Increment–and–Freeze: Every Cache, Everywhere, All of the Time

Michael A. Bender
Stony Brook University
NY, USA
bender@cs.stonybrook.edu

Daniel DeLayo
Stony Brook University
NY, USA
ddelayo@cs.stonybrook.edu

Bradley C. Kuszmaul
CA, USA
kuszmaul@gmail.com

William Kuszmaul
Massachusetts Institute of Technology
MA, USA
william.kuszmaul@gmail.com

Evan West
Stony Brook University
NY, USA
etwest@cs.stonybrook.edu

ABSTRACT

One of the most basic algorithmic problems concerning caches is to compute the LRU hit-rate curve on a given trace. Unfortunately, the known algorithms exhibit poor data locality and fail to scale to large caches. It is widely believed that the LRU hit-rate curve cannot be computed efficiently enough to be used in online production settings. This has led to a large literature on heuristics that aim to approximate the curve efficiently.

In this paper, we show that the poor data locality of past algorithms can be avoided. We introduce a new algorithm, called Increment–and–Freeze, for computing exact LRU hit-rate curves. The algorithm achieves RAM-model complexity $O(n \log n)$, external-memory complexity $O(\frac{n}{B} \log n)$, and parallelism $\Theta(\log n)$. We also present two theoretical extensions of Increment–and–Freeze, one that achieves SORT complexity in the external-memory model, and one that achieves a parallel span of $O(\log^2 n)$ which is near linear parallelism, while maintaining work efficiency.

We implement Increment–and–Freeze [5] and obtain a speedup of up to 9× over the classical augmented-tree algorithm on a single processor. On 16 threads, the speedup becomes as large as 60×. In comparison to the previous state-of-the-art parallel algorithm, Increment–and–Freeze achieves a speedup of up to 10× when both algorithms use the same number of threads.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; **Caching and paging algorithms**; **Parallel algorithms**; *Parameterized complexity and exact algorithms*; • **General and reference** → *Performance*.

KEYWORDS

LRU, Hit-rate Curves, Miss-ratio Curves, Success Function, Reuse Distance, Stack Distance, Working Set, Divide-And-Conquer, Caching, External-Memory, IO-Optimal, Parallelism

ACM Reference Format:
Michael A. Bender, Daniel DeLayo, Bradley C. Kuszmaul, William Kuszmaul, and Evan West. 2023. Increment–and–Freeze: Every Cache, Everywhere, All of the Time. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '23), June 17–19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3558481.3591085

1 INTRODUCTION

The basic theory of caches is well understood. Although the optimal cache-eviction strategy OPT [3, 20] cannot be implemented by an online algorithm, the strategy of always evicting the least-recently-used item (LRU) [20] is known to be provably $(1 + \varepsilon)$-competitive with $O(1/\varepsilon)$-factor resource augmentation [24]. Most modern caches implement variations of LRU; these variations include simplifications that reduce overhead and workload-specific optimizations. In recent decades we have seen the emergence of *giant caches*, especially in the context of Content Distribution Networks (CDNs). These caches can span 100s of machines and cost millions of dollars a year to run.

Since these caches are so large, altering the cache size or eviction strategy in response to a changing workload can result in significant cost savings and performance benefits. For engineers in charge of running such a cache, there are several core questions that they want to keep track of: Could they shrink the cache (therefore saving resources) while achieving a similar hit rate? Could they grow the cache by a small amount and achieve a significantly smaller miss rate? Are the ways in which the cache approximates LRU hurting its performance in comparison to a true LRU cache? And to what degree are the optimizations that the cache makes beyond LRU leading to better performance? All of these questions are made more difficult by the fact that the answers change over time. The optimizations that led to better performance on a previous data set may have little or no benefit on the current one. And the cache size that made sense earlier may be suboptimal now.

A natural strategy for answering these "what-if" questions is to compute the ***LRU hit-rate curve*** for the cache over time. That is, for each time period (e.g., each day) and for each possible cache size k, what is the hit rate that would have been achieved by an LRU cache of size k in that time period? However, the task of actually *computing* the LRU hit-rate curve can itself be daunting. Even on relatively small caches, the time to compute the hit-rate curve often ends up exceeding the *execution time* of the trace under analysis by multiple orders of magnitude [21].

Whereas cache sizes have scaled tremendously in past decades, our algorithms for computing LRU hit-rate curves have not. The classical augmented-tree algorithm, due to Bennett and Kruskal [6] in 1975, uses an augmented binary search tree in order to compute the hit-rate curve on a trace of size n in time $O(n \log n)$. Although the algorithm's running time is near-optimal in the RAM model [31], it struggles to scale to large applications, both because it is difficult to parallelize without incurring a significant memory blowup [21] and because it exhibits poor data locality, incurring $\Theta(n \log n)$ misses to CPU cache. This augmented-tree approach has remained the only $O(n \log n)$ algorithm in the literature for more than four decades.

