

An Associativity Threshold Phenomenon in Set-Associative Caches

Michael A. Bender* Rathish Das[†] Martín Farach-Colton[‡] Guido Tagliavini[§]

Abstract

In an α -way set-associative cache, the cache is partitioned into disjoint sets of size α , and each item can only be cached in one set, typically selected via a hash function. Set-associative caches are widely used and have many benefits, e.g., in terms of latency or concurrency, over fully associative caches, but they often incur more cache misses. As the set size α decreases, the benefits increase, but the paging costs worsen.

In this paper we characterize the performance of an α -way set-associative LRU cache of total size k , as a function of $\alpha = \alpha(k)$. We prove the following, assuming that sets are selected using a fully random hash function:

- For $\alpha = \omega(\log k)$, the paging cost of an α -way set-associative LRU cache is within *additive* $O(1)$ of that of a fully-associative LRU cache of size $(1 - o(1))k$, with probability $1 - 1/\text{poly}(k)$, for all request sequences of length $\text{poly}(k)$.
- For $\alpha = o(\log k)$, and for all $c = O(1)$ and $r = O(1)$, the paging cost of an α -way set-associative LRU cache is *not* within a *factor* c of that of a fully-associative LRU cache of size k/r , for some request sequence of length $O(k^{1.01})$.
- For $\alpha = \omega(\log k)$, if the hash function can be occasionally changed, the paging cost of an α -way set-associative LRU cache is within a *factor* $1 + o(1)$ of that of a fully-associative LRU cache of size $(1 - o(1))k$, with probability $1 - 1/\text{poly}(k)$, for request sequences of arbitrary (e.g., super-polynomial) length.

Some of our results generalize to other paging algorithms besides LRU, such as least-frequently used (LFU).

1 Introduction

In an α -way *set-associative cache*, the cache is partitioned into disjoint sets, each of size α , and there is an *indexing function* [57] that selects the unique set into which each item can be placed. The set size α is also known as the *associativity* of the cache.

Set-associative caches are extensively used in hardware [8, 35, 36, 13, 56, 49], because they are faster, use simpler circuitry, and consume less power [28, 29, 44, 33] than *fully associative caches*, which can store any item in any slot in the cache. Set-associative caches are also used as a building block in the design of highly-concurrent software caches [1, 48], because the sets in a set-associative cache are independent and can operate in parallel. The smaller the set size α is, the bigger the benefits are.

However, the smaller the set size α , the worse the *paging cost*, that is, the number of cache misses. This is because an item can only be placed in one of the cache slots in its allocated set, so

*Stony Brook University. bender@cs.stonybrook.edu

[†]University of Liverpool. rathish.das@liverpool.ac.uk

[‡]Rutgers University. martin@farach-colton.com

[§]Rutgers University. guido.tag@rutgers.edu

when more than α frequently accessed items land in the same set, they contend for the slots in the set, causing increased paging.

Thus, when a set-associative cache is being designed, α must be chosen small enough to provide the desired latency, power consumption, chip area or concurrency; and α must be large enough to keep the paging cost low. How can a cache designer optimize for α given this tradeoff? Measuring the benefits of a small α depends on details that are beyond the scope of this paper. Here, we focus on the other side of the tradeoff: *what is the increase in paging cost of an α -way set-associative cache, relative to a fully associative cache, as a function of α ?*

Answering this question is a well-known open problem in the theory [54, 55, 51, 59] and practice [29, 44, 2, 10] of set-associative caches. The only prior theoretical work that partially answered the question did so under the assumption that item requests come from a distribution [54, 55, 51, 59]; their bounds do not hold against an adversary. Proving a worst-case bound in the adversarial setting has remained open.

Existing theory literature has also analyzed alternative types of caches with limited associativity, namely companion caches [16, 39, 17] and two-level caches [43]. However, because these analyses compare the paging cost against the optimal algorithm for the limited-associative cache in consideration, they do not provide any information about the paging cost relative to a fully associative cache, so they do not help the cache designer pick the right associativity to trade off benefits against paging costs.

1.1 This paper

In this paper we solve the open problem of analyzing the paging cost of a set-associative cache as compared to a fully associative cache, with a focus on the case of set-associative least-recently used (LRU) caches. In these caches, when an item is fetched but the designated set is full, the least-recently accessed item from that set is evicted. We assume that the indexing function h is a hash function that selects sets uniformly and independently at random.

Our main results establish a remarkable threshold phenomenon in set-associative caches. Specifically, if a cache of size k has associativity $\alpha = o(\log k)$, then the paging costs are large, even compared to that of a fully-associative cache whose size is only a small fraction of k ; and if $\alpha = \omega(\log k)$, the paging costs are similar to that of a fully-associative cache of roughly the same size k . Thus, on the one hand, cache designers should make sure the associativity is at least logarithmic, and, on the other hand, they should be aware that increasing the associativity beyond logarithmic offers diminishing returns.

Competitive analysis for long request sequences Let k be the size of the cache. We show that if $\alpha = \omega(\log k)$, then α -way set-associative LRU is 1-competitive with fully associative LRU with probability $1 - 1/\text{poly}(k)$, using $(1 + \Theta(\sqrt{\log(k)/\alpha}))$ -resource augmentation, on any request sequence of length at most $\text{poly}(k)$ ¹ (See Section 3 for the formal definitions of competitiveness and resource augmentation.)

In fact, this analysis follows from a more general theorem that establishes a tradeoff among all the parameters involved, namely the set size, the resource augmentation factor, the success probability and the length of the longest supported request sequence. Moreover, the theorem works for a class of paging algorithms that we call *stable*. Roughly speaking, when a stable algorithm is run on a small cache and is given only part of the request sequence, its paging decisions approximate that of the same algorithm running on a large cache and given the whole request sequence. Besides

¹In this paper, $\text{poly}(k)$ means *any* polynomial in k , as long as we choose the hidden constants appropriately.

LRU, this class contains least-frequently used (LFU), and a generalization of LRU, known as LRU- K [41]. This paper introduces and studies the class of stable algorithms, which may be of independent interest.

The limits of set-associative caches Notice that the aforementioned competitive analysis holds for all request sequences up to certain length (which is a function of the parameters involved), but not *arbitrarily long* request sequences. Is our analysis to blame for this or is it a limitation of set-associative caches? More generally, can any of the parameters of the analysis be improved? That is, can we prove 1-competitiveness (or even $O(1)$ -competitiveness) on sub-logarithmic associativity, or on super-polynomial input sequences, or using less than $1 + \Theta(\sqrt{\log(k)/\alpha})$ resource augmentation?

We show that the answer to all of these questions is essentially negative, via a more general tradeoff. We prove that given a set size α , a resource augmentation factor r , and a competitive ratio c , and provided r is not extremely large, there exists an adversarial request sequence that is not too long (as a function of α , r and c), on which α -way set-associative LRU is *not* c -competitive with fully associative LRU using r -resource augmentation. Using this tradeoff we establish that our analysis is tight:

- *Arbitrary length:* Set-associative LRU is *not* competitive on arbitrarily long request sequences.
- *Associativity:* α -way set-associative LRU is *not* c -competitive with fully associative LRU using r -resource augmentation on request sequences of length as small as $O(k^{1.01})$, for any $\alpha = o(\log k)$, $r = O(1)$ and $c = O(1)$.
- *Resource augmentation:* α -way set-associative LRU is *not* c -competitive with fully associative LRU using r -resource augmentation on request sequences of length as small as $O(k^{2.01})$, for any $\alpha = \Omega(\log k)$, $r = 1 + o(\sqrt{\log(k)/\alpha})$ and $c = O(1)$.
- *Super-polynomial length:* α -way set-associative LRU is *not* c -competitive with fully-associative LRU using r -resource augmentation on request sequences of length ℓ , for any $\alpha = \Omega(\log k)$, $r = 1 + \Theta(\sqrt{\log(k)/\alpha})$, $c = \text{poly}(k)$ and super-polynomial ℓ .

In fact, these negative results hold not only for LRU, but for all conservative paging algorithms, which includes FIFO, LFU and clock.

Competitive analysis for arbitrarily long request sequences Set-associative caches fail on long enough input sequences because it does not take too long for an adversary (even one that is oblivious to the random choices of the indexing function) to produce a malicious collection of items that are unevenly distributed into the sets. Such a collection will cause some set to become oversubscribed, thus forcing unnecessary cache misses due to the associativity constraints; these are called **conflict misses** [29]. Repeatedly querying the items from one of these malicious collections drives the miss count up in the set-associative cache, but, importantly, not in a fully associative cache (as long as the collection is smaller than the size of the cache), so the performance gap can be arbitrarily large.

To circumvent this issue, we use **rehashing**: We equip the set-associative cache with the ability to flush all the items and draw a new hash function. We show that rehashing every $\text{poly}(k)$ cache misses makes set-associative LRU almost 1-competitive with fully associative LRU, on request sequences of arbitrary length. Specifically, for $\alpha = \omega(\log k)$, α -way set-associative LRU with rehashing is $(1 + 1/\text{poly}(k))$ -competitive with fully associative LRU with probability $1 - 1/\text{poly}(k)$,

using $(1 + \Theta(\sqrt{\log(k)/\alpha}))$ -resource augmentation, on request sequences on arbitrary length. Furthermore, we show that the rehashing can be done gradually, attaining the same performance guarantees as a stop-the-world-type rehashing.

1.2 Organization

The paper is organized as follows. In Section 2 we describe the related work. In Section 3 we review basic facts about paging algorithms, and present the terminology and probability tools that we will use in the rest of the paper. In Section 4 we present the competitive analysis of set-associative caches for long (but not arbitrarily long) request sequences. In Section 5 we establish the performance limits of set-associative caches. In Section 6 we augment set-associative LRU with the ability to rehash, and analyze it on arbitrarily long request sequences. Finally, in Section 7 we investigate stable algorithms in depth.

2 Related work

The foremost algorithmic technique to analyze the paging cost is competitive analysis [12]. There is a long line of work on competitive analysis of paging algorithms, which mostly focuses on fully associative algorithms [53, 24, 15, 21, 22, 61, 62, 5, 4, 19, 20, 6].

In the reduced-associativity setting (sometimes called *restricted caching*), existing work focuses mostly on two types of caches, namely *two-level caches* [43] and *companion caches* [16, 39, 17], which are generalizations of set-associative caches. Two-level caches are, essentially, set-associative caches where items can be mapped into multiple sets, and copied between any of them for a fraction of the cost of a cache miss. This means that, occasionally, it may be convenient to make room in a set by moving one of its items to another set, instead of evicting it. Companion caches are a combination of a set-associative cache (the *main* cache) with a, typically smaller, fully associative cache (the *companion*)². In every case, the aim is to design online algorithms that are competitive against an optimal offline algorithm for the specific cache organization at hand. The resulting algorithms are not necessarily practical—they are complex, and may not meet the extremely tight latency requirements of hardware caches.

Recent work has considered paging algorithms that are subject to the placement rules of set-associative caches, but not their eviction rules—pages can be evicted from any set, at any time. The idea is to have a set-associative cache mirror the paging decisions of a fully associative paging algorithm, which is simulated on the side. Every time the fully associative algorithm evicts a page, the set-associative cache evicts the same page, even if the the set the victim is occupying is underfull. Crucially, the set-associative cache is resource-augmented with respect to the simulation, so it is unlikely that a set ever becomes full. Using this technique, Bender et al. [11] showed that every online fully associative algorithm can be made to work with the associativity restrictions of set-associative caches. The downside is that simulating a fully associative algorithm is computationally expensive. In a very different context, Frigo et al. [26, Lemma 16] proved a similar simulation result specifically for LRU, albeit using larger resource augmentation.

