5.8% ± 8.3%**	**-4.6%**
RFB 1	**0.0%**	**11.7%**	-2.6%	0.1%	27.1%	2.0%	**0.1%**	16.0%	**3.3%**	3.7%	1.3%	1.0%	**-0.8%**	**4.8% ± 8.5%**	**-5.5%**

Frequency-Burstiness pipeline (*RFB pipeline*), which consists of a pipeline with three blocks: the first is LRU*, the second is LFU*, and the third is our new LBU policy (presented in Section 2.3). For each trace, we consider both the (best) static RFB pipeline (which is obtained by a grid search over all possible configurations, and can only be known in hindsight), as well as our adaptive pipeline (as presented in Section 3.1). For the latter, we consider both the full-ghost, as well as sampled ghosts, implementations. We refer to RFB_R as the RFB pipeline with a sampling rate of R. E.g., RFB_1 is the adaptive pipeline which uses full ghost caches (i.e., with a sampling rate of 1), whereas $\text{RFB}_{1/4}$ is the adaptive pipeline with sampled ghosts, where items are sampled with probability $\frac{1}{4}$. In our evaluations, we also consider a simpler pipeline that does not have an LBU block, denoted by RF_1 (in the full ghost implementation). In all evaluations of pipeline policies, the cache resources are initially divided equally across the blocks.

4.1 Competitive Evaluation

Table 4 summarizes our results, comparing the performance of our proposed solutions (namely, pipeline policies) with state-of-the-art alternative solutions. A positive number indicates an increase in latency, while a negative number indicates that the algorithm achieves a lower latency than the baseline. The table specifies the size of the cache used for each trace according to the guidelines specified earlier in this section.

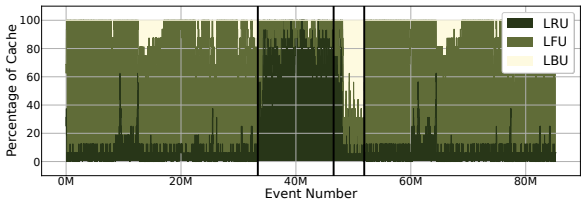
Table 3 shows that different traces are best handled by distinct configurations, with LBU providing clear benefits in some cases (e.g., IBM12, Twi28, and minor improvements in Twi1, Meta2, Meta4, IBM34, and IBM24). More importantly, in many traces the incorporation of LBU in only part of the pipeline yields better results, as reflected in the average performance of RFB_1 compared to RF_1 in Table 4.

The table further shows the performance of the state-of-the-art policies and the individual components of the RFB Pipeline, which, on average, perform at least 10% worse than the baseline policy and exhibit a larger standard deviation than our proposed adaptive solution. Indeed, for some traces, some state-of-the-art policies outperform the baseline (i.e., Adaptive CA-WTinyLFU [21], ARC [45], SIEVE [64], S3FIFO [62]). However, none of these state-of-the-art policies fares *consistently well* across all traces.

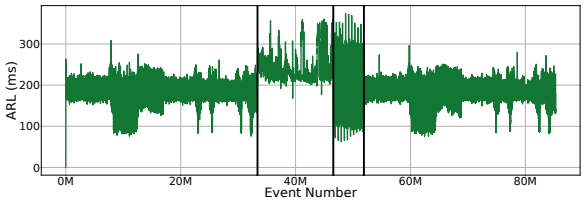
When considering the performance of our proposed adaptive pipeline policies, they all outperform the state-of-the-art policies. In particular, our full ghost RFB pipeline policy performs within 4.8% of the baseline on average, with a small standard deviation. *This amounts to almost 10% performance improvement when compared with state-of-the-art policies*.

Our results further unveil some of the *reasons* for the improvement in performance; First, we note that our RF_1 pipeline policy may seem similar to the architecture underlying the Adaptive-CA-WTinyLFU policy. A significant difference between these policies is the new hill climber algorithm employed in our pipeline architecture. We note that, on average, RF_1 exhibits an improvement of almost 25% when compared with Adaptive-CA-WTinyLFU, which can be mostly attributed to our novel hill-climbing approach.

Second, we note that, on average, our sampled ghost policy $\text{RFB}_{1/4}$ achieves nearly the same performance as the full ghost hill-climbing pipeline (within 1 percentage point) and delivers on average a 9% improvement over the next-best state-of-the-art algorithm. This highlights that most benefits of our approach can be attained while keeping the computational and memory overheads at bay, representing an overall significant improvement over state-of-the-art algorithms.