This lack of locality poses a significant bottleneck. By definition, the program under analysis incurs at most one cache miss for each memory access in the trace, and typically less than one miss per hundred accesses [2]. As a result, any hit-rate-curve algorithm incurring $O(\log n)$ cache misses per access experiences far more misses than the trace it is processing. Moreover, the footprint of a trace produced by large distributed cache may massively exceed the cache of the machine analyzing it. The result is that hit-rate computations are easily bottlenecked by cache-misses and IOs.[1]

The deficits of the augmented-tree algorithm have led to the widespread belief that exact LRU hit-rate curves cannot be computed efficiently enough for online use in production systems [26, 29]. Thus, there has been a great deal of work on heuristics for efficiently *approximating* the LRU hit-rate curve [15–17, 23, 26, 29, 31]. These heuristics are observed to achieve high experimental accuracy, but do not offer provable correctness guarantees.

In this paper, we revisit the claim that exact LRU hit-rate curves are necessarily impractical. We introduce a new algorithm that is significantly faster than the classic augmented-tree approach (both in serial and in parallel), and we show that this algorithm naturally generalizes to achieve SORT time in the external-memory model and near-optimal parallelism in the CREW PRAM model.

This Paper: The Increment-and-Freeze Algorithm. In this paper, we present a new $O(n \log n)$-time algorithm which we call Increment–and–Freeze. The most basic version of this algorithm achieves parallelism $\Theta(\log n)$ and incurs a cost of $O(\frac{n}{B} \log n)$ in the external-memory model. Improving the data-locality is especially important as it results in significant speedups over the classic augmented-tree algorithm even when using a single thread.

On the theoretical side, we also present extensions of the Increment–and–Freeze algorithm that achieve even stronger bounds on IO efficiency and parallelism. In the external memory model, we achieve a bound of $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ IOs. In the CREW PRAM model, we achieve $O(n \log n)$ work with $O(\log^2 n)$ span. This implies a parallelism of $\Theta(n/\log n)$, which is optimal up to polylogarithmic factors.

We implement and evaluate Increment–and–Freeze as well as a variation that is sensitive to limits on the maximum cache size. This limit can either be a user-specified parameter or the natural bound of the number of unique request-ids. With a cache size limit of k, our implementation performs work $O(n \log k)$, achieves $\Theta(\log k)$-fold parallelism, and has an external-memory bound of $O(\frac{n}{B} \log k)$. We remark that our implementation produces the hit-rate curve not just at the end of the trace T, but also at regular intervals of size $O(k)$.

Increment–and–Freeze achieves a speedup of 4×–9× over the classical augmented-tree algorithm on a single processor. On 16 threads, the speedup becomes as large as 60×. In comparison to the previous state-of-the-art parallel algorithm, Increment–and–Freeze achieves a speedup of 2×–10× when both algorithms use the same number of threads. For all experiments, the biggest speedups occurred on the largest traces. A trace that took 13 hours for the classical augmented-tree algorithm was processed in only 13 minutes by Increment–and–Freeze (Section 9).

2 RELATED WORK

In 1970, Mattson et al. [20] gave an $O(n^2)$ algorithm for computing the LRU hit-rate curve. They showed that each access x has an ***LRU stack distance*** d that determines whether the access is a cache hit or miss: an access with LRU stack distance d is an LRU cache hit iff (1) the cache has size d or larger, and (2) the address has been accessed at least once before. They gave a simple characterization of LRU stack distance: the number of distinct items accessed between the current time t and previous time t_0 that x was accessed (where $t_0 = 0$ if x was never accessed before). The $O(n^2)$ algorithm follows by storing the distinct addresses in a stack, sorted by their most recent access time. Whenever an address is accessed, it is moved to the front of the stack, and its LRU stack distance is given by the position that it previously occupied. The algorithm can be parameterized by the average LRU stack distance s, taking time $O(ns)$—variations of this algorithm were studied by Kim et al. [18].

In 1975, Bennett and Kruskal [6] reduced the time to $O(n \log n)$ by storing Mattson et al.'s stack as an augmented search tree. This approach has since seen many variations [2, 22, 25] (see, also, Byrne's survey [13]). Several modern implementations [21, 25] have converged to using a splay tree whose leaves are the members of the stack and whose internal nodes store additional information so that the average time per operation is $O(\log n)$. These algorithms can also be implemented to run in time $O(n \log k)$ where k is the maximum cache size that we wish to consider.

Unfortunately, even the $O(n \log n)$ time achieved by the augmented-tree algorithm is typically viewed as too heavy-weight for practical online use in production systems [26, 29]. Thus, researchers have resorted to *approximation algorithms*, which aim to produce accurate estimates for the LRU hit-rate curve. There has been a great deal of work on *heuristic-based* algorithms [15–17, 23, 26, 29], which have been observed to achieve high accuracy on experimental inputs but which do not offer provable guarantees. On the theoretical side, there has also been work by Drudi et al. [14] on space-efficient single-pass streaming algorithms for approximating the LRU hit-rate curve at p uniformly spaced points—they offer a nearly optimal bound of $O(p^2 \varepsilon^{-2} \operatorname{polylog} n)$ bits on the space needed to achieve additive error ε, and they prove that achieving a small multiplicative error requires linear space. There has also been work on implementing approximate versions

[1]As a convention, we will often use the term 'IO' instead of 'cache miss' when referring to the cache misses incurred by our algorithms. This is both to disambiguate between the cache misses incurred by our algorithm versus those incurred by the trace and because the term IO is standard when performing analyses in the external-memory model [1].

of the augmented-tree algorithm that use sketching techniques and bloom filters in order to improve performance [28].