As the associativity decreases, the performance gap between set- and full-associativity widens. In the extreme case of 1-way set-associative caches, also known as *direct-mapped caches*, Agrawal et al. [7] showed that their paging cost has the undesired property of being a linear factor away from that of the *worst* offline fully associative algorithm, in expectation.

²In the hardware architecture literature, companion caches are known as *victim caches* [31].

Before competitive analysis became the de facto standard for analyzing online algorithms, probabilistic models were the prominent tool. For the analysis of storage systems, the most popular one is the *independent reference model* (IRM) [18, 34, 47], in which every item in the system is accessed independently with some fixed probability. Using the IRM, Smith [54, 55] compared the probability of a cache miss in a fully associative LRU and in a set-associative LRU, assuming an access distribution called *linear paging model*. He concluded that the two probabilities are asymptotically equal, as the size of the sets grows. Of course, because the IRM is a distributional model of the input, IRM analyses do not hold for adversarial workloads, and their conclusions strongly depend on the choice of the access distribution.

Other lines of research include analyzing the paging cost of set-associative caches for specific workloads [50, 25, 27], and emulating external-memory algorithms [3] that were originally designed for fully associative caches in set-associative caches [52].

3 Preliminaries

Paging algorithms A paging algorithm \mathcal{A} is parametrized by the size of the cache it operates on. Given a positive integer k , we denote \mathcal{A}_k the algorithm \mathcal{A} using a cache of size k . Let \mathcal{U} be the universe of items. Let $\sigma \in \mathcal{U}^*$ be a sequence of item requests. The **cost** of \mathcal{A}_k on σ , denoted $C(\mathcal{A}_k, \sigma)$, is defined as the number of cache misses that \mathcal{A}_k incurs throughout σ .

We denote σ_i the i -th element in σ . For any $x \in \mathcal{U}$, we denote σx the sequence that results from appending x to σ . For any $X \subseteq \mathcal{U}$, we write $\sigma[X]$ the subsequence of σ that results from removing every request that is not for an item in X . The notation $x \in \sigma$ means that item x is present in the request sequence σ .

We denote $\mathcal{A}_k(\sigma)$ the set of items cached by \mathcal{A}_k after serving the sequence σ . When the input sequence σ is clear from the context, we denote $\mathcal{A}_k(i)$ the set of items cached by \mathcal{A}_k right after σ_i is served. Denote $\text{Out}(\mathcal{A}_k, \sigma, x) := \mathcal{A}_k(\sigma) \setminus \mathcal{A}_k(\sigma x)$ the set of items evicted by \mathcal{A}_k in response to an access to x right after the sequence σ has been served.

Competitive analysis We compare paging algorithms using competitive analysis [53]. A paging algorithm \mathcal{A} is *c-competitive* with paging algorithm \mathcal{B} **with probability** p using *r-resource augmentation* on request sequences of **length** ℓ if, for every request sequence σ with $|\sigma| \leq \ell$,

$$\Pr [C(\mathcal{A}_k, \sigma) \leq c \cdot C(\mathcal{B}_{k'}, \sigma) + O(1)] \geq p,$$

where $k = rk'$. Here, the probability is taken over the randomness of the algorithms. In this paper, the randomness typically comes from the indexing hash function. We say that \mathcal{A} is **not c-competitive** with \mathcal{B} using r -resource augmentation on request sequences of length ℓ if \mathcal{A} is *not* c -competitive with \mathcal{B} with probability $1/2$ using r -resource augmentation on request sequences of length ℓ . We will always assume that the input σ is oblivious to the random bits used by the algorithms, that is, σ is produced by an **oblivious adversary**. All paging algorithms in this paper are **online**, that is, each request arrives only after the previous one has been served by the algorithm.

Classes of paging algorithms Among the prominent deterministic paging algorithms are least-recently used (LRU), least-frequently used (LFU), first-in first-out (FIFO), and clock. A well-known generalization of LRU, called LRU- K [42, 41, 14], always evicts the item whose K -th most recent

access is the least recent. In particular, regular LRU is LRU-1³. A paging algorithm is *lazy* [37] if it fetches an item from slow memory only when it causes a cache miss, evicts at most one item upon a cache miss, and only evicts an item when the cache is full. LRU- K , FIFO, LFU and clock all fall into this category. For comparison, flush-when-full [32] is a non-lazy algorithm that flushes the whole cache when an item is fetched but every slot is taken. A paging algorithm is *conservative* [60] if it incurs at most k cache misses on any consecutive subsequence of the input that contains at most k distinct items. LRU, FIFO, LFU and clock are all conservative; flush-when-full is *not* conservative.

Concentration bounds We will use the following standard variant of the Chernoff bound (see, e.g., [23]).

Theorem 1. *Let X_1, \dots, X_n be negatively associated 0/1 random variables. Let $X := \sum_i X_i$. For every $\varepsilon \in (0, 1)$,*

$$\Pr[X \geq (1 + \varepsilon) \mathbb{E}[X]] \leq \exp(-\varepsilon^2 \mathbb{E}[X] / 3),$$

and

$$\Pr[X \leq (1 - \varepsilon) \mathbb{E}[X]] \leq \exp(-\varepsilon^2 \mathbb{E}[X] / 2).$$

We will also use the well-known fact that Chernoff bounds are tight.

Theorem 2 ([40]). *Let X_1, \dots, X_n be independent 0/1 random variables with $\Pr[X_i = 1] = p \leq 1/2$ for every i . Let $X := \sum_i X_i$. For every $\varepsilon \in [0, 1/p - 2]$,*

$$\Pr[X \geq (1 + \varepsilon) \mathbb{E}[X]] \geq \frac{1}{4} \exp(-2\varepsilon^2 \mathbb{E}[X]).$$

4 Set-Associative Caches Are Competitive on Long Sequences

We model set-associative and fully associative caches as paging algorithms. For any paging algorithm \mathcal{A} and any $\alpha = \alpha(k)$ that divides k , we define α -*way set-associative* \mathcal{A} as the following paging algorithm.

Algorithm: α -way set-associative \mathcal{A} with cache size k

- Pick a fully random hash function $h : \mathcal{U} \rightarrow [k/\alpha]$.
- Divide the cache into k/α sets of size α slots each, and run a separate instance of \mathcal{A}_α on each set.
- When an item x is requested, execute the request on set $h(x)$.

We denote by $\langle \mathcal{A} \rangle$ the α -way set-associative \mathcal{A} algorithm. We omit α in the notation, as only a single α will be in consideration at all times. Because the case $\alpha = k$ represents a fully associative cache, we always assume $\alpha < k$. To avoid confusion between sets in a cache and the mathematical meaning of a set, we will refer to the sets in a set-associative cache as *buckets*.

The goal of this section is to show that the number of cache misses that $\langle \mathcal{A} \rangle$ incurs is at most approximately the same as that of \mathcal{A} , for any lazy algorithm \mathcal{A} that belongs to a class that we call *stable algorithms*, which we will define shortly (in Section 4.2). In particular, this class contains

³LRU-2 often outperforms LRU [41], because it is less sensitive to isolated accesses, as it only deems an item important when it has been recently accessed at least two times.

the LFU and LRU- K algorithms. At the end of this section we will have proven the following theorem.

Theorem 3. *Let \mathcal{A} be a lazy stable paging algorithm. Let $\delta = \delta(k)$ be such that $\delta \geq \sqrt{12 \ln(k/\alpha)/\alpha}$ and $\delta \leq 1/2$ for all k large enough. Then, α -way set-associative \mathcal{A} is 1-competitive with \mathcal{A} w.p. $1 - \exp(-\delta^2 \alpha/24)$ using $(1-\delta)^{-1}$ -resource augmentation on request sequences of length $O(\exp(\delta^2 \alpha/24))$.*

Roughly speaking, Theorem 3 says that the paging cost of $\langle \mathcal{A} \rangle_k$ is approximately the same as that of $\mathcal{A}_{k'}$, where $k' = (1 - \delta)k$, as long as δ is not too small, and as long as the request sequence is not too long. The tunable parameter δ is the capacity gap between the set-associative cache and its fully associative counterpart, and it controls the tradeoff between the resource augmentation factor, the success probability and the length of the request sequence. On the one hand, to minimize the resource augmentation factor $(1 - \delta)^{-1}$, we want δ to be as small as possible. On the other hand, to maximize the success probability and the sequence length, which are both increasing functions of δ , we want δ to be as large as possible. Perhaps the most interesting region in this tradeoff curve is in the lower end of δ , which asymptotically minimizes the resource augmentation while still providing polynomial success probability and support for polynomial sequences, as the following result shows.

Proposition 1. *Let \mathcal{A} be a lazy stable paging algorithm. Suppose $\alpha = \omega(\log k)$. Then, α -way set-associative \mathcal{A} is 1-competitive with \mathcal{A} w.h.p.⁴ using $(1 + \Theta(\sqrt{\log(k)/\alpha}))$ -resource augmentation on request sequences of length $\text{poly}(k)$.*

Proof. For an arbitrary constant $c > 0$, let $\delta := \sqrt{24c \ln(k)/\alpha}$. Because $\alpha = \omega(\log k)$, δ is in the appropriate range for all k large enough. This choice yields success probability $1 - k^{-c}$ and sequence length $O(k^c)$. The resource augmentation factor satisfies

$$1 + \delta \leq (1 - \delta)^{-1} \leq 1 + 2\delta,$$

where the second inequality is because $\delta \leq 1/2$. Thus, $(1 - \delta)^{-1} = 1 + \Theta(\delta) = 1 + \Theta(\sqrt{\log(k)/\alpha})$. \square

4.1 Proof Outline

Before diving into the details of Theorem 3, we give an outline of its proof for the particular case of $\mathcal{A} = \text{LRU}$. Call an item *stale* if it is older (with respect to the LRU order) than every item in the fully associative cache. Intuitively, evicting items that are *not* stale is bad, because an access to them would cause a hit in the fully associative cache but a miss in the set-associative cache. We call an eviction of a non-stale item a *bad* eviction. The proof is divided into two steps: First, we prove that if few evictions are bad, then the cost of the set-associative cache is at most approximately the same as that of the fully associative cache. Second, we prove that the probability that an eviction is bad is low; this implies that the total number of bad evictions is small.

We upper bound the probability that an eviction is bad using the following chain of observations: If an item x is evicted and x is not stale, then none of the pages in x 's bucket are stale either. This implies that x as well as all other items in x 's bucket are present in the fully associative cache. Therefore, the set of items in the fully associative cache have the undesirable property that, when hashed into the set-associative cache, they cause some bucket to become overfull. Crucially, this set of items is deterministic, so the property is provably unlikely by a balls-and-bins-type analysis of the bucket loads.

⁴With high probability (w.h.p.) means with probability $1 - 1/\text{poly}(k)$.

4.2 Stable Paging Algorithms

Notice that in the proof outline we have used the following property of LRU: if a bucket in the set-associative cache evicted an item x that is present in the fully associative cache, then all items in x 's bucket are also present in the fully associative cache. Stable algorithms formalize this property.

We say that a paging algorithm \mathcal{A} is **stable** if, for every $\tau \in \mathcal{U}^*$, $X \subseteq \mathcal{U}$, $z \in X$, and positive integers $a > b$,

$$\text{Out}(\mathcal{A}_b, \tau[X], z) \cap \mathcal{A}_a(\tau z) \neq \emptyset \text{ implies } \mathcal{A}_b(\tau[X]z) \subseteq \mathcal{A}_a(\tau z). \quad (1)$$

In words, if an instance of \mathcal{A} is given a request sequence τz , a smaller instance is given a subsequence $\tau[X]z$, and if in response to the access to the final item z the smaller cache evicts an item that is also present in the larger cache, then the smaller cache is fully contained in the larger cache.