(a) The proportions of the blocks during the workload



(b) The average request latency of the cache during the time frame between two adaptation decisions

Figure 3: Analysis of adaptation over the concatenated traces IBM24, IBM10, IBM12×10, IBM24. The concatenation points are marked by the vertical lines in the plots.

4.2 Adaptive RFB-Pipeline: Under the Hood

We next evaluate the adaptivity of our hill climber for the RFB pipeline under shifting workloads. To this end, we construct a synthetic trace by concatenating IBM24, IBM10, and IBM12 (repeated 10 times), followed by a final IBM24. Keys are replaced between repetitions to prevent the cache from reusing prior state, ensuring that it must adapt at each transition. The cache size is fixed at 512, which is optimal for IBM10 and IBM24, and half the optimal size for IBM12.

Based on the best static RFB configurations in Table 4, the expected behavior is clear: IBM24 favors LFU*, IBM10 favors LRU*, and IBM12, characterized by bursty access patterns, allocates most resources to LBU with smaller shares to LRU* and LFU*.

We recall that our adaptive architecture seeks to identify resource reallocation opportunities and respond to workload dynamics. Figure 3 shows the behavior of the RFB pipeline on the concatenated trace, highlighting the reallocation of resources across blocks (Figure 3a) and the average request latency in each segment (Figure 3b).

Thanks to its adaptive nature, our approach responds to changes in real time and avoids many of the misses that occur under static misconfigurations. On the concatenated trace, our method outperforms the best static pipeline by 4–5.5 percentage points and achieves up to a 10% improvement over the closest competitor, LRB. These results show that our approach can effectively *learn* the workload on the fly and quickly adapt, yielding consistent performance gains. This is evident in the rapid resource reallocations at the start of IBM10 and again when IBM12 begins. We also note that the similar dynamics observed in both runs of IBM24 indicate that RFB's performance and behavior are largely *independent of the initial allocation of resources*.

The adaptability also explains why our algorithm outperforms the best static configuration for traces IBM24 and Twi28 (Table 4), by dynamically adjusting quota allocations as needed.

Lastly, we note that adaptive RFB reallocates resources in a manner closely matching the best static RFB for each trace independently, as shown in Table 3. Consistently, the average request latency follows the same trends observed for the independent traces in Table 4.

5 Discussion and Future Work

Our work tackles bursty traffic, which undermines the classic hit-or-miss cache model by introducing delayed hits accesses that arrive while an item is still being fetched into the cache. In particular, we design a burstiness-aware policy, LBU, that targets reducing the tolls incurred by such delayed hits, while ignoring other biases in the workload.

We further design a generic, and modular, pipeline caching architecture, which combines modules implementing distinct policies which target distinct biases. Our approach dynamically allocates space across these modules by periodically simulating candidate resize operations on the pipeline and applying the best-performing configuration to subsequent requests. We are able to maintain metadata and compute overheads of performing such simulations at bay by employing a lightweight sampling technique inspired by MiniSim [56]. Thanks to its adaptability, adaptive RFB can exploit trends within the trace and surpass the best static RFB configuration as is evident in IBM24 (Table 4).

Moreover, the results show that RFB's performance and behavior are largely *independent of the initial resource allocation*: in both segments where trace IBM24 appears, the observed dynamics are very similar despite significantly different initial allocations.

Our results demonstrate a plug-and-play design philosophy that enables rapid composition and evolution of cache strategies: to address a new access pattern, one merely adds a simple policy (akin to LBU) to the pipeline and allows the adaptive mechanism to self-tune allocations. We believe this approach will catalyze innovation in adaptive caching across diverse domains.

Looking ahead, we aim to integrate RFB into existing open-source projects. RFB already has a strong starting point, as it is implemented within Caffeine's simulator. Caffeine [43] itself is widely adopted and heavily used by the community, making it an ideal deployment target.

The primary challenge is to develop a Caffeine branch that replaces its internal caching policy with RFB, and to evaluate this branch on real-world projects that already rely on Caffeine, without requiring changes to their application logic. From a systems perspective, the main technical hurdle is efficient parallelization, which is not currently addressed in our simulation-based implementation.

Availability: Our full artifact is available at GitHub [35].