Zhong et al. [31] proposed that, rather than approximating the hit-rate curve itself, one could instead approximate the LRU stack distances for each access. They gave an elegant algorithm that, for any constant $\varepsilon > 0$, can be used to obtain a $(1+\varepsilon)$-approximation for every LRU stack distance in time $O(n \log \log n)$. One downside of this approach is that it does not translate into an accuracy guarantee for the hit-rate curve that is produced.

Finally, Niu et al. [21] tackle the problem of parallelizing the exact augmented-tree algorithm. The problem is that dependencies across time translate into serial dependencies for the algorithm. Nonetheless, Niu et al. [21] show that, if one is willing to incur a significant memory blowup, then parallelism is still possible: they break the trace into large chunks, they process the chunks in parallel, and then they perform cleanup work in order to handle the dependencies between chunks. To ensure that the cleanup work doesn't dominate, each chunk needs to have size larger than the the number u of distinct addresses in the trace. One consequence is that, in order to achieve a p-fold parallel speedup, one must use memory $\Omega(up)$. This approach, known as PARDA [21], works well for sizes on the order of a CPU cache [21], but becomes limited on large caches by its memory behavior. The blowup in memory usage restricts PARDA's parallelism in many of our experiments, and the poor data locality of the underlying augmented-tree algorithm further limits performance. At moderately large cache sizes, PARDA is slower on 16 threads than Increment–and–Freeze is on a single thread.

Although we have focused here on the LRU hit-rate curve, there has also been a great deal of work on computing the hit-rate curve for the offline optimal cache-eviction strategy OPT. Surprisingly, although the optimal eviction strategy cannot be determined online, its miss rate can: in 1966, Bélády [3] described an online algorithm known as MIN that calculates the number of cache hits for an optimal cache of a given size for a given trace. In 1970, Mattson et al. [20] showed that the offline Furthest-in-The-Future algorithm is offline optimal, and then in 1974, Bélády and Palermo [4] proved the correctness of MIN (that is, that MIN and Furthest-in-The-Future produce the same results).

Standard augmented-tree approaches can be used to construct offline OPT for a given cache size k on a given trace of size n in time $O(n \log k)$. This approach does not generalize to producing the entire hit-rate curve in time $O(n \log n)$. For several decades it remained open to compute the hit-rate curve for caches of sizes in the range $1, \ldots, k$ in any time better than $O(nk \log k)$. The first breakthrough on this occurred in 2011 when Bilardi et al. [8] gave an $O(n\sqrt{k} \log k)$-time algorithm. In subsequent work, the same set of authors [9] were able to achieve a bound of $O(n \log k)$. The practical performance of these algorithms, sampling approaches to OPT, and their applications to heterogeneous memory hierarchies have been explored [7, 27, 30], and remain interesting directions of work.

3 PRELIMINARIES

A ***trace*** $T = \langle t_1, t_2, \ldots, t_n \rangle$ is a sequence of memory accesses, where $t_i \in [u]$ denotes the address referenced by the i-th memory access.

Computing the hit-rate curve of T is equivalent to, for each $t_i \in T$, finding the stack distance of t_i. The stack distance of an access t_i is equal to the number of unique addresses accessed between t_i and $t_j = t_i$ the previous time this address appeared in T.

For each access t_i, define $\text{prev}(i)$ to be the largest $j < i$ such that $t_j = t_i$. If and only if there is no such j, then let $\text{prev}(i)$ be 0. Similarly, define $\text{next}(i)$ to be the smallest $j > i$ such that $t_j = t_i$, or to be ∞ if no such j exists. Define the ***distance vector*** $\langle d_1, d_2, \ldots, d_n \rangle$ so that d_i is the number of distinct addresses in the sequence $t_i, t_{i+1}, \ldots, t_{\text{next}(i)-1}$. On an LRU cache of size k, the access t_i will be a ***cache hit*** if and only if $\text{prev}(i) \neq 0$ and $d_{\text{prev}(i)} \leq k$. The ***LRU hit-rate curve*** $H_T : \{1, 2, \ldots, n\} \to [0, 1]$ is, for each cache size k, what fraction $H_T(k)$ of accesses are hits. That is,

$$H_T(k) = \frac{|\{i \mid \text{prev}(i) \neq 0 \text{ and } d_{\text{prev}(i)} \leq k\}|}{n}. \quad (1)$$

The task of computing H_T naturally decomposes into three phases: the ***pre-processing phase***, during which one computes $\text{prev}(i)$ and $\text{next}(i)$ for each $i \in [n]$; the ***distance computation phase***, during which one computes the distance vector $\langle d_1, d_2, \ldots, d_n \rangle$; and the ***post-processing phase*** during which one constructs H_T using (1).

We will analyze our algorithms in three standard models: the RAM model [19], the external-memory model [1], and the CREW PRAM model [12]. The ***external-memory model*** measures cost in terms of IOs (such as block transfers or cache misses) and is governed by two additional parameters M and B, where M is the size of internal memory and B is the external-memory block size (such as the cache-line transfer size, or the size of a disk block). The ***CREW PRAM model*** measures a parallel algorithm by its ***work*** (i.e., how long it would take to run in serial) and by its ***span*** (i.e., how long it would take to run in parallel on infinitely many processors). The ***parallelism*** of a parallel algorithm is then defined to be the work divided by the span. All of our CREW PRAM algorithms can be implemented as fork-join parallel programs [19].