We can also think of stability as a seamless mechanism that corrects the differences between any two caches managed by instances of \mathcal{A} . Roughly speaking, the contrapositive of Eq. (1) says that \mathcal{A}_b always evicts items that are not present in \mathcal{A}_a first. Thus, intuitively, stable algorithms are effective at managing the buckets of a set-associative cache because they try to approximate the state of a larger fully associative cache.

We defer a thorough discussion of stable algorithms to Section 7, in which we investigate the scope of this class of algorithms, and provide insight into the kind of features that distinguish paging algorithms that are stable from those that are not. The following fact is a corollary of Section 7.

Lemma 1. *LRU-K and LFU are stable. FIFO and clock are not stable.*

4.3 Auxiliary Results

Let \mathcal{X} and \mathcal{Y} be any two paging algorithms, with arbitrary cache sizes. We say that an item x is a **bad eviction** of \mathcal{X} with respect to \mathcal{Y} at time i (we omit \mathcal{X} , \mathcal{Y} or i when they are clear from the context) if x is evicted by \mathcal{X} due to access σ_i and $x \in \mathcal{Y}(i)$, that is, $x \in \text{Out}(\mathcal{X}, \sigma_1 \dots \sigma_{i-1}, \sigma_i) \cap \mathcal{Y}(i)$.

Lemma 2. *Let \mathcal{X} and \mathcal{Y} be any two paging algorithms with arbitrary cache sizes, such that they only fetch a page when it is requested. Then,*

$$C(\mathcal{X}, \sigma) \leq C(\mathcal{Y}, \sigma) + B,$$

where B is the number of bad evictions of \mathcal{X} with respect to \mathcal{Y} throughout σ .

Proof. Observe that $C(\mathcal{X}, \sigma) - C(\mathcal{Y}, \sigma) = M - H$, where H is the number of cache hits in \mathcal{X} that are misses in \mathcal{Y} , and M is the number of misses in \mathcal{X} that are hits in \mathcal{Y} . We call the latter **bad misses**. Thus, it suffices to show that $M \leq B$.

Suppose σ_i is a bad miss. Notice that σ_i must have already been accessed before time i , or else it wouldn't be present in \mathcal{Y} at time i , as items are only fetched when they are requested. Therefore, σ_i must have been evicted by \mathcal{X} in the past. Let $j < i$ be the most recent time when σ_i was evicted by \mathcal{X} . It is enough to show that σ_i was a bad eviction (of \mathcal{X} with respect to \mathcal{Y} at time j); this will establish an injection from bad misses into bad evictions, as there cannot be two misses of the same item without being evicted in-between. Thus, to conclude the proof, we need to show that $\sigma_i \in \mathcal{Y}(j)$.

During the time interval (j, i) , item σ_i was not accessed, or else j wouldn't be the most recent eviction time of σ_i in \mathcal{X} . Since σ_i is a hit in \mathcal{Y} at time i , item σ_i was not accessed during (j, i) , and items are only fetched upon a request, we must have $\sigma_i \in \mathcal{Y}(j)$. Hence, σ_i was a bad eviction of \mathcal{X} with respect to \mathcal{Y} at time j . \square

In particular, Lemma 2 holds for lazy algorithms. (Later on, in Section 6, we will use this result in full generality, on paging algorithms that perform multiple evictions during a time step.)

Next, we prove a standard balls-and-bins bound for our particular setting.

Lemma 3. *Let $\delta = \delta(k)$ be such that $\delta \geq \sqrt{12 \ln(k/\alpha)/\alpha}$ and $\delta \leq 1/2$ for all k large enough. Suppose $k' = (1 - \delta)k$ balls are randomly and independently thrown into k/α bins. Then, for all k large enough, the maximum load among all bins is greater than α with probability at most $\exp(-\delta^2 \alpha/12)$.*

Proof. Let $m := k' = (1 - \delta)k$ be the number of balls, and let $n := k/\alpha$ be the number of bins. Let $h := m/n = (1 - \delta)k/(k/\alpha) = (1 - \delta)\alpha$ be the average load of a bin. Let L_i be the load of bin i . This is a sum of m independent binary random variables, each indicating whether a ball lands in bin i . By the Chernoff bound from Theorem 1,

$$\begin{aligned} \Pr[L_i \geq (1 + \delta) \mathbb{E}[L_i]] &\leq \exp\left(-\frac{\delta^2 \mathbb{E}[L_i]}{3}\right) \\ &= \exp\left(-\frac{\delta^2 (1 - \delta)\alpha}{3}\right) \quad (\text{as } \mathbb{E}[L_i] = h = (1 - \delta)\alpha) \\ &\leq \exp\left(-\frac{\delta^2 \alpha}{6}\right). \quad (\text{as } 1 - \delta \geq 1/2) \end{aligned}$$

Notice that

$$(1 + \delta) \mathbb{E}[L_i] = (1 + \delta)(1 - \delta)\alpha = (1 - \delta^2)\alpha < \alpha.$$

Hence, the load of bin i is larger than α with probability at most $\exp(\delta^2 \alpha/6)$. By a union bound over all n bins, the maximum load across all bins is larger than α with probability

$$\begin{aligned} n \exp(-\delta^2 \alpha/6) &= \frac{k}{\alpha} \exp(-\delta^2 \alpha/6) \quad (\text{by definition of } n) \\ &= \exp\left(-\frac{\delta^2 \alpha}{6} + \ln\left(\frac{k}{\alpha}\right)\right) \\ &\leq \exp\left(-\frac{\delta^2 \alpha}{12}\right). \quad (\text{as } \delta^2 \alpha/6 \geq 2 \ln(k/\alpha), \text{ by hypothesis}) \end{aligned}$$

This concludes the proof. \square

4.4 Proof of Theorem 3

For every time step i , let B_i be the indicator random variable that is 1 when σ_i causes a bad eviction in $\langle \mathcal{A} \rangle_k$ with respect to $\mathcal{A}_{k'}$.

Claim 1. *We have $\Pr[B_i = 1] \leq \exp(-\delta^2 \alpha/12)$.*

Proof. Suppose an item x is evicted by $\langle \mathcal{A} \rangle_k$ in response to σ_i . Then, σ_i must hash to the same bucket as x . Suppose x is a bad eviction. Let $\tau := \sigma_1 \dots \sigma_{i-1}$. Then, $x \in \text{Out}(\langle \mathcal{A} \rangle_k, \tau, \sigma_i) \cap \mathcal{A}_{k'}(i)$. Let $X := \{y \in \mathcal{U} \mid h(y) = h(x)\}$ be the set of items that hash to x 's bucket. Then, $x \in \text{Out}(\mathcal{A}_\alpha, \tau[X], \sigma_i) \cap \mathcal{A}_{k'}(i)$, because x is evicted from the set-associative cache if and only if it is evicted from its bucket. Because \mathcal{A} is stable, this implies that $\mathcal{A}_\alpha(\tau[X]\sigma_i) \subseteq \mathcal{A}_{k'}(\tau\sigma_i)$, that is, all items in x 's bucket are also in $\mathcal{A}_{k'}(i)$. Thus, if $S_i := \mathcal{A}_{k'}(i)$, we conclude that

$$\Pr[B_i = 1] \leq \Pr[\text{There are } \alpha + 1 \text{ items in } S_i \text{ that hash to the same bucket}].$$

To upper bound this probability we model the event as a balls-and-bins game, in which each ball represents an item, and each bin represents a bucket. Crucially, the set S_i is oblivious to the randomness of $\langle \mathcal{A} \rangle_k$, so the ball placements are independent and uniformly random. There are at most $|S_i| \leq k'$ balls and k/α bins, so by Lemma 3 this probability is at most $\exp(-\delta^2\alpha/12)$. \square

Let $B := \sum_{i=1}^{|\sigma|} B_i$ be the number of bad evictions; there is at most 1 bad eviction at each step, since \mathcal{A} is lazy. By Claim 1 and linearity of expectation, $\mathbb{E}[B] \leq |\sigma| \exp(-\delta^2\alpha/12)$. By Markov's inequality, $\Pr[B \geq |\sigma| \exp(-\delta^2\alpha/24)] \leq \exp(-\delta^2\alpha/24)$.

Because \mathcal{A} is lazy, we may apply Lemma 2 to conclude that

$$C(\langle \mathcal{A} \rangle_k, \sigma) \leq C(\mathcal{A}_{k'}, \sigma) + |\sigma| \exp(-\delta^2\alpha/24),$$

with probability $1 - \exp(-\delta^2\alpha/24)$. When $|\sigma| = O(\exp(\delta^2\alpha/24))$, this implies that $\langle \mathcal{A} \rangle_k$ is 1-competitive against $\mathcal{A}_{k'}$. This concludes the proof.

5 Set-Associative Caches Are Not Competitive on Sequences of Arbitrary Length

The major limitation of Theorem 3 is that it does not work for request sequences of arbitrary length. That is, for any valid choice of α and δ , competitiveness of the set-associative cache is only guaranteed up to a certain number of requests. A natural question is whether we can extend this result to arbitrarily long input sequences, perhaps by allowing for a larger resource augmentation factor, or for a larger competitive ratio.

The main result of this section is a negative answer to this question for all conservative algorithms, which includes LRU.

Theorem 4. *Let \mathcal{A} be a conservative paging algorithm. Suppose that $|\mathcal{U}| \geq 16k \exp(8(1-\delta)^{-1}\delta^2\alpha)$ and $(1-\delta)^{-1} \leq k/(2\alpha)$ for all large enough k . Then, α -way set-associative \mathcal{A} is not c -competitive with \mathcal{A} using $(1-\delta)^{-1}$ -resource augmentation on request sequences of length*

$$O(c\alpha k \exp(16(1-\delta)^{-1}\delta^2\alpha)).$$

Roughly speaking, Theorem 4 states that as long as the resource augmentation $r = (1-\delta)^{-1}$ is not too large and the universe \mathcal{U} is large enough, there is a hard limit on the length of the request sequences that a set-associative cache can handle before its performance drops.

Corollary 1. *Let \mathcal{A} be a conservative paging algorithm. Suppose \mathcal{U} , α and $r = (1-\delta)^{-1}$ satisfy the hypotheses of Theorem 4. Then, for every $c \geq 1$, α -way set-associative \mathcal{A} is not c -competitive with \mathcal{A} using r -resource augmentation on arbitrarily long request sequences.*

This corollary applies, for instance, to all $\alpha \leq k/8$ and $r \leq 4$, provided that the universe is large enough. We can further use Theorem 4 to show that competitiveness is impossible in some common regimes of α , r and c . For comparison, recall that 1-competitiveness on polynomial request sequences is possible for $\alpha = \omega(\log k)$ and $r = 1 + \Theta(\sqrt{\log(k)/\alpha})$, by Proposition 1.

Proposition 2. *Let \mathcal{A} be a conservative paging algorithm. Suppose (for simplicity) that \mathcal{U} is infinite. In the following cases, α -way set-associative \mathcal{A} is not c -competitive with \mathcal{A} using r -resource augmentation on request sequences of length $O(\alpha k^{1+o(1)}) = O(k^{2.01})$:*

1. for $\alpha = \Omega(\log k)$ such that $\alpha \leq k/3$, $r = 1 + o(\sqrt{\log(k)/\alpha})$ and $c = O(1)$;

2. for $\alpha = o(\log k)$, $r = O(1)$ and $c = O(1)$;

3. for $\alpha = 1$, $r = o(\log k)$ and $c = O(1)$.