References

[1] Shahid Akhtar, Andre Beck, and Ivica Rimac. Caching online video: Analysis and proposed algorithm. *ACM Trans. Multimedia Comput. Commun. Appl.*, 13(4):48:1–48:21, August 2017.

[2] Ismail Ari, Ahmed Amer, Robert B Gramacy, Ethan L Miller, Scott A Brandt, and Darrell DE Long. ACME: Adaptive caching using multiple experts. In *WDAS*, volume 2, pages 143–158, 2002.

[3] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating content management techniques for web proxy caches. *SIGMETRICS Perform. Evaluation Rev.*, 27(4):3–11, 2000.

[4] Martin Arlitt, Rich Friedrich, and Tai Jin. Performance evaluation of web proxy cache replacement policies. *Perform. Eval.*, 39(1-4):149–164, February 2000.

[5] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S. Berger. Caching with delayed hits. In *SIGCOMM*, page 495–513, 2020.

[6] Multiple Authors. A collection of open access traces. https://github.com/cacheMon/cache_dataset.

[7] Emre Bakkal, Ismail Sengor Altingovde, and Ismail Hakki Toroslu. Cost-aware result caching for meta-search engines. In *SIGIR*, page 739–742, 2015.

[8] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *NSDI*, pages 389–403, 2018.

[9] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, 1966.

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

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

[12] Abhirup Chakraborty and Ajit Singh. Cost-aware caching schemes in heterogeneous storage systems. *J. Supercomput.*, 56(1):56–78, 2011.

[13] Jiayi Chen, Nihal Sharma, Tarannum Khan, Shu Liu, Brian Chang, Aditya Akella, Sanjay Shakkottai, and Ramesh K Sitaraman. Darwin: Flexible learning-based CDN caching. In *SIGCOMM*, pages 981–999, 2023.

[14] Ludmila Cherkasova. Improving www proxies performance with greedy-dual-size-frequency caching policy. Technical report, HP Labs, 1998.

[15] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55:29–38, 2004.

[16] Xiaoming Du and Cong Li. SHARC: Improving adaptive replacement cache with shadow recency cache management. In *Middleware*, page 119–131, 2021.

[17] Shahriar Ebrahimi, Reza Salkhordeh, Seyed Ali Osia, Ali Taheri, Hamid R. Rabiee, and Hossen Asadi. RC-RNN: Reconfigurable cache architecture for storage systems using recurrent neural networks. *IEEE Trans. Emerg. Top. Comput.*, 10(3):1492–1506, 2022.

[18] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive software cache management. In *Middleware*, page 94–106, 2018.

[19] Gil Einziger, Ohad Eytan, Roy Friedman, and Benjamin Manes. Lightweight robust size aware cache management. *ACM Trans. Storage*, 18(3), August 2022.

[20] Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A highly efficient cache admission policy. *ACM Trans. Storage*, 13(4):35:1–35:31, 2017.

[21] Gil Einziger, Omri Himelbrand, and Erez Waisbard. Boosting cache performance by access time measurements. *ACM Trans. Storage*, 19(1):8:1–8:29, 2023.

[22] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. It's time to revisit LRU vs. FIFO. In *HotStorage*, 2020.

[23] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.

[24] Brian C. Forney and Andrea C. Arpaci-Dusseau. Storage-Aware caching: Revisiting caching for heterogeneous storage systems. In *FAST*, pages 61–74, 2002.

[25] Shahram Ghandeharizadeh, Sandy Irani, Jenny Lam, and Jason Yap. CAMP: A cost adaptive multi-queue eviction policy for key-value stores. In *Middleware*, pages 289–300, 2014.

[26] Raluca Halalai, Pascal Felber, Anne-Marie Kermarrec, and François Taïani. Agar: A caching system for erasure-coded data. In *ICDCS*, pages 23–33, 2017.

[27] John L. Hennessy and David A. Patterson. *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann, 5th edition, 2012.

[28] Binbing Hou and Feng Chen. GDS-LC: A latency- and cost-aware client caching scheme for cloud storage. *ACM Trans. Storage*, 13(4):40:1–40:33, 2017.

[29] IBM. Snia - ibm object-storage traces. https://iotta.snia.org/traces/key-value.

[30] Jaeheon Jeong and Michel Dubois. Cost-sensitive cache replacement algorithms. In *HPCA*, pages 327–337, 2003.