Both the pre-processing and post-processing phases reduce straightforwardly to a constant number of sort and prefix-sum operations. Since it is well understood how to implement both sorting and prefix sums efficiently in all three models (and even simultaneously [11]), we will not concern ourselves with this here. Instead we will focus our technical discussion on the distance computation phase.

4 INCREMENT–AND–FREEZE ALGORITHM

This section introduces the ***Increment–and–Freeze algorithm***, which is a new $O(n \log n)$-time algorithm for computing the distance vector $\langle d_1, d_2, \ldots, d_n \rangle$ of a trace T. The algorithm has cost $O((n/B) \log n)$ in the external-memory model and has parallelism $\Theta(\log n)$ in the CREW PRAM model. We will also see in later sections how to improve these bounds further with additional algorithmic ideas.

Let $A[1, 2, \ldots, n]$ be an array initialized with zeros. The algorithm will perform a series of "Increment" and "Freeze" operations on the array. A ***Freeze***(i) operation freezes the value of array element $A[i]$, preventing it from ever being modified again (if $i = 0$, then the Freeze operation does nothing). An ***Increment***(i, j, r) operation increments the array elements $A[i], \ldots, A[j]$ each by r—but any array element that has been frozen is not affected by the increment (if $i > j$, then the Increment operation does nothing).

Consider the operation sequence,

$$\mathcal{S} = \langle \mathbf{I}_1, \mathbf{F}_1, \mathbf{I}_2, \mathbf{F}_2, \ldots, \mathbf{I}_n, \mathbf{F}_n \rangle,$$

where $\mathbf{I}_i = \text{Increment}(\text{prev}(i), i-1, 1)$ and $\mathbf{F}_i = \text{Freeze}(\text{prev}(i))$. Note that all of the Increment operations $\mathbf{I}_i$ increment by only 1—the ability to increment by $r > 1$ will be useful for merging neighboring Increment operations. Each I_i increments all unfrozen address from its previous occurrence up to, but not including, itself. After performing the increment, we freeze i's previous occurrence.

We now prove that the operation sequence $\mathcal{S}$ computes the distance vector $\langle d_1, \ldots, d_n \rangle$.

Lemma 4.1. *If the operation sequence $\mathcal{S}$ is performed on array A, then the result is $A[i] = d_i$ for each $i \in [n]$.*

Proof. The array element $A[i]$ is Frozen by $F_{\text{next}(i)}$ (unless $\text{next}(i) = \infty$). Therefore the increment operations I_j for $j > \text{next}(i)$ are not applied to $A[i]$. The increment operations I_j that apply to $A[i]$ are therefore the ones for which

$$\text{prev}(j) \le i < j \le \text{next}(i).$$

That is, $A[i]$ is incremented once for each trace element t_j that appears in $t_{i+1}, \ldots, t_{\text{next}(i)}$ and such that $\text{prev}(j) \le i$. The total number of times that $A[i]$ is incremented by the operation sequence $\mathcal{S}$ is therefore equal to d_i, the number of distinct addresses in $t_{i+1}, \ldots, t_{\text{next}(i)}$. □

We perform the operation sequence $\mathcal{S}$ by recursively "projecting" the sequence onto smaller and smaller sub-arrays of A. For an operation $I = \text{Increment}(i, j, r)$, define ***proj***$(I)$ of I onto an interval $[a, b]$ to be the operation I' in which the interval $[i, j]$ is shrunk into $[a, b]$, that is, $I' = \text{Increment}(\max(i, a), \min(j, b), r)$. For an operation $F = \text{Freeze}(i)$, define ***proj***(F) of the operation onto an interval $[a, b]$ to be $\text{Freeze}(i)$ if $i \in [a, b]$ and $\text{Freeze}(0)$ otherwise. When an operation does not fall within the range $[a, b]$ then its projection will do nothing. We refer to these do-nothing operations as null operations—that is, $\text{Increment}(i, j, r)$ is null if $i > j$ and $\text{Freeze}(i)$ is null if $i = 0$. Define the ***projection*** of an operation sequence $\mathcal{X} = \langle x_1, \ldots, x_m \rangle$ onto an interval $[a, b]$ to be the sequence $\langle x'_1, \ldots, x'_m \rangle$ where x'_i is the projection of x_i onto $[a, b]$.

Define the ***shrunk projection*** of $\mathcal{X}$ onto $[a, b]$ to be the projection $\langle x'_1, \ldots, x'_m \rangle$ but shrunk by the removal of redundant operations. There are two ways that the projection $\langle x'_1, \ldots, x'_m \rangle$ can be shrunk. First, all null operations can be removed. Second, whenever adjacent operations both increment the same range $[i, j]$, the first operation by some r_1 and the second operation by some r_2, the operations can be combined to a single $\text{Increment}(i, j, r_1 + r_2)$ operation. This processes can be repeated until every pair of adjacent Increment operations operate on distinct ranges.

The Increment–and–Freeze algorithm performs $\mathcal{S}$ on $A[1, 2, \ldots, n]$ via a simple divide-and-conquer approach. The algorithm first computes the shrunk projections of $\mathcal{S}$ onto $[1, \lfloor n/2 \rfloor]$ and $[\lfloor n/2 \rfloor + 1, n]$, which we call $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively. The algorithm then recursively applies the operation sequences $\mathcal{S}_1$ and $\mathcal{S}_2$ to the sub-arrays $A[1], \ldots, A[\lfloor n/2 \rfloor]$ and to $A[\lfloor n/2 \rfloor + 1], \ldots, A[n]$, respectively. The algorithm stops recursing once it gets to a sub-array of size 1, at which point the algorithm simply applies the operations in the sequence one after another.