Proof. The proof of all three cases is similar, so we will only prove the first one. Let $\delta := 1 - 1/r$. Notice that $(1 - \delta)^{-1} = r$. Because $\alpha = \Omega(\log k)$, we have $r = 1 + o(1)$ and, therefore, $r = 1 + o(1) \leq 3/2 \leq k/(2\alpha)$, so we are in the conditions of Theorem 4. Thus, α -way set-associative \mathcal{A} is not c -competitive with \mathcal{A} using r -resource augmentation on request sequences of length

$$\begin{aligned} O(c\alpha k \exp(16r\delta^2\alpha)) &= O(\alpha k \exp(16r\delta^2\alpha)) && (\text{as } c = O(1)) \\ &= O(\alpha k \exp(o(\log k))) && (\text{as } \delta = o(\sqrt{\log(k)/\alpha}) \text{ and } r = 1 + o(1)) \\ &= O(\alpha k^{1+o(1)}). \end{aligned}$$

□

Interestingly, part 3 in Proposition 2 implies that direct-mapped caches (i.e., set-associative caches with $\alpha = 1$) are *not* constant-competitive with a fully associative cache using sub-logarithmic resource augmentation on request sequences as small as $O(k^{1+o(1)})$. This justifies the conventional wisdom that the performance of direct-mapped caches is subpar.

Parts 1 and 2 in Proposition 2 imply that Proposition 1 is optimal, in the sense that the set size and resource augmentation factor cannot be shrunk, even if we permit competitive ratio $O(1)$ and sub-polynomial success probability (but at least $1/2$). The following proposition shows that, in fact, the polynomial length of the request sequences in Proposition 1 cannot be improved either, even if the competitive ratio is polynomial.

Proposition 3. *Suppose that \mathcal{U} is infinite. For $\alpha = \Omega(\log k)$ such that $\alpha \leq k/3$, $c = \text{poly}(k)$ and $r = 1 + \Theta(\sqrt{\log(k)/\alpha})$, α -way set-associative \mathcal{A} is not c -competitive with \mathcal{A} using r -resource augmentation on request sequences of super-polynomial length.*

The proof is analogous to that of Proposition 2, so we omit it.

5.1 An Auxiliary Result

Consider a balls-and-bins game, in which m balls are thrown randomly and independently into n bins. Let $h := m/n$. A bin is said to be *a -saturated* if, at the end of the game, the bin contains at least $h + a$ balls. Let $f(n, m, \varepsilon) := n \exp(-2\varepsilon^2 h)$.

Lemma 4. *Suppose m balls are thrown into $n \geq 2$ bins. Let $h := m/n$. For every $\varepsilon \in [0, n-2]$, the number of εh -saturated bins is more than $f(n, m, \varepsilon)/8$ with probability at least $1 - \exp(-f(n, m, \varepsilon)/32)$.*

Proof. Let X_i be an indicator random variable that is 1 exactly when bin i is εh -saturated. Let L_i be the load of bin i . Then, $\Pr[X_i = 1] = \Pr[L_i \geq h + \varepsilon h]$. The random variable L_i is a sum of m i.i.d. 0/1 random variables, each one indicating whether a ball lands in bin i . In particular, $\mathbb{E}[L_i] = h$. Because each ball has probability $1/n \leq 1/2$ of landing in bin i , and $\varepsilon \in [0, n-2]$, we have, by the reverse Chernoff bound from Theorem 2, that

$$\Pr[L_i \geq h + \varepsilon h] \geq \frac{1}{4} \exp(-2\varepsilon^2 h).$$

Let $X := \sum_i X_i$ be the number of εh -saturated bins. By linearity of expectation, $\mathbb{E}[X] = \sum_i \mathbb{E}[X_i] \geq f(n, m, \varepsilon)/4$.

Using standard results about negatively associated random variables [58], one can show that the random variables X_1, \dots, X_n are negatively associated. Thus, by the Chernoff bound,

$$\begin{aligned} \Pr[X \leq f(n, m, \varepsilon)/8] &\leq \Pr[X \leq \mathbb{E}[X]/2] \\ &\leq \exp(-\mathbb{E}[X]/8) \leq \exp(-f(n, m, \varepsilon)/32). \end{aligned}$$

□

5.2 Proof of Theorem 4

Consider the request sequence produced by the following adversary.

Adversary: Request sequence of Theorem 4

1. Let $s := 16 \exp(8(1 - \delta)^{-1} \delta^2 \alpha)$ and $t := c\alpha s^2$.
2. Pick $S_1, \dots, S_s \subseteq \mathcal{U}$ disjoint sets of $k' = (1 - \delta)k$ items.
3. For $i = 1, \dots, s$:
 - (a) Repeat t times:
 - i. Sequentially access every item in S_i .

Notice that the adversary can carry out step 2 because the universe is large enough, by assumption. Let σ be the request sequence produced. Its length is

$$|\sigma| = k't < kt = kc\alpha s^2 = O(c\alpha k \exp(8(1 - \delta)^{-1} \delta^2 \alpha)^2),$$

as required. We will show that $C(\langle \mathcal{A} \rangle_k, \sigma) \geq cC(\mathcal{A}_{k'}, \sigma) + \omega(1)$ with probability strictly larger than $1/2$, which implies the claim.

Because $\mathcal{A}_{k'}$ is conservative, it only misses the first time each page in S_i is accessed. Thus, $C(\mathcal{A}_{k'}, \sigma) = k's$. Now we analyze the cost of $\langle \mathcal{A} \rangle_k$ on σ .

Claim 2. *In each iteration of the loop of step 3, there are at least $2c\alpha s$ evictions in $\langle \mathcal{A} \rangle_k$ with probability at least $1 - \exp(-k/(2\alpha s))$.*

Proof. Fix an execution of step 3-a-i. We model the accesses to items in S_i as a balls-and-bins game, in the usual way: each ball represents an item, and each bin represents a bucket. There are $m = (1 - \delta)k$ balls thrown into $n = k/\alpha$ bins; these are at least 2 bins, since $\alpha \leq k/2$. Let $h := m/n = (1 - \delta)\alpha$. Let $\varepsilon := 2\delta/(1 - \delta)$. Notice that $\varepsilon \leq n - 2$, because this is equivalent to the assumption $(1 - \delta)^{-1} \leq k/(2\alpha)$. We have

$$h + \varepsilon h = (1 + \varepsilon)h = (1 + \varepsilon)(1 - \delta)\alpha > \alpha.$$

In the last inequality we have used the choice of ε . Thus, if a bin is εh -saturated, its associated bucket incurred at least one eviction, regardless of the state of $\langle \mathcal{A} \rangle_k$ at the beginning of the step. Let X be the number of εh -saturated bins. The number of evictions in $\langle \mathcal{A} \rangle_k$ during this step is, hence, at least X .

Crucially, for a fixed iteration of the loop of step 3 (the outer loop) the random bin choices in the balls-and-bins game of *every* iteration of the loop of step 3-a (the inner loop) are the same. Thus, the number of evictions during a fixed iteration of the outer loop is at least tX .

Notice that

$$\begin{aligned}
f(n, m, \varepsilon) &= f\left(\frac{k}{\alpha}, (1 - \delta)k, \frac{2\delta}{1 - \delta}\right) \\
&= \frac{k}{\alpha} \exp\left(-2 \left(\frac{2\delta}{1 - \delta}\right)^2 (1 - \delta)\alpha\right) \\
&= \frac{k}{\alpha} \exp\left(-\frac{8\delta^2\alpha}{1 - \delta}\right) = \frac{16k}{\alpha s}.
\end{aligned}$$

By Lemma 4, $X \geq f(n, m, \varepsilon)/8 = 2k/(\alpha s)$ with probability at least $1 - \exp(-f(n, m, \varepsilon)/32) = 1 - \exp(-k/(2\alpha s))$. The proof concludes by noting that $t(2k/(\alpha s)) = 2csk$. \square

Hence, the cost of each iteration of the loop of step 3 is lower bounded by

$$2csk \geq csk' + \Omega(k) = cC(\mathcal{A}_{k'}, \sigma) + \Omega(k),$$

with probability $1 - \exp(-k/(2\alpha s))$. Because the S_i 's are disjoint, and the hash function is fully independent, this event takes place independently across all iterations of the loop of step 3. Hence, the probability that some iteration of step 3 has cost at least $cC(\mathcal{A}_{k'}, \sigma) + \Omega(k)$ is at least

$$1 - \exp(-k/(2\alpha s))^s = 1 - \exp(-k/(2\alpha)) \geq 1 - e^{-1} \approx 0.63.$$

Therefore, $C(\langle \mathcal{A} \rangle_k, \sigma) \geq cC(\mathcal{A}_{k'}, \sigma) + \omega(1)$ with probability at least 0.63.

6 From Long Sequences To All Sequences

The proof of Theorem 4 shows that if we wish $\langle \mathcal{A} \rangle$ to remain competitive on all request sequences, it must be immune to adversarial sequences that exploit repetition. A natural way to attempt this is by providing the algorithm with the ability to draw new hash functions, or **rehashing**. Then, whenever the algorithm deems the current hash function compromised, it transitions to a new one.⁵

We will rehash every $\text{poly}(k)$ *cache misses*. Notice that running periodic rehashes after some fixed number of accesses (independent of the number of cache misses) is detrimental, because it gives the adversary infinitely many opportunities to exploit load imbalances in the allocations. In fact, the adversary can now simply fix a single set of $k' = (1 - \delta)k$ items, and sequentially access every element in the set, ad infinitum.

A downside of rehashing is that it can potentially hurt the paging cost, because it “cools down” the buckets. Upon a rehash, buckets must abruptly switch to caching a different set of items, of which they have limited (or no) information about past accesses. When the paging decisions of \mathcal{A} are largely dependent on the full history of accesses (e.g., in LFU), bucket instances become impaired after a rehash, and may never catch up.

A paging algorithm that does not suffer from this problem is LRU. This is because LRU depends only on the most recent access of the items, so as soon as an item x is accessed for the first time after a rehash, the new bucket assigned to x has the most up-to-date access information about x . In the rest of this section we focus on $\mathcal{A} = \text{LRU}$.

When a new hash function is drawn, the items currently hashed with the old function must be remapped. Consider the following simple method, that we call **full flushing**: When a rehash is

⁵Interestingly, rehashing has been used in recent cache designs to protect from timing attacks that exploit conflict misses [45, 46].

triggered, flush the cache and draw a new hash function. If those items are accessed again, they will be placed in their new buckets.

Algorithm: α -way set-associative LRU with full flushing and cache size k

- Augment α -way set-associative LRU with cache size k as follows:
 - Keep a counter of the number of cache misses.
 - Let $d \geq 2$ be an arbitrary constant. Every k^d misses evict all the items in cache, draw a new hash function, and reset the counter.

We denote this algorithm $\langle \text{LRU} \rangle_k^{\text{FF}}$.

Theorem 5. *Suppose $\alpha = \omega(\log k)$. Then, α -way set-associative LRU with full flushing is $(1 + 1/\text{poly}(k))$ -competitive with LRU w.h.p. using $(1 + \Theta(\sqrt{\log(k)/\alpha}))$ -resource augmentation on all request sequences.*

Proof. For an arbitrary constant $c > d$, let $\delta := \sqrt{36c \ln(k)/\alpha}$. Let $k' := (1 - \delta)k$. We will compare the paging cost of $\langle \text{LRU} \rangle_k^{\text{FF}}$ with that of $\text{LRU}_{k'}$.

Partition σ into phases as follows. Each phase is a maximal sequence of time steps in which $\text{LRU}_{k'}$ incurs exactly one cache miss. (The only miss takes place in the first access of the phase.) The first phase starts at the beginning of σ , and all other phases start right after the previous one. Let π_r be the r -th phase. We further partition each π_r into subphases. Each subphase starts either at the beginning of π_r or when a rehash is triggered in $\langle \text{LRU} \rangle_k^{\text{FF}}$, and extends until right before the next rehash or the end of π_r , whatever comes first. We denote π_r^s the s -th subphase within π_r .

Let i_r be the time step when π_r begins, and let $S_r := \text{LRU}_{k'}(i_r)$. Note that because $\text{LRU}_{k'}$ only misses on the very first access of π_r , the set of items cached by $\text{LRU}_{k'}$ is invariantly S_r during the phase.