[31] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *SIGMETRICS*, pages 31–42, 2002.

[32] Jaeyeon Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS performance and the effectiveness of caching. *IEEE/ACM Trans. Netw.*, 10(5):589–603, 2002.

[33] G. Karakostas and D. N. Serpanos. Exploitation of different types of locality for web caches. In *ISCC*, pages 207–212, 2002.

[34] Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.

[35] Nadav Keren, Gil Einziger, and Gabriel Scalosub. Adaptive pipeline cache. https://github.com/NadavKeren/Adaptive-Pipeline-Cache, 2026.

[36] Cong Li. DLIRS: Improving low inter-reference recency set cache replacement policy with dynamics. In *SYSTOR*, pages 59–64, 2018.

[37] Cong Li, Jia Bao, and Haitao Wang. Optimizing low memory killers for mobile devices using reinforcement learning. In *IWCMC*, pages 2169–2174, 2017.

[38] Conglong Li and Alan Cox. GD-Wheel: A cost-aware replacement policy for key-value stores. In *EuroSys*, pages 5:1–5:15, 2015.

[39] Shuang Liang, Ke Chen, Song Jiang, and Xiaodong Zhang. Cost-aware caching algorithms for distributed storage servers. In *DISC*, pages 373–387, 2007.

[40] Kaiyang Liu, Jun Peng, Jingrong Wang, and Jianping Pan. Optimal caching for low latency in distributed coded storage systems. *IEEE/ACM Trans. Netw.*, 30(3):1132–1145, 2022.

[41] Yanfei Lv, Xuexuan Chen, and Bin Cui. ACAR: An adaptive cost aware cache replacement approach for flash memory. In *WAIM*, pages 558–569, 2010.

[42] Steven Lyons and Raju Rangaswami. To cache or not to cache. *Algorithms*, 17(7):301, 2024.

[43] Ben Manes. Caffeine: A high performance caching library for java 8. *https://github.com/ben-manes/caffeine*, 2016.

[44] Peter Manohar and Jalani Williams. Lower bounds for caching with delayed hits. *CoRR*, abs/2006.00376, 2020.

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

[46] Meta and Twitter. Carnegie mellon university pdl cluster. https://ftp.pdl.cmu.edu/pub/datasets/twemcacheWorkload/cacheDatasets/.

[47] Giovanni Neglia, Damiano Carra, Mingdong Feng, Vaishnav Janardhan, Pietro Michiardi, and Dimitra Tsigkari. Access-time-aware cache algorithms. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 2(4):21:1–21:29, 2017.

[48] Rifat Ozcan, Ismail Altingovde, and Ozgur Ulusoy. Cost-aware strategies for query result caching in web search engines. *ACM Trans. Web*, 5(2):9:1–9:25, 2011.

[49] Sejin Park and Chanik Park. FRD: A filtering based buffer cache algorithm that considers both frequency and reuse distance. In *MSST*, 2017.

[50] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A case for MLP-aware cache replacement. In *ISCA*, pages 167–178, 2006.

[51] Faizal Riaz-ud Din and Markus Kirchberg. Acme-DB: An adaptive caching mechanism using multiple experts for database buffers. In *ICEIS*, pages 72–81. 2006.

[52] Ketan Shah, Anirban Mitra, and Dhruv Matani. An $O(1)$ algorithm for implementing the LFU cache eviction scheme, 2010.

[53] Won Wook Song, Jeongyoon Eo, Taegeon Um, Myeongjae Jeon, and Byung-Gon Chun. Blaze: Holistic caching for iterative data processing. In *EuroSys*, pages 370–386, 2024.

[54] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed Belady for content distribution network caching. In *NSDI*, pages 529–544, 2020.

[55] Storage Networking Industry Association's Input/Output Traces Tools and Analysis Technical Work Group (IOTTA TWG). Snia iotta. https://iotta.snia.org/traces/key-value.

[56] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *ATC*, pages 487–498, 2017.

[57] Xiaoye Wang, Xuan Li, Linji Wang, Tingyi Ruan, and Pochun Li. Adaptive cache management for complex storage systems using CNN-LSTM-based spatiotemporal prediction. *CoRR*, abs/2411.12161, 2024.

[58] Gang Yan and Jian Li. Towards latency awareness for content delivery network caching. In *ATC*, pages 789–804, 2022.