We define the ***size*** of a subproblem to be the size of its shrunk projection. That is, if a recursive subproblem is responsible for a sub-interval $\mathcal{I}$ of the array, and size of the shrunk projection of $\mathcal{S}$ onto $\mathcal{I}$ is ℓ, then the subproblem has size ℓ.

At the i-th level of recursion, there are 2^i recursive subproblems. A single Increment operation in $\mathcal{S}$ could potentially project to many different subproblems. Nonetheless, we can prove that, for each level of recursion, the sum of the sizes of the recursive subproblems is $O(n)$.

Lemma 4.2. *For any discrete interval $\mathcal{I} = \{a, a+1, \ldots, b\}$, the shrunk projection $\mathcal{X}_\mathcal{I}$ of $\mathcal{S}$ onto $\mathcal{I}$ has size $|\mathcal{X}_\mathcal{I}| = O(|\mathcal{I}|)$.*

Proof. Say that an $\text{Increment}(i, j, r)$ operation appears ***actively*** in the shrunk projection $\mathcal{X}_\mathcal{I}$ if the range $[i, j]$ intersects $\mathcal{I}$ without fully containing $\mathcal{I}$, and say that the operation appears ***passively*** in $\mathcal{X}_\mathcal{I}$ if the range $[i, j]$ contains $\mathcal{I}$.

Whenever two consecutive Increment operations in $\mathcal{X}_\mathcal{I}$ both apply to the full range $\mathcal{I}$, they are combined together. Define $C_\mathcal{I}$ to be the number of Freeze operations that appear in $\mathcal{X}_\mathcal{I}$ plus the number of Increment operations that appear actively in $\mathcal{X}_\mathcal{I}$. Then

$$|\mathcal{X}_\mathcal{I}| \le 2C_\mathcal{I} + 1, \tag{2}$$

since no two consecutive operations in $\mathcal{X}_\mathcal{I}$ can both be passive appearances of Increment operations.

Since each $i \in \mathcal{I}$ gets frozen at most once by $\mathcal{S}$, the number of Freeze operations in $\mathcal{X}_\mathcal{I}$ is at most $|\mathcal{I}|$. Notice that, for each index $i \in \mathcal{I}$, the operation sequence $\mathcal{S}$ contains at most one Increment operation of the form $\text{Increment}(i, \text{next}(i)-1, 1)$ (if $\text{next}(i) \le n$) and one Increment operation of the form $\text{Increment}(\text{prev}(i+1), i, 1)$ (if $i + 1 \le n$). Thus the number of active Increment operations is at most $2|\mathcal{I}|$. This implies that $C_\mathcal{I} \le 3|\mathcal{I}|$, which by (2) gives $|\mathcal{X}_\mathcal{I}| = O(|\mathcal{I}|)$. □

Using Lemma 4.2, we can now analyze the Increment–and–Freeze algorithm.

Theorem 4.3. *Increment–and–Freeze computes the distance vector $\langle d_1, \ldots, d_n \rangle$ of a trace T of length n in time $O(n \log n)$ in the RAM model. The algorithm incurs cost $O(\frac{n}{B} \log n)$ in the external-memory model. Finally, the algorithm can be implemented in the CREW PRAM model to have work $O(n \log n)$ and span $O(n)$.*

Proof. The algorithm's correctness follows from Lemma 4.1. Thus we focus on the costs in each of the three models. We will repeatedly use the fact that, by Lemma 4.2, every recursive subproblem in the i-th level of recursion is guaranteed to have size $O(n/2^i)$.

At the i-th level of recursion, there are 2^i recursive subproblems each of which has size $O(n/2^i)$. It follows that, in the RAM model, there is $O(n)$ work per level of recursion and $O(n \log n)$ total work. The external-memory cost of a subproblem of size $O(n/2^i)$ is at most $O\left(\frac{n/2^i}{B} + 1\right)$. Additionally, if a subproblem has size B or smaller, the entire subproblem can be completed in $O(1)$ IOs (including all of its recursive descendants), meaning that we can treat its descendants as being free. Since we need to consider only the costs of subproblems where $n/2^i = \Omega(B)$, the cost of such a subproblem can be rewritten simply as $O\left(\frac{n/2^i}{B}\right)$. The total IO cost per level of recursion is therefore at most $O(n/B)$, implying a total external-memory cost of $O\left(\frac{n}{B}\log n\right)$ for the algorithm.

Finally, we can run the subproblems in each level of recursion in parallel. The i-th level of recursion therefore has span $O(n/2^i)$, implying a total span of $\sum_i O(n/2^i) = O(n)$. Since the work is $O(n \log n)$, this completes the analysis in the CREW PRAM model. □

5 AN EXTERNAL-MEMORY ALGORITHM

The algorithm from Section 4 achieves $O(\frac{n}{B}\log n)$ IOs in the external-memory model. In this section we introduce External–Increment–and–Freeze, which achieves an IO bound of

$$O\left(\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B}\right).$$

As discussed in Section 3, it suffices to achieve this bound for computing the distance vector $\langle d_1, \ldots, d_n\rangle$.