Divide the bad evictions of $\langle \text{LRU} \rangle_k^{\text{FF}}$ into two classes: those that are due to the flushes, and the rest. We call the latter ones **regular bad evictions**. Let F be the number of bad evictions due to flushes, and let R be the number of regular bad evictions. By Lemma 2,

$$C(\langle \text{LRU} \rangle_k^{\text{FF}}, \sigma) \leq C(\text{LRU}_{k'}, \sigma) + R + F. \quad (2)$$

Since the number of rehashes is at most a $1/k^d$ -fraction of the total number of cache misses of $\langle \text{LRU} \rangle_k$, and every rehash causes k evictions,

$$F \leq k^{-d+1} C(\langle \text{LRU} \rangle_k^{\text{FF}}, \sigma). \quad (3)$$

Combining Eq. (2) and Eq. (3),

$$\begin{aligned} C(\langle \text{LRU} \rangle_k^{\text{FF}}, \sigma) &\leq \frac{1}{1 - k^{-d+1}} (C(\text{LRU}_{k'}, \sigma) + R) \\ &\leq (1 + 2k^{-d+1}) (C(\text{LRU}_{k'}, \sigma) + R). \end{aligned} \quad (4)$$

Claim 3. *We have $\Pr[\text{There is a regular bad eviction during } \pi_r^s] \leq k^{-3c}$* 6

⁶This claim is similar to Claim 1 but it differs in two crucial ways: it deals with rehashes, and it targets a whole subphase (which can be arbitrarily long) as opposed to a single point in time.

Proof. Suppose a regular bad eviction of some item x in $\langle \text{LRU} \rangle_k$ occurs during π_r^s , say at time step j . Then, $x \in \text{LRU}_{k'}(j)$. By definition of LRU, every other item in x 's bucket has been accessed more recently than x . This implies that all of those items are in $\text{LRU}_{k'}(j)$ as well.⁷ Thus, there are $\alpha + 1$ items in $\text{LRU}_{k'}(j)$ that hash to the same bucket. Furthermore, since the set of cached items in $\text{LRU}_{k'}$ doesn't change during a phase, we have $\text{LRU}_{k'}(j) = S_r$, so, in fact, all the items in x 's bucket are in S_r . Hence,

$$\begin{aligned} & \Pr[\text{There is a regular bad eviction during } \pi_r^s] \\ & \leq \Pr[\text{There are } \alpha + 1 \text{ items in } S_r \text{ that hash to the same bucket}]. \end{aligned} \quad (5)$$

We upper bound this probability using a balls-and-bins analysis. Crucially, S_r is a deterministic set. The choice of δ satisfies the hypothesis of Lemma 3, so the probability that S_r causes a bucket overflow is at most $\exp(-\delta^2\alpha/12) = k^{-3c}$, as desired. \square

Claim 4. *If there is no regular bad eviction during π_r^s , then it's either the first subphase or the last one in π_r .*

Proof. We prove the contrapositive. Suppose π_r^s is neither the first nor the last subphase. Then, π_r^s begins with a rehash, and ends right before the following rehash. Thus, there are k^d cache misses in $\langle \text{LRU} \rangle_k^{\text{FF}}$ during π_r^s . Because only items from S_r are requested during π_r , and $\langle \text{LRU} \rangle_k^{\text{FF}}$ misses more than $k > |S_r|$ times during π_r^s , there exists some item $x \in S_r$ that was evicted by $\langle \text{LRU} \rangle_k^{\text{FF}}$ at least once during π_r^s . \square

Let R_r be the number of regular bad evictions during π_r .

Claim 5. *We have $\mathbb{E}[R_r] \leq O(k^{-2c})$.*

Proof. Since every subphase can have at most k^d misses, it can also have at most k^d evictions. In particular, it can have at most k^d regular bad evictions. Thus,

$$\Pr[R_r \geq \ell] \leq \Pr[\text{At least } \lceil \ell/k^d \rceil \text{ subphases in } \pi_r \text{ have regular bad evictions}].$$

If $\lceil \ell/k^d \rceil$ subphases in π_r have regular bad evictions, then, by Claim 4, either $\pi_r^1, \pi_r^2, \dots, \pi_r^{\lceil \ell/k^d \rceil}$ all have regular bad evictions, or $\pi_r^2, \pi_r^3, \dots, \pi_r^{\lceil \ell/k^d \rceil + 1}$ all do, because all intermediate subphases have regular bad evictions. Thus, by union bound,

$$\begin{aligned} \Pr[R_r \geq \ell] & \leq \Pr\left[\pi_r^1, \pi_r^2, \dots, \pi_r^{\lceil \ell/k^d \rceil} \text{ all have regular bad evictions}\right] \\ & \quad + \Pr\left[\pi_r^2, \pi_r^3, \dots, \pi_r^{\lceil \ell/k^d \rceil + 1} \text{ all have regular bad evictions}\right] \\ & \leq 2(k^{-3c})^{\lceil \ell/k^d \rceil}. \end{aligned} \quad (6)$$

(because subphases use independent hash functions, and by Claim 3)

Finally,

$$\begin{aligned} \mathbb{E}[R_r] & = \sum_{\ell \geq 1} \Pr[R_r \geq \ell] = \sum_{\ell \geq 1} 2(k^{-3c})^{\lceil \ell/k^d \rceil} \\ & = 2k^d \sum_{a \geq 1} (k^{-3c})^a \leq k^d O(k^{-3c}) \\ & = O(k^{-2c}). \end{aligned} \quad (\text{as } c > d)$$

⁷Notice that this part of the proof would fail for LRU- K (with $K \geq 2$) and LFU, because rehashes clear the access history in the set-associative cache.

□

Observe that the number of phases is exactly $C(\text{LRU}_{k'}, \sigma)$. By Claim 5 and linearity of expectation, $\mathbb{E}[R] \leq C(\text{LRU}_{k'}, \sigma) O(k^{-2c})$. By Markov's inequality, $R \geq C(\text{LRU}_{k'}, \sigma) O(k^{-c})$ with probability $O(k^{-c})$. Plugging this into Eq. (4), we get that

$$\begin{aligned} C(\langle \text{LRU} \rangle_k^{\text{FF}}, \sigma) &\leq (1 + 2k^{-d+1})(1 + O(k^{-c})) C(\text{LRU}_{k'}, \sigma) \\ &\leq (1 + O(k^{-d+1})) C(\text{LRU}_{k'}, \sigma) \end{aligned} \quad (\text{as } c > d)$$

with probability $1 - O(k^{-c})$. This concludes the proof. □

6.1 Incremental Flushing

The downside of full flushing is that it brings the execution to a halt and performs up to k uninterrupted writes to slow memory. Alternatively, we can deamortize the flushing, spreading out the evictions over time. We call this method *incremental flushing*, and it works as follows: When a rehash is triggered, we draw a new hash function, and immediately resume the execution. At k arbitrary points in time before the next rehash, pick an arbitrary cached item that is still not remapped (i.e., it was cached when the rehash began, and it is still inhabiting its old bucket), and evict it. If during the course of the execution a non-remapped item is hit, we insert it into the new bucket (potentially causing an eviction, if the new bucket is full), and command the LRU instance of the old bucket to delete it. The rehash concludes when there are no more items mapped with the old function; at that point we discard the old hash function. Importantly, because every rehash finishes before the next one begins, there are at most two hash functions in use at all times. On every access, we search the requested item using both of them.

With incremental flushing, however, two complications arise in the competitive analysis from Theorem 5, stemming from the fact that there are two hash functions in use simultaneously. The first one is that because the indexing function is changing throughout a subphase, the hash function in the upper bound from Eq. (5) depends on the time at which the bad eviction occurs. Fortunately, this does not require a fix, because no matter the particular moment in time, the changing hash function will always be fully random, which is all we need to further upper bound the probability via the balls-and-bins process.

The second issue is that the random placements *before* a rehash begins are *not* independent of the placements *after* the rehash begins. This means that the random choices in the subphases are no longer independent, and the step following Eq. (6) does not hold. There is a simple fix for this: Because every hash function only lives during at most two consecutive subphases, we can bound the probability that a bad eviction occurs on *every other* subphase. Thus, the bound becomes

$$\Pr[R_r \geq \ell] \leq 2(k^{-3c})^{\lceil \ell/k^d \rceil / 2},$$

which only changes the constants in the end result. The rest of the proof holds without changes.

We denote α -way set-associative LRU with incremental flushing as $\langle \text{LRU} \rangle^{\text{IF}}$.

Proposition 4. *Theorem 5 holds for $\langle \text{LRU} \rangle^{\text{IF}}$.*

6.2 Competitive Analysis Against the Optimal Offline Algorithm

It is well known that LRU is $(1 + 1/(r - 1))$ -competitive against the offline optimal fully associative paging algorithm OPT, using r -resource augmentation [53]. Combining this with Theorem 5 and Proposition 4, we can compare set-associative LRU caches with OPT.

Proposition 5. Suppose $\alpha = \omega(\log k)$. For all $r > 1$, $\langle \text{LRU} \rangle^{FF}$ and $\langle \text{LRU} \rangle^{IF}$ are $(1 + 1/(r - 1) + o(1))$ -competitive with OPT w.h.p. using $(1 + o(1))r$ -resource augmentation on all request sequences.

Proof. Let $k' = k/(1 + \Theta(\sqrt{\log(k)/\alpha}))$ and $k'' = k'/r$. We will compare $\langle \text{LRU} \rangle_k^{FF}$ with $\text{OPT}_{k''}$. The resource augmentation is, thus, $k/k'' = (1 + \Theta(\sqrt{\log(k)/\alpha}))r$. We have, with high probability,

$$\begin{aligned} C(\langle \text{LRU} \rangle_k^{FF}, \sigma) &\leq (1 + 1/\text{poly}(k)) C(\text{LRU}_{k'}, \sigma) && \text{(by Theorem 5)} \\ &\leq (1 + 1/\text{poly}(k))(1 + 1/(r - 1)) C(\text{OPT}_{k''}, \sigma) \\ &\quad \text{(by the competitive analysis of LRU)} \\ &= (1 + 1/(r - 1) + 1/\text{poly}(k)) C(\text{OPT}_{k''}, \sigma). \end{aligned}$$

For $\langle \text{LRU} \rangle^{IF}$ the proof is, of course, analogous. \square

In particular, setting $r = 2$, we conclude that set-associative LRU caches are $(2 + o(1))$ -competitive with OPT w.h.p. using $(2 + o(1))$ -resource augmentation on all request sequences.

7 The Class of Stable Paging Algorithms

In this section we set out to understand the class of stable algorithms. We wish to answer the following questions: Which known paging algorithms are stable? Which ones are not? More generally, are there necessary and/or sufficient conditions for an algorithm to be stable?

Throughout this section we assume that items come from a totally ordered space. As usual, all the paging algorithms we consider are online.

7.1 Stack Algorithms

Stable algorithms are closely related to stack algorithms [38]. A paging algorithm \mathcal{A} is a **stack algorithm** if $\mathcal{A}_k(\sigma) \subseteq \mathcal{A}_{k+1}(\sigma)$ for every σ and k . The main property of stack algorithms is that they avoid Belady's anomaly⁸ because there cannot be a cache miss on a cache of size $k + 1$ unless there is also a cache miss on a cache of size k . Thus, algorithms that experience Belady's anomaly, like FIFO and clock [9], are *not* stack algorithms.