[59] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and K.V. Rashmi. Fifo can be better than lru: the power of lazy promotion and quick demotion. In *The 19th Workshop on Hot Topics in Operating Systems (HotOS 23)*, pages 1–8, 2023.

[60] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208, 2020.

[61] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 503–518. USENIX Association, April 2021.

[62] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. FIFO queues are all you need for cache eviction. In *SOSP*, pages 130–149, 2023.

[63] M. Zhang, Q. Wang, Z. Shen, and P. P. C. Lee. Parity-only caching for robust straggler tolerance. In *MSST*, pages 257–268, 2019.

[64] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K.V. Rashmi. SIEVE is simpler than LRU: an efficient turn-key eviction algorithm for web caches. In *NSDI*, pages 1229–1246, 2024.

A LBU Implementation Details

At this section we detail the implementation choices made in order to generate the results in the paper.

We begin by describing how we evaluate the burstiness score of an item i, at any time t. For any request $r = (i,t)$ which is a fetch request (i.e., $i \notin C_t \cup I(F_t)$), we issue the fetch request from the data source, and calculate $s(i,t)$. By definition, this value is available at time $t + L_i$. I.e., as soon as i is made available in the cache (at time $t + L_i$), we set its burstiness score to be $\beta_i = s(i,t)$.

As discussed in the section 2.2, the burstiness score associated with a fetch request might fail to capture the actual burstiness of the item. In order to address this possible handicap we perform *additional* evaluations of the burstiness as long as $i \in C_t \cup I(F_t)$ as follows: For every $k = 1, 2, \ldots$ we consider the first request for item i, $r = (i,t')$, arriving after time $f_{t'}(i) + k \cdot \frac{L_i}{\ell}$, for some constant ℓ. We recall that $f_{t'}(i)$ is the (latest) fetch time of i prior to time t'. For each such request $r = (i,t')$, if $s(i,t') > \beta_i$, we update the burstiness score of i to be $\beta_i = s(i,t')$. We note that this value is available only at time $t' + L_i$. Figure 4 illustrates the division of the time frame to calculate the burst score. It follows that for every item i, there is a *constant number* (i.e., ℓ) of outstanding accumulated request latency values being computed, corresponding to a sliding window of length L_i. This makes sure that the memory and computational overheads of estimating β_i are kept at bay. We note that the value of ℓ is a parameter governing the trade-off between how often we attempt to improve our evaluation of β_i, and how much memory is required to calculate such an improved evaluation. In particular, as ℓ increases, we test to see if β_i can be better evaluated against the accumulated request latency of more requests.

To ensure that LBU does not retain items with high burstiness scores indefinitely, we introduce a decay mechanism for the β_i values. For every item i whose burstiness score β_i has not been updated in the past $A \cdot L_i$ time, we set its new value to $\beta_i(1-\alpha)$, where $\alpha \in (0,1)$ is a decay parameter. This mechanism guarantees that items that were highly bursty in the distant past do not indefinitely gain priority over items that exhibited more recent bursty behavior, even if they were less bursty overall.

We now finalize our description of the proposed LBU policy by providing additional implementation details. Whenever a new item is offered for admittance into the LBU cache, and the cache is full, LBU seeks to evict the item i with the lowest current burstiness score β_i. That is, if the newly offered item itself has a burstiness score lower than the minimum β_i among all items currently stored in the cache, then LBU rejects the new item and the cache is unchanged.

We begin by targeting an implementation where no aging is performed on the burstiness scores of items in the LBU. The state of the LBU cache may change in reaction to two types of events: When an arriving request corresponds to an item that is currently in the LBU, and when an item is admitted to the LBU and some other item is evicted. Since no aging is performed, then a simple priority queue implemented as a binary Min-Heap would result in a time complexity of $O(\log M)$ for each of the above events.

Regardless of the underlying cost model, cache policies are often presented as monolithic designs. As a result, even a policy intended for delayed hits must incorporate recency and frequency heuristics that were originally studied under the simpler hit-ratio model. Attempting to handle all aspects in a single design leads to complex algorithms that are difficult to develop, deploy, and maintain.