Part of what makes Increment–and–Freeze nice to work with is that it has essentially the same recursive structure as quicksort. This analogue extends to the external-memory setting: External–Increment–and–Freeze will have essentially the same relationship to Increment–and–Freeze as the external-memory distribution sort algorithm [1] has to quicksort.

Let c be a large positive constant to be determined later. The External–Increment–and–Freeze algorithm has the following recursive structure: If a subproblem is on an interval of size M/c or smaller (this is a ***base-case subproblem***), then it is solved in internal memory, and the components of the distance vector that it computes are then written to external memory. Otherwise, the subproblem is an ***internal subproblem***, and it is implemented to have recursive fanout M/B. In more detail, if an internal subproblem takes as input the shrunk projection $\mathcal{S}_{\mathcal{I}}$ of $\mathcal{S}$ of some interval $\mathcal{I}$, then it outputs the shrunk projections $\mathcal{S}_{\mathcal{I}_1}, \ldots, \mathcal{S}_{\mathcal{I}_r}$ on $r = M/B$ equal-sized intervals $\mathcal{I}_1, \ldots, \mathcal{I}_r$.

Since a subproblem's input $\mathcal{S}_{\mathcal{I}}$ may not fit into internal memory, the subproblem breaks the input into blocks of size $O(B)$ and processes each block one after another. Additionally, since the outputs $\mathcal{S}_{\mathcal{I}_1}, \ldots, \mathcal{S}_{\mathcal{I}_r}$ may also not fit in internal memory, the subproblem keeps a buffer of size $O(B)$ for each output $\mathcal{S}_{\mathcal{I}_i}$ and flushes the buffer to external memory whenever it fills up. We can perform the projection sequentially because the interval(s) that an operation projects to can be determined independently from other operations and when merging redundant operations we need only consider the previous operation in the shrunk projection.[2]

Theorem 5.1. *Assume that $n \geq B$. External–Increment–and–Freeze computes the distance vector $\langle d_1, \ldots, d_n\rangle$ for a trace of size n with an external-memory cost of*

$$O\left(\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B}\right).$$

Proof. We begin by confirming that base-case subproblems really are small enough to process in internal memory. Since each base-case subproblem is on an interval of size M/c, and since c is a sufficiently large positive constant, we have by Lemma 4.2 that the input to the subproblem takes space at most $M/2$. Thus the entire subproblem can be evaluated in internal memory, as claimed in the algorithm description.

Next we observe that, by design, every subproblem is on an interval of size $\Omega(B)$ (since the branching factor is M/B and each internal subproblem is on an interval of size $\Omega(M)$). Also observe that, since there are $O(\log_{M/B}(n/B))$ levels of recursion, and by Lemma 4.2 the total size of all subproblems at a given level of recursion is $O(n)$, the total size of all subproblems is $O(n\log_{M/B}(n/B))$. We will make use of both of these facts throughout the rest of the proof.

If a subproblem is on an interval of size $\ell = \Omega(B)$, then its input is guaranteed to have size $O(\ell)$ (by Lemma 4.2), and will therefore take time $O(\ell/B+1) = O(\ell/B)$ to read. Summing across the subproblems, the total time spent reading inputs is $O((n/B)\log_{M/B} n)$ IOs.

If an internal subproblem is on an interval of size $\ell = \Omega(M)$, then its M/B outputs will have cumulative size $O(\ell)$ (again, by Lemma 4.2). Thus, the subprobem will incur at most $O(\ell + M/B) = O(\ell/B)$ IOs writing its output. Summing across the internal subproblems, the total time spent writing outputs is $O((n/B)\log_{M/B} n)$ IOs.

Finally, if a base-case subproblem is on an interval of size $\ell = \Omega(B)$, then its output will consist of ℓ distance-vector entries, and will require $O(\ell/B + 1) = O(\ell/B)$ IOs to write to external memory. Summing over the base-case subproblems, the total time spent writing distance-vector entries to external memory is at most $O(n/B)$ IOs. □

6 ACHIEVING NEAR-OPTIMAL PARALLELISM

In this section, we show how to modify the Increment–and–Freeze algorithm in order to achieve near-optimal parallelism in the CREW PRAM model. The algorithm that we introduce, which we call Parallel–Increment–and–Freeze, achieves span $O(\log^2 n)$, work $O(n\log n)$, and parallelism $O(n/\log n)$.

Recall that the main algorithmic sub-routine used in Increment–and–Freeze is the following: We take as input the shrunk projection $\mathcal{S}_{\mathcal{I}}$ of $\mathcal{S}$ onto some interval $\mathcal{I}$, and we produce as output the shrunk projections $\mathcal{S}_{\mathcal{I}_1}$ and $\mathcal{S}_{\mathcal{I}_2}$ of $\mathcal{S}$ onto the two halves $\mathcal{I}_1$ and $\mathcal{I}_2$ of $\mathcal{I}$. We refer to this as a ***partition routine***. The partition routine takes $O(n)$ time in serial. Our main challenge is to implement a parallel version of the routine that still incurs $O(n)$ work, but that has span polylog n.