What kind of paging algorithms are stack? Roughly speaking, stack algorithms choose eviction victims based on a ranking that does not depend on k . This is formalized as follows. Let $\{\preceq_\sigma\}_{\sigma \in \mathcal{U}^*}$ be a family of total orders over \mathcal{U} . We say that a paging algorithm \mathcal{A} **conforms to** $\{\preceq_\sigma\}$ if, for every $\tau \in \mathcal{U}^*$, $z \in \mathcal{U}$ and k , if $x \in \mathcal{A}_k(\tau) \setminus \text{Out}(\mathcal{A}_k, \tau, z)$ and $y \in \text{Out}(\mathcal{A}_k, \tau, z)$, then $x \preceq_{\tau z} y$. When \mathcal{A} is lazy, this means that the only evicted item y is exactly the largest according to $\preceq_{\tau z}$, among the items in $\mathcal{A}_k(\tau)$.

The following theorem exhibits the close relationship between stack algorithms and order families. Although this connection is known [38], we are not aware of any previous work with a full proof, so we include it here.

Theorem 6. Let \mathcal{A} be a lazy paging algorithm. Then, \mathcal{A} is a stack algorithm if and only if there exists a family $\{\preceq_\sigma\}$ such that \mathcal{A} conforms to it.

To prove this theorem we will need the following technical lemma.

⁸A paging algorithm \mathcal{A} has **Belady's anomaly** [9] if, for some σ and integers $a > b$, $C(\mathcal{A}_a, \sigma) > C(\mathcal{A}_b, \sigma)$. That is, unexpectedly, the smaller cache performs better than the larger cache on some request sequence.

Lemma 7. *Let \mathcal{A} be a lazy paging algorithm. Then, for every τ , k and z , if*

$$\mathcal{A}_k(\tau) \subseteq \mathcal{A}_{k+1}(\tau) \text{ and } \text{Out}(\mathcal{A}_{k+1}, \tau, z) \cap \mathcal{A}_k(\tau z) \neq \emptyset,$$

then

$$\text{Out}(\mathcal{A}_k, \tau, z) \cap \mathcal{A}_{k+1}(\tau z) \neq \emptyset.$$

The proof of this lemma is deferred to the end of this section.

Proof of Theorem 6. (\Rightarrow) Let $\Sigma = \{\sigma_1, \dots, \sigma_{|\Sigma|}\}$. We will define the i -th smallest element with respect to \preceq_σ . Let $s = |\Sigma|$. If $i > s$, the answer is the $(i - s)$ -th smallest item in the set $\mathcal{U} \setminus \Sigma$ of unaccessed items. Otherwise, $i \leq s$. If $i = 1$, the answer is $\sigma_{|\Sigma|}$. Otherwise, $1 < i \leq s$. Notice that $|\mathcal{A}_j(\sigma)| = j$ for all $j \leq s$, as \mathcal{A} is a lazy algorithm. Combining this with the inclusion property of stack algorithms, we deduce that $|\mathcal{A}_i(\sigma) \setminus \mathcal{A}_{i-1}(\sigma)| = 1$. We define the i -th smallest element w.r.t. \preceq_σ as the only item in $\mathcal{A}_i(\sigma) \setminus \mathcal{A}_{i-1}(\sigma)$.

This order family satisfies that $\mathcal{A}_k(\sigma)$ is exactly the set of the $\min\{k, s\}$ smallest items w.r.t. \preceq_σ . This implies that \mathcal{A} conforms to $\{\preceq_\sigma\}$.

(\Leftarrow) We want to show that $\mathcal{A}_k(\sigma) \subseteq \mathcal{A}_{k+1}(\sigma)$ for every σ and k . We fix k and use induction on $|\sigma|$. The claim is trivial for $|\sigma| = 0$. Let $\sigma = \tau z$, and assume that the claim holds for τ , that is,

$$\mathcal{A}_k(\tau) \subseteq \mathcal{A}_{k+1}(\tau). \quad (7)$$

If z does not cause an eviction in \mathcal{A}_{k+1} , then Eq. (7) and the fact that \mathcal{A} is lazy imply that $\mathcal{A}_k(\tau z) \subseteq \mathcal{A}_{k+1}(\tau z)$, so we are done.

Otherwise, let $x \in \text{Out}(\mathcal{A}_{k+1}, \tau, z)$. Since \mathcal{A} is lazy, no other item was evicted by \mathcal{A}_{k+1} . Hence, by Eq. (7), it suffices to show that $x \notin \mathcal{A}_k(\tau z)$. By way of contradiction, assume $x \in \mathcal{A}_k(\tau z)$. Then,

$$x \in \text{Out}(\mathcal{A}_{k+1}, \tau, z) \cap \mathcal{A}_k(\tau z).$$

By Lemma 7, there exists a

$$y \in \text{Out}(\mathcal{A}_k, \tau, z) \cap \mathcal{A}_{k+1}(\tau z).$$

Thus, in response to z , \mathcal{A}_k evicted y and not x , and \mathcal{A}_{k+1} evicted x and not y . Because \mathcal{A} conforms to \preceq_σ , this means that $x \preceq_\sigma y$ and $y \preceq_\sigma x$, a contradiction. This completes the induction. \square

7.2 A Necessary Condition

Next, we establish the relationship between stable algorithms and stack algorithms.

Theorem 7. *Let \mathcal{A} be a lazy paging algorithm. If \mathcal{A} is stable, then \mathcal{A} is stack.*

Proof. We want to show that $\mathcal{A}_k(\sigma) \subseteq \mathcal{A}_{k+1}(\sigma)$ for every k and σ . The proof's structure is similar to that of Theorem 6. Once again, we fix k and use induction on $|\sigma|$. The case $|\sigma| = 0$ is trivial. Let $\sigma = \tau z$, and assume that the claim holds for τ , that is, $\mathcal{A}_k(\tau) \subseteq \mathcal{A}_{k+1}(\tau)$. Because \mathcal{A} is lazy, it suffices to show that if z causes \mathcal{A}_{k+1} to evict an item x , then $x \notin \mathcal{A}_k(\tau z)$.

Suppose that $x \in \text{Out}(\mathcal{A}_{k+1}, \tau, z)$ and, to reach a contradiction, that $x \in \mathcal{A}_k(\tau z)$. Then, by Lemma 7, there exists a $y \in \text{Out}(\mathcal{A}_k, \tau, z) \cap \mathcal{A}_{k+1}(\tau z)$.

Intuitively, this contradicts the fact that \mathcal{A} is stable, as the smaller instance \mathcal{A}_k should have evicted x , which is no longer in the larger instance \mathcal{A}_{k+1} , as opposed to y , which is still in \mathcal{A}_{k+1} . Formally, by the definition of stable algorithm for $X = \mathcal{U}$, $a = k + 1$ and $b = k$, we have that $y \in \text{Out}(\mathcal{A}_k, \tau, z) \cap \mathcal{A}_{k+1}(\tau) \neq \emptyset$ implies $\mathcal{A}_k(\tau z) \subseteq \mathcal{A}_{k+1}(\tau z)$, which gives a contradiction, because $x \in \mathcal{A}_k(\tau z) \setminus \mathcal{A}_{k+1}(\tau z)$. \square

This theorem provides a necessary condition for lazy stable algorithms, which we can use to rule out some well-known lazy paging algorithms that are not stack algorithms.

Corollary 2. *FIFO and clock are not stable.*

One may wonder whether stable and stack algorithms are, in fact, the same, at least when restricted to the class of lazy algorithms. We show that this is not the case.

Proposition 6. *There exists a lazy stack algorithm that is not stable.*

Proof. Let \mathcal{R} be the paging algorithm that always evicts the item x with the largest number of requests between the last two accesses to x ; this metric is known as the **reuse distance** [30]. Let $\Phi(\sigma, x)$ be the number of requests between the last 2 accesses to item x in σ . We define $\Phi(\sigma, x) = \infty$ if x has not been accessed at least twice. Consider the total order $x \preceq_\sigma y$ if and only if $\Phi(\sigma, x) < \Phi(\sigma, y)$ or $(\Phi(\sigma, x) = \Phi(\sigma, y) \text{ and } x \leq y)$. This defines an order family, and \mathcal{R} conforms to it, by definition. Thus, by Theorem 6, \mathcal{R} is a stack algorithm.

We claim that \mathcal{R} is not stable. Let $\mathcal{U} = \{A, B, C, Y, Z\}$ be the universe of items. We will compare \mathcal{R}_3 and \mathcal{R}_4 . Let $\sigma = AYZZZZABYYBC$. Let $X = \{A, B, C, Y\}$ be the set of items read by \mathcal{R}_3 . We will show that the definition of stable algorithm is violated in the last access of σ .

On the one hand, \mathcal{R}_3 reads the sequence $\sigma[X] = AYABYYBC$, and $Y \preceq_{\sigma[X]} A \preceq_{\sigma[X]} B$. Thus, during the final access to C , \mathcal{R}_3 evicts B . On the other hand, \mathcal{R}_4 reads the whole sequence σ , and $Y \preceq_\sigma Z \preceq_\sigma B \preceq_\sigma A$. Hence, when C is accessed, \mathcal{R}_4 evicts A . Because \mathcal{R}_3 evicted B (which is still in \mathcal{R}_4) and it still contains A (which is not in \mathcal{R}_4), \mathcal{R} is not stable. \square

7.3 A Sufficient Condition

Suppose some paging algorithm conforms to a family $\{\preceq_\sigma\}$. Notice that, in the extreme case, the orders can change abruptly from one step to the next, that is, \preceq_σ may be completely different from $\preceq_{\sigma x}$. This is because the orders are simply a rule the algorithm uses to decide which item to evict, and this rule is free to change at any point in time.

We will restrict our attention to order families that do not present this type of instability. We say that $\{\preceq_\sigma\}$ is **monotone** if for every σ , and $x, y, z \in \sigma$ with $y \neq z$,

$$\text{if } x \preceq_\sigma y, \text{ then } x \preceq_{\sigma z} y.$$

In words, a family is monotone when (1) an access can only change the relative order of the accessed item, and (2) the accessed item does not become larger in the order.

Lemma 8. *Let \mathcal{A} be a lazy paging algorithm that conforms to a monotone family $\{\preceq_\sigma\}$. Fix an arbitrary request sequence σ . Let $n = |\sigma|$, $\Sigma_n = \{\sigma_1, \dots, \sigma_n\}$, and $s = |\Sigma_n|$. Let x_1, \dots, x_s be the elements of Σ_n ordered by \preceq_σ . Then, for every $k \leq s$,*

$$\{x_1, \dots, x_{k-1}\} \subseteq \mathcal{A}_k(\sigma).$$

Lemma 8 says that the cache contents are essentially dictated by the order family, except perhaps for the largest item in the cache. Its proof can be found at the end of this section. The following corollary captures a crucial implication.

Corollary 3. *Let \mathcal{A} be a lazy paging algorithm that conforms to a monotone family $\{\preceq_\sigma\}$. Then, for every σ , k , and $y \in \sigma$, if $y \notin \mathcal{A}_k(\sigma)$, then there exists at most one $x \in \mathcal{A}_k(\sigma)$ such that $y \preceq_\sigma x$.*

Recall that stable algorithms have the property that the paging decisions of two instances serving request sequences σ and $\sigma[X]$, respectively, are “entangled”. We wish to capture this feature via order families. We say that $\{\preceq_\sigma\}$ is **self-similar** if for every σ , $X \subseteq \mathcal{U}$, and $x, y \in \sigma[X]$,

$$\text{if } x \preceq_{\sigma[X]} y, \text{ then } x \preceq_\sigma y.$$

We are now ready to state the main theorem of this section.