We propose a different path: a pipeline architecture composed of simple, narrowly focused cache policies. Each policy addresses one aspect of access behavior, and the pipeline dy-

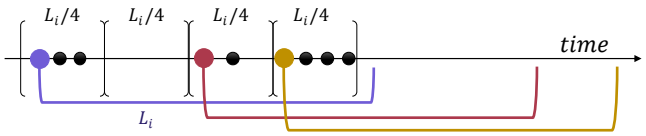


Figure 4: To calculate the burst scores when an item is already in the cache, we divide the time frame into L_i/ℓ windows (in this illustration, $\ell = 4$) and select the first query in each window as a *virtual fetch request*. These virtual fetch requests are then used to compute the burst value within a L_i time frame. Upon receiving a new query, we update at most ℓ accumulators by computing the time difference between the current query and each of these virtual fetch requests.

namically adjusts their relative sizes during execution to improve performance. To demonstrate the approach, we formed a state-of-the-art policy by designing the simple LBU policy to handle just bursts, and combining it with two existing policies for the well known recency and freuqnecy heuristics. Thus, we made a state-of-the-art policy by adding the logic to handle bursts, and we did not have to create a new monolith.

In our implementation we make use of a lazy-updates heuristic which reduces the running time for handling requests in the LBU cache; We rebuild the heap every $\delta \cdot M$ requests (for some constant δ). This yields an amortized complexity of $O(1/\delta)$ per request. Whenever a request for an item in the LBU arrives, we update its burstiness score accordingly, and perform a single transition of the item down the heap if valid. Our goal is to keep the heap ordered as possible, while not performing costly $O(\log M)$ operations as possible. Performing this single step takes $O(1)$ operations, and allows propagating less desirable items up the heap, so after a while these may be considered as the victims of the policy.

Furthermore, whenever an item is offered for admittance to the LBU, we only compare it against the item residing in the root of the heap, which encompasses an $O(1)$ toll in terms of complexity. Only in case that an actual eviction is performed (i.e., the item offered for admittance has a higher burstiness score than the item at the root), do we actually evict the item at the root, and push the newly admitted item down the heap, at a cost of $O(\log M)$ in terms of complexity. This heuristic results in an overall amortized complexity of $O(1+1/\delta)$ per request, plus an $O(\log M)$ toll for every actual eviction. It should be noted that this lazy approach might fail to evict the item with the minimum burstiness score (since the item at the root of the heap might not be the item with the minimum burstiness score at the time of eviction). The possible "non-minimality" of the item at the root of the heap increases as we move further away from the most recent rebuilding of the heap (or the most recent eviction), however, since we do partially perform the necessary update in the heap, the items residing near the root of the tree are less desirable then other items that moved down the heap.

In addition to the heap management, aging is applied to each item in the LBU cache when the heap is rebuilt (once every $\delta \cdot M$ requests). Therefore, thanks to the definition of the burstiness score, between heap rebuilds, the values of items in the cache may only increase, thus, an item's score may only grow, thus pushing it further from the root of the heap, and making it a less likely candidate for eviction.

B Artifact Appendix

Abstract

The artifact contains the scripts and utilities used to conduct the experiments described in section 4, and serves as an extendable foundation for future experimentation. The codebase is documented in detail within the README.md file in [35].

Scope

The artifact is intended to provide a straightforward means of reproducing the paper's main results within a fully configured execution environment, while granting complete access to the underlying code to facilitate extension of the proposed methods. Concretely, it contains the code that generated the raw data used to produce Table 4 and the figures presented in section 4.2. The artifact does not, however, include the traces on which the experiments depend; these must be obtained from their respective sources, as described in the artifact's README.

Contents

The artifact consists of the following modules: (i) mechanisms for trace processing, (ii) a fork of the Caffeine's simulator in which the pipeline policy has been implemented, alongside the majority of the contending algorithms, (iii) a script running the experiments performed in this paper, (iv) a script that creates the graphs of section 4.2, (v) the LHD simulator, and (vi) LRB simulator. Components (i)-(iv), which are based on the latest OS and Python versions, were packed into a single container, whereas components (v) and (vi), which require a legacy environment, were packed into a separate container.

The code and configuration files provided in this artifact serve as a reference for further experimentation, including exploration of varying settings. We also provide the Dockerfiles within the artifact to allow creating the container images from scratch.

Hosting

The artifact is publicly available on GitHub [35], with the container images hosted on Docker Hub. The required traces can be retrieved from their original sources: the SNIA IOTTA Repository [55] for the IBM and Twitter traces, and GitHub [6] for the Meta traces.