Our algorithm will make use of the ***parallel-prefix-sum*** operation. Given a sequence $a_1, a_2, \ldots, a_m$ of $m \leq n$ items, and given an associative operator $\circ$, a parallel-prefix sum computes $b_1, b_2, \ldots, b_m$ where $b_i = a_1 \circ a_2, \circ \cdots \circ a_i$. Parallel prefix sums have $O(m)$ work and $O(\log m)$ span in the CREW PRAM model [10].

One classic way to use a parallel prefix sum is to remove holes from a sequence: given a sequence $x_1, x_2, \ldots, x_m$, where some x_is are null, one can construct a new sequence $x'_1, x'_2, \ldots, x'_{m'}$ consisting only of the non-null x_i's from the original sequence. This task, which we refer to as ***sequence compression***, is performed as follows: (1) define $a_1, \ldots, a_m$ so that a_i indicates whether x_i is null; (2) use a parallel prefix sum to compute $b_1, \ldots, b_m$ where $b_i = \sum_{j=1}^{i} a_j$; and (3) build $x'_1, x'_2, \ldots$ where each non-null x_i becomes x'_{b_i}.

[2]There is a small subtlety here: whenever we write the buffer for $\mathcal{S}_{\mathcal{I}_i}$ to external memory, the first operation of the buffer may need to be combined with the operation that precedes it (i.e., the operation that was at the end of the most recent previous buffer for $\mathcal{S}_{\mathcal{I}_i}$ to have been written to external memory).

In addition to sequence compression, we will use parallel prefix sums in a second slightly more sophisticated way:

Lemma 6.1 (The Cluster Sum Lemma). *Consider a sequence of pairs $a_1, a_2, \dots, a_m$, where each a_i is either of the form $(1, 0)$ or of the form $(0, k_i)$ for some $k_i \in \mathbb{Z}$.*

Define the $\circ$ operator to combine two pairs $(a, b), (c, d) \in \{0, 1\} \times \mathbb{Z}$ using the formula

$$(a, b) \circ (c, d) = \begin{cases} (c, d) & \text{if } c = 1 \\ (a, b + d) & \text{otherwise.} \end{cases}$$

Then $\circ$ is associative, and the second coordinate of $(x_i, y_i) = a_1 \circ a_2 \circ \dots \circ a_i$ can be interpreted as follows: Let r be the largest integer such that $a_{i-r}, a_{i-r+1}, \dots, a_i$ all have zeros in their first components; then $y_i = \sum_{j=i-r}^{i} k_j$ if $r \geq 0$ and $y_i = 0$ otherwise.

Proof. We begin by confirming that $\circ$ is associative. Consider $(a, b) \circ (c, d) \circ (e, f)$.

If $e = 1$, then both $((a, b) \circ (c, d)) \circ (e, f)$ and $(a, b) \circ ((c, d) \circ (e, f))$ evaluate simply to $(1, f)$.

If $e = 0$ and $c = 1$, then both $((a, b) \circ (c, d)) \circ (e, f)$ and $(a, b) \circ ((c, d) \circ (e, f))$ evaluate to $(1, d + f)$.

Finally, if $e = 0$ and $c = 0$, then both $((a, b) \circ (c, d)) \circ (e, f)$ and $(a, b) \circ ((c, d) \circ (e, f))$ evaluate to $(a, b + d + f)$. Thus $\circ$ is associative.

Next we confirm the interpretation of $(x_i, y_i) = a_1 \circ a_2 \circ \dots \circ a_i$. Let r be the largest integer such that $a_{i-r}, a_{i-r+1}, \dots, a_i$ all have zeros in their first components. If $r = 0$, then $a_i = (1, 0)$, and it is straightforward to confirm that (x_i, y_i) will therefore also be $(1, 0)$.

Suppose $r > 0$. Since all of $a_{i-r}, a_{i-r+1}, \dots, a_i$ have 0s in their first coordinates, we have that

$$a_{i-r} \circ a_{i-r+1} \circ \dots \circ a_i = \left(0, \sum_{j=i-r}^{i} k_j\right).$$

If $r = i - 1$, then we are done. Otherwise, a_{i-r-1} must be $(1, 0)$, meaning that

$$a_1 \circ a_2 \circ \dots \circ a_{i-r-1} = (1, 0).$$

Thus

$$a_1 \circ a_2 \circ \dots \circ a_i = (1, 0) \circ \left(0, \sum_{j=i-r}^{i} k_j\right) = \left(1, \sum_{j=i-r}^{i} k_j\right).$$

□

We can now describe how to implement a partition routine in parallel. Recall that we are given as input the projection $\mathcal{S}_{\mathcal{I}}$ of $\mathcal{S}$ onto an interval $\mathcal{I}$, and we wish to produce $\mathcal{S}_{\mathcal{I}_1}$ and $\mathcal{S}_{\mathcal{I}_2}$, where $\mathcal{I}_1$ and $\mathcal{I}_2$ are the two halves of $\mathcal{I}$. Without loss of generality, we can focus on computing $\mathcal{S}_{\mathcal{I}_1}$.

We begin by replacing each operation in $\mathcal{S}_{\mathcal{I}}$ with its projection onto I_1. Next, we apply sequence compression to eliminate any null operations. Call the resulting sequence $x_1, x_2, \dots, x_m$.