Theorem 8. *Let \mathcal{A} be a lazy paging algorithm. If \mathcal{A} conforms to a family $\{\preceq_\sigma\}$ that is monotone and self-similar, then \mathcal{A} is stable.*

Proof. Let $\tau \in \mathcal{U}^*$, $X \subseteq \mathcal{U}$, $z \in X$, and $a > b$. According to the definition of stable algorithm, we wish to show that $\text{Out}(\mathcal{A}_b, \tau[X], z) \cap \mathcal{A}_a(\tau z) \neq \emptyset$ implies $\mathcal{A}_b(\tau[X]z) \subseteq \mathcal{A}_a(\tau z)$. Assume the antecedent; let $x \in \text{Out}(\mathcal{A}_b, \tau[X], z) \cap \mathcal{A}_a(\tau z)$. Let $y \in \mathcal{A}_b(\tau[X]z)$, and let us show that $y \in \mathcal{A}_a(\tau z)$.

If $y = z$, there is nothing to show; assume $y \neq z$. Because $x \in \text{Out}(\mathcal{A}_b, \tau[X], z)$, we also have $x \neq z$.

Since $y \in \mathcal{A}_b(\tau[X]z)$, $y \neq z$ and \mathcal{A} is lazy, we have $y \in \mathcal{A}_b(\tau[X])$. Hence, in response to z , x was evicted by \mathcal{A}_b while y remained cached, so $y \preceq_{\tau[X]z} x$. Since $y \neq z$, the monotonicity property yields

$$y \preceq_{\tau[X]} x. \quad (8)$$

Then, the self-similarity property implies

$$y \preceq_\tau x. \quad (9)$$

Since $x \neq z$, we have, by monotonicity,

$$y \preceq_{\tau z} x. \quad (10)$$

We claim that $y \preceq_\tau z$. Since $x \in \text{Out}(\mathcal{A}_b, \tau[X], z)$ and \mathcal{A} is lazy, we have $z \notin \mathcal{A}_b(\tau[X])$. Notice that it cannot be $z \preceq_{\tau[X]} y$, because Eq. (8) would imply $z \preceq_{\tau[X]} x$, but $z \notin \mathcal{A}_b(\tau[X])$ cannot be smaller than two items in $\mathcal{A}_b(\tau[X])$ w.r.t. $\preceq_{\tau[X]}$, by Corollary 3. Hence, $y \preceq_{\tau[X]} z$. By the self-similarity property,

$$y \preceq_\tau z. \quad (11)$$

Recall that our goal is to show that $y \in \mathcal{A}_a(\tau z)$. For the sake of contradiction, assume $y \notin \mathcal{A}_a(\tau z)$. Since $y \neq z$ and \mathcal{A} is lazy, we have

$$y \notin \mathcal{A}_a(\tau). \quad (12)$$

Analogously, since $x \in \mathcal{A}_a(\tau z)$, $x \neq z$ and \mathcal{A} is lazy, we have

$$x \in \mathcal{A}_a(\tau). \quad (13)$$

To avoid contradicting Corollary 3, Eq. (9), Eq. (11), Eq. (12) and Eq. (13) imply that

$$z \notin \mathcal{A}_a(\tau), \quad (14)$$

and that

$$w \preceq_\tau y \text{ for every } w \in \mathcal{A}_a(\tau) \setminus \{x\}. \quad (15)$$

Monotonicity implies that

$$w \preceq_{\tau z} y \text{ for every } w \in \mathcal{A}_a(\tau) \setminus \{x\}. \quad (16)$$

Thus, by Eq. (10) and Eq. (16), x is the largest item in $\mathcal{A}_a(\tau)$ w.r.t. $\preceq_{\tau z}$. By Eq. (14), the request z was a miss in \mathcal{A}_a (and the cache was full, because \mathcal{A} is lazy and there have already been evictions in the past, namely that of y). Therefore, $x \in \text{Out}(\mathcal{A}_a, \tau, z)$, which is a contradiction. \square

7.4 Examples of Stable Algorithms

Lemma 5. *LRU-K is stable.*

Proof. We will use Theorem 8 on $\mathcal{A} = \text{LRU-K}$. Let $\Phi(\sigma, x)$ be the number of requests since the K -th most recent access to item x in σ . We define $\Phi(\sigma, x) = \infty$ if x has been accessed less than K times. Let $x \preceq_\sigma y$ if and only if $\Phi(\sigma, x) < \Phi(\sigma, y)$ or $(\Phi(\sigma, x) = \Phi(\sigma, y) \text{ and } x \leq y)$. This defines the order family that LRU-K conforms to. This family is monotone, because accessing an item does not increase its K -th most recent access time, and the relative order of all other items remains the same, as their K -th most recent access times increase by 1 after the new access. To conclude the proof we show that $\{\preceq_\sigma\}$ is also self-similar.

Suppose $x \preceq_{\sigma[X]} y$ with $x, y \in \sigma[X]$, and let us show that $x \preceq_\sigma y$. If $x = y$, the claim is trivial, so assume $x \neq y$. If $\Phi(\sigma[X], x) = \Phi(\sigma[X], y)$, then they must both be equal to ∞ , because the K -th most recent access times of two different items cannot match. Thus, both x and y have been accessed less than K times in $\sigma[X]$ and, consequently, in σ , so $\Phi(\sigma, x) = \Phi(\sigma, y) = \infty$. Hence, $x \preceq_\sigma y$.

Otherwise, $\Phi(\sigma[X], x) \neq \Phi(\sigma[X], y)$. Without loss of generality, assume $\Phi(\sigma[X], x) < \Phi(\sigma[X], y)$. Starting from $\sigma[X]$, we reinsert each of the accesses removed from σ , one by one. Crucially, none of these accesses are to x or y . Since the K -th most recent access to y is older than that of x , every time we insert an item access in the sequence either both become one unit time older, or only y does. Hence, at the end of the process, the K -th most recent access to y is still older than that of x , i.e., $\Phi(\sigma, x) < \Phi(\sigma, y)$. Thus, $x \preceq_\sigma y$, completing the proof. \square

Lemma 6. *LFU is stable.*

Proof. We only sketch the proof, since it is similar to the preceding one. Let $\Phi(\sigma, x)$ be the number of accesses to x in σ . Let $x \preceq_\sigma y$ if and only if $\Phi(\sigma, x) > \Phi(\sigma, y)$ or $(\Phi(\sigma, x) = \Phi(\sigma, y) \text{ and } x \leq y)$. LFU conforms to this order family, which can be shown to be monotone and self-similar. By Theorem 8, LFU is stable. \square

7.5 Missing Proofs

In the rest of this section we prove Lemma 7 and Lemma 8. We restate them here for convenience.

Lemma 7. *Let \mathcal{A} be a lazy paging algorithm. Then, for every τ , k and z , if*

$$\mathcal{A}_k(\tau) \subseteq \mathcal{A}_{k+1}(\tau) \text{ and } \text{Out}(\mathcal{A}_{k+1}, \tau, z) \cap \mathcal{A}_k(\tau z) \neq \emptyset,$$

then

$$\text{Out}(\mathcal{A}_k, \tau, z) \cap \mathcal{A}_{k+1}(\tau z) \neq \emptyset.$$

Proof. In order to streamline the argument, we will not explicitly invoke the laziness of \mathcal{A} every time we use it. Suppose that

$$\mathcal{A}_k(\tau) \subseteq \mathcal{A}_{k+1}(\tau), \tag{17}$$

and

$$\text{Out}(\mathcal{A}_{k+1}, \tau, z) \cap \mathcal{A}_k(\tau z) \neq \emptyset, \tag{18}$$

and let us show that $\text{Out}(\mathcal{A}_k, \tau, z) \cap \mathcal{A}_{k+1}(\tau z) \neq \emptyset$.

Let $x \in \text{Out}(\mathcal{A}_{k+1}, \tau, z) \cap \mathcal{A}_k(\tau z)$, as per Eq. (18). Since \mathcal{A}_{k+1} evicted x to fetch z , it had a full cache when z was accessed. Then, \mathcal{A}_k was also full at that time. Notice that z was a miss in \mathcal{A}_k ;

otherwise, if $z \in \mathcal{A}_k(\tau)$, then, by Eq. (17), $z \in \mathcal{A}_{k+1}(\tau)$, contradicting the fact that z caused an eviction in \mathcal{A}_{k+1} . Hence, \mathcal{A}_k was full when it missed z , and it must have evicted an item, say

$$y \in \text{Out}(\mathcal{A}_k, \tau, z). \quad (19)$$

To conclude the proof, we show that $y \in \mathcal{A}_{k+1}(\tau z)$. Eq. (19) implies that $y \in \mathcal{A}_k(\tau)$. Then, by Eq. (17),

$$y \in \mathcal{A}_{k+1}(\tau). \quad (20)$$

Notice that $y \neq x$, since $x \in \mathcal{A}_k(\tau z)$ and $y \notin \mathcal{A}_k(\tau z)$. In particular, y was not evicted by \mathcal{A}_{k+1} . Combining this with Eq. (20), we conclude that $y \in \mathcal{A}_{k+1}(\tau z)$, as desired. \square

Lemma 8. *Let \mathcal{A} be a lazy paging algorithm that conforms to a monotone family $\{\preceq_\sigma\}$. Fix an arbitrary request sequence σ . Let $n = |\sigma|$, $\Sigma_n = \{\sigma_1, \dots, \sigma_n\}$, and $s = |\Sigma_n|$. Let x_1, \dots, x_s be the elements of Σ_n ordered by \preceq_σ . Then, for every $k \leq s$,*

$$\{x_1, \dots, x_{k-1}\} \subseteq \mathcal{A}_k(\sigma).$$

Proof. Once again, we will use the fact that \mathcal{A} is lazy without being explicit about it. The proof is by induction on n . The case $n = 0$ is trivial. Let $n \geq 1$. Let $\tau = \sigma_1 \dots \sigma_{n-1}$. Let $y_1, \dots, y_{|\Sigma_{n-1}|}$ be the items of Σ_{n-1} ordered by \preceq_τ .

Fix $k \leq s$. By the inductive hypothesis,

$$\{y_1, \dots, y_{k-1}\} \subseteq \mathcal{A}_k(\tau). \quad (21)$$

Since the order family is monotone,

$$\{x_1, \dots, x_{k-1}\} \subseteq \{y_1, \dots, y_{k-1}\} \cup \{\sigma_n\}. \quad (22)$$

(Intuitively, when σ_n is accessed, it is taken from its current position in the ordered list of items y_1, y_2, \dots and pulled forward; if σ_n was not accessed before, it is inserted somewhere in the list.)

If $\sigma_n \in \mathcal{A}_k(\tau)$, then Eq. (21) implies that

$$\{y_1, \dots, y_{k-1}\} \cup \{\sigma_n\} \subseteq \mathcal{A}_k(\sigma).$$

Thus, by Eq. (22), $\{x_1, \dots, x_{k-1}\} \subseteq \mathcal{A}_k(\sigma)$.

Otherwise, $\sigma_n \notin \mathcal{A}_k(\tau)$. By Eq. (21), we have $\mathcal{A}_k(\tau) = \{y_1, \dots, y_{k-1}, z\}$ for some z that is larger than every other item in $\mathcal{A}_k(\tau)$ w.r.t. \preceq_τ . By monotonicity, z is also the largest in $\mathcal{A}_k(\tau)$ w.r.t. \preceq_σ . Thus, when σ_n is accessed, z is evicted, and

$$\mathcal{A}_k(\sigma) = \{y_1, \dots, y_{k-1}, \sigma_n\}.$$

By Eq. (22), we conclude that $\{x_1, \dots, x_{k-1}\} \subseteq \mathcal{A}_k(\sigma)$, as desired. This completes the induction. \square

8 Conclusion

What is the smallest associativity α a cache designer may use such that hit rate of set-associative cache is similar to that of a fully associative cache? The answer to this decades-old question is to set α slightly above logarithmic in the cache size. This performance guarantee, however, only lasts for a bounded amount of time. If the cache designer further wishes the set-associative cache to sustain comparable performance on arbitrarily long executions, periodically rehashing after a polynomial number of cache misses suffices.

Acknowledgments

We thank the anonymous reviewers of SPAA '23 for their feedback that helped us improve the presentation of the paper. We gratefully acknowledge support from NSF grants CCF-2118830, CCF-2106827, CSR-1763680, CNS-1938709, CNS-2118620 and CCF-2106999. This research was also supported in part by NSERC.

References

- [1] Dolev Adas, Gil Einziger, and Roy Friedman. Limited associativity makes concurrent software caches a breeze. In *Proceedings of the 23rd International Conference on Distributed Computing and Networking (ICDCN)*, pages 87–96, New York, NY, USA, 2022. Association for Computing Machinery.
- [2] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst.*, 6(4):393–431, November 1988.
- [3] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, sep 1988.
- [4] Kunal Agrawal, Michael A. Bender, Rathish Das, William Kuszmaul, Enoch Peserico, and Michele Scquizzato. Green paging and parallel paging. In *Proc. 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 493–495, July 2020.
- [5] Kunal Agrawal, Michael A. Bender, Rathish Das, William Kuszmaul, Enoch Peserico, and Michele Scquizzato. Tight bounds for parallel paging and green paging. In *Proc. 32th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3022–3041, January 2021.
- [6] Kunal Agrawal, Michael A Bender, Rathish Das, William Kuszmaul, Enoch Peserico, and Michele Scquizzato. Online parallel paging with optimal makespan. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 205–216, 2022.
- [7] Kunal Agrawal, Michael A. Bender, and Jeremy T. Fineman. The worst page-replacement policy. *Theory of Computing Systems*, 44:175–185, 2 2009.
- [8] Kathirgamar Aingaran, Sumti Jairath, Georgios Konstantinidis, Serena Leung, Paul Loewenstein, Curtis McAllister, Stephen Phillips, Zoran Radovic, Ram Sivaramakrishnan, David Smentek, and Thomas Wicki. M7: Oracle’s next-generation sparcs processor. *IEEE Micro*, 35(2):36–45, 2015.
- [9] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Commun. ACM*, 12(6):349–353, jun 1969.
- [10] James Bell, David Casasent, and C. Gordon Bell. An investigation of alternative cache organizations. *IEEE Transactions on Computers*, C-23(4):346–351, 1974.
- [11] Michael A. Bender, Abhishek Bhattacharjee, Alex Conway, Martín Farach-Colton, Rob Johnson, Sudarsun Kannan, William Kuszmaul, Nirjhar Mukherjee, Don Porter, Guido Tagliavini, Janet Vorobyeva, and Evan West. Paging and the address-translation problem. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 105–117, New York, NY, USA, 2021. Association for Computing Machinery.

- [12] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, USA, 1998.
- [13] Bill Bowhill, Blaine Stackhouse, Nevine Nassif, Zibing Yang, Arvind Raghavan, Charles Morganti, Chris Houghton, Dan Krueger, Olivier Franza, Jayen Desai, Jason Crop, Dave Bradley, Chris Bostak, Sal Bhimji, and Matt Becker. The xeon® processor e5-2600 v3: A 22nm 18-core product family. In *2015 IEEE International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 1–3. IEEE Computer Society, 2015.
- [14] Joan Boyar, Martin R. Ehmsen, Jens S. Kohrt, and Kim S. Larsen. A theoretical comparison of lru and lru-k. *Acta Informatica*, 47:359–374, 2010.
- [15] Joan Boyar, Lene M. Favrholdt, and Kim S. Larsen. The relative worst-order ratio applied to paging. *J. Comput. Syst. Sci.*, 73(5):818–843, August 2007.
- [16] Mark Brehob, Richard Enbody, Eric Torng, and Stephen Wagner. On-line restricted caching. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 374–383, USA, 2001. Society for Industrial and Applied Mathematics.
- [17] Niv Buchbinder, Shahar Chen, and Joseph (Seffi) Naor. Competitive algorithms for restricted caching and matroid caching. In Andreas S. Schulz and Dorothea Wagner, editors, *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA)*, pages 209–221, Berlin, Heidelberg, September 2014. Springer-Verlag.
- [18] Edward G. Coffman and Peter J. Denning. *Operating Systems Theory*. Prentice Hall Professional Technical Reference, 1973.
- [19] Rathish Das, Kunal Agrawal, Michael A Bender, Jonathan Berry, Benjamin Moseley, and Cynthia A Phillips. How to manage high-bandwidth memory automatically. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 187–199, 2020.
- [20] Daniel DeLayo, Kenny Zhang, Kunal Agrawal, Michael A Bender, Jonathan Berry, Rathish Das, Benjamin Moseley, and Cynthia A Phillips. Automatic hbm management: Models and algorithms. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2022.
- [21] Reza Dorrigiv and Alejandro López-Ortiz. Closing the gap between theory and practice: New measures for on-line algorithm analysis. In *International Workshop on Algorithms and Computation (WALCOM)*, pages 13–24. Springer-Verlag Berlin Heidelberg, 2008.
- [22] Reza Dorrigiv, Alejandro López-Ortiz, and J. Ian Munro. On the relative dominance of paging algorithms. *Theor. Comput. Sci.*, 410(38–40):3694–3701, September 2009.
- [23] Devdatt Dubhashi and Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, USA, 1st edition, 2009.
- [24] Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel D. Sleator, and Neal E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, December 1991.
- [25] James D. Fix. The set-associative cache performance of search trees. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 565–572, USA, 2003. Society for Industrial and Applied Mathematics.

- [26] M. Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 1999 IEEE 40th Annual Symposium on Foundations of Computer Science (FOCS)*, page 285, Los Alamitos, CA, USA, October 1999. IEEE Computer Society.
- [27] John S. Harper, Darren J. Kerbyson, and Graham R. Nudd. Analytical modeling of set-associative cache behavior. *IEEE Transactions on Computers*, 48(10):1009–1024, October 1999.
- [28] Mark D. Hill. A case for direct-mapped caches. *Computer*, 21(12):25–40, December 1988.
- [29] Mark D. Hill and Alan J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.
- [30] Song Jiang and Xiaodong Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, pages 31–42, New York, NY, USA, 2002. Association for Computing Machinery.
- [31] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Computer Architecture News*, 18, 1990.
- [32] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. Competitive snoop caching. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 244–254, 1986.
- [33] Richard E. Kessler, R. Jooss, Alvin R. Lebeck, and Mark D. Hill. Inexpensive implementations of set-associativity. In *The 16th Annual International Symposium on Computer Architecture (ISCA)*, pages 131–139. IEEE Computer Society, 1989.
- [34] William F. King. Analysis of demand paging algorithms. In *IFIP Congress*, 1971.
- [35] Georgios K. Konstantinidis, Hongping Penny Li, Francis Schumacher, Venkat Krishnaswamy, Hoyeol Cho, Sudesna Dash, Robert P. Masleid, Chaoyang Zheng, Yuanjung David Lin, Paul Loewenstein, Heechoul Park, Vijay Srinivasan, Dawei Huang, Changku Hwang, Wenjay Hsu, Curtis McAllister, Jeff Brooks, Ha Pham, Sebastian Turullols, Yifan Yanggong, Robert Golla, Alan P. Smith, and Ali Vahidsafa. Sparc m7: A 20 nm 32-core 64 mb l3 cache processor. *IEEE Journal of Solid-State Circuits*, 51(1):79–91, 2016.
- [36] Nasser Kurd, Muntaquim Chowdhury, Edward Burton, Thomas P. Thomas, Christopher Mozak, Brent Boswell, Manoj Lal, Anant Deval, Jonathan Douglas, Mahmoud Ellassal, Ankireddy Nalamalpu, Timothy M. Wilson, Matthew Merten, Srinivas Chennupaty, Wilfred Gomes, and Rajesh Kumar. Haswell: A family of ia 22nm processors. In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 112–113. IEEE Computer Society, 2014.
- [37] Mark S. Manasse, Lyle A. McGeoch, and Daniel D. Sleator. Competitive algorithms for server problems. *J. Algorithms*, 11(2):208–230, may 1990.
- [38] Richard L. Mattson, Jan Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, June 1970.

- [39] M. Mendel and Steven S. Seiden. Online companion caching. *Theoretical Computer Science*, 324(2–3):183–200, September 2004.
- [40] Nima Mousavi. How tight is chernoff bound? Notes, 2012.
- [41] Elizabeth J. O’Neil, Patrick E. O’Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 297–306, New York, NY, USA, 1993. Association for Computing Machinery.
- [42] Elizabeth J. O’Neil, Patrick E. O’Neil, and Gerhard Weikum. An optimality proof of the lru-k page replacement algorithm. *Journal of the ACM*, 46(1):92–112, January 1999.
- [43] Enoch Peserico. Online paging with arbitrary associativity. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 555–564, USA, 2003. Society for Industrial and Applied Mathematics.
- [44] S. Prybylski, M. Horowitz, and J. Hennessy. Performance tradeoffs in cache design. In *Proceedings of the 15th Annual International Symposium on Computer Architecture (ISCA)*, pages 290–298, Washington, DC, USA, 1988. IEEE Computer Society.
- [45] Moinuddin K. Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 775–787. IEEE Press, 2018.
- [46] Moinuddin K. Qureshi. New attacks and defense for encrypted-address cache. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, pages 360–371, New York, NY, USA, 2019. Association for Computing Machinery.
- [47] Gururaj S. Rao. Performance analysis of cache memories. *Journal of the ACM*, 25(3):378–395, July 1978.
- [48] RocksDB. Block cache. RocksDB wiki, 2022. Last accessed: 2023-01-09.
- [49] Efraim Rotem, Adi Yoaz, Lihu Rappoport, Stephen J. Robinson, Julius Yuli Mandelblat, Arik Gihon, Eliezer Weissmann, Rajshree Chabukswar, Vadim Basin, Russell Fenger, Monica Gupta, and Ahmad Yasin. Intel alder lake cpu architectures. *IEEE Micro*, 42(3):13–19, 2022.
- [50] Peter Sanders. Accessing multiple sequences through set associative caches. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 655–664, Berlin, Heidelberg, 1999. Springer-Verlag.
- [51] Rathijit Sen and David A. Wood. Reuse-based online models for caches. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 279–292, New York, NY, USA, 2013. Association for Computing Machinery.
- [52] Sandeep Sen, Siddhartha Chatterjee, and Neeraj Dumir. Towards a theory of cache-efficient algorithms. *Journal of the ACM*, 49(6):828–858, November 2002.
- [53] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, February 1985.

- [54] Alan J. Smith. On the effectiveness of set associative page mapping and its application to main memory management. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE)*, pages 286–292, Washington, DC, USA, 1976. IEEE Computer Society Press.
- [55] Alan J. Smith. A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Transactions on Software Engineering*, 4(2):121–130, March 1978.
- [56] Simon M. Tam, Harry Muljono, Min Huang, Sitaraman Iyer, Kalapi Royneogi, Nagmohan Satti, Rizwan Qureshi, Wei Chen, Tom Wang, Hubert Hsieh, Sujal Vora, and Eddie Wang. Skylake-sp: A 14nm 28-core xeon® processor. In *2018 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 34–36. IEEE Computer Society, 2018.
- [57] Nigel Topham and Antonio González. Randomized cache placement for eliminating conflicts. *IEEE Transactions on Computers*, 48(2):185–192, February 1999.
- [58] David Wajc. Negative association - definition, properties, and applications, 2017.
- [59] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. Hotl: A higher order theory of locality. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 343–356, New York, NY, USA, 2013. Association for Computing Machinery.
- [60] Neal Young. *Competitive Paging and Dual-Guided on-Line Weighted Caching and Watching Algorithms*. PhD thesis, Princeton University, USA, 1992.
- [61] Neil Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, 1994.
- [62] Neil Young. On-line file caching. *Algorithmica*, 33(3):371–383, 2002.