Call x_i ***passive*** if it can be merged with x_{i+1}, ***semi-active*** if it can be merged with x_{i-1} but not with x_{i+1}, and ***active*** if it cannot be merged with either x_{i-1} or x_{i+1}. For each semi-active operation x_ℓ, and the maximal run $x_i, \dots, x_{\ell-1}$ of passive operations preceding it, we wish to compress all of $x_i, \dots, x_\ell$ into a single Increment operation. The new Increment operation should affect the same interval as did x_ℓ, but should have an increment amount given by $w_\ell := \sum_{j=i}^{\ell} k_j$, where $k_i, \dots, k_\ell$ are the increment amounts for each of $x_i, \dots, x_\ell$.

We can use a cluster-sum operation (Lemma 6.1) to compute w_ℓ for each semi-active operation x_ℓ. This tells each semi-active operation what its new increment amount should be. We can then use sequence compression (treating passive operations as null) to eliminate the passive operations. This results in the shrunk projection $\mathcal{S}_{I_1}$ that we wished to compute.

Each cluster-sum and sequence-compression operation has linear work and logarithmic span. Thus we have a work-efficient implementation of Parallel Partition with span $O(\log n)$. Within each level of recursion in the Parallel–Increment–and–Freeze, we can run the Parallel Partitions in parallel. Thus, across the $O(\log n)$ levels of recursion, we have total span $O(\log^2 n)$. Thus we have our desired result in the CREW PRAM model:

Theorem 6.2. *Parallel–Increment–and–Freeze is a CREW PRAM algorithm that computes the distance vector $\langle d_1, \dots, d_n \rangle$ for a trace T of size n. It has work $O(n \log n)$ and span $O(\log^2 n)$.*

7 PARAMETERIZING BY A MAXIMUM CACHE SIZE (OR THE UNIVERSE SIZE)

In this section, we extend Increment–and–Freeze to be parameterized by a *maximum cache size k* that we wish to consider. That is, rather than computing the full hit-rate curve $H_T(1), \dots, H_T(n)$, we instead compute $H_T(1), \dots, H_T(k)$. Note that, even if we wish to consider all cache sizes, one can still set $k = u$ where u is the number of distinct addresses[3] in T, since we trivially have that $H_T(i) = H_T(u)$ for all $i > u$.

The main result of this section is that Increment–and–Freeze can be modified to take time $O(n \log k)$ and $O(k)$ memory. The basic approach is to partition the input $T = \langle t_1, \dots, t_n \rangle$ into chunks $T = \{C_1, \dots, C_{n/k}\}$ of size $\Theta(k)$. We also show how to achieve strong external-memory and parallelism bounds, also parameterized by k.

Our basic approach is to process each chunk, one after another, using the standard Increment–and–Freeze algorithm. We perform additional bookkeeping in order to handle interactions between chunks. The approach is similar to the one taken in previous work by Niu et al. [21], with the exception of a modification that we introduce at the end of the section to achieve polylog n span.

Before continuing, it is convenient to perform a slight change of notation for how we discuss the distance vector. Define the ***forward distance vector*** $f = \langle f_1, f_2, \dots, f_n \rangle$ so that f_i is the number of distinct addresses in the sequence $t_{\text{prev}(i)+1}, \dots, t_i$. (In contrast d_i counts the distinct addresses in $t_i, \dots, t_{\text{next}(i)-1}$.) It is straightforward to use $\langle f_1, \dots, f_n \rangle$ in place of $\langle d_1, \dots, d_n \rangle$ (or even to simply convert between the two in SORT time). However, for this section, it will be more convenient to discuss the forward distance vector.

Also, because we only care about cache of size k or smaller, it suffices to compute $\min(k + 1, f_i)$ for each f_i. If $f_i \geq k + 1$, then we do not care about its specific value, since t_i is guaranteed to be a cache miss on all caches of size k or smaller.

For any prefix $P_i = C_1 \cdot C_2 \cdot \dots \cdot C_i$ of the chunks, define Q_i to be the subsequence of P_i that contains each address in P_i in sorted order by their final access times in P_i (from least-recently

[3]It is not necessary for u to be known in advance. In the case of $k = u$, k increases each time a unique address is encountered in T.

accessed to most-recently accessed). Finally, let $\overline{Q}_i$ be the suffix of Q_i consisting of the final k accesses (or all of Q_i if $|Q_i| \le k$).

Our next lemma tells us that, by performing Increment–and–Freeze on the trace $\overline{Q_i} \cdot C_i$ (that is, C_i concatenated onto $\overline{Q_i}$), we can correctly recover all values (truncated down to $k+1$) of the forward distance vector $\langle f_1, \dots, f_n \rangle$ that correspond to chunk C_i.

Lemma 7.1. *Suppose $C_i = \langle t_{a+1}, \dots, t_b \rangle$. Let $R_i = \overline{Q_i} \cdot C_i$ be the trace obtained by appending $\overline{Q}_i$ and C_i together, and define $\langle r_1, r_2, \dots \rangle$ to be the forward distance vector for R_i.*

Then the relationship between the r_is and the distances in the true forward distance vector $\langle f_1, \dots, f_n \rangle$ is as follows: for each $j \in \{1, 2, \dots, |C_i|\}$, we have

$$\min\left(k+1, r_{|\overline{Q}_i|+j}\right) = \min(k+1, f_{a+j}).$$

Proof. Define $m = |Q_i|$. We begin by considering the case where $m \le k$, meaning that $\overline{Q}_i = Q_i$. In this case we claim that, for each $j \in \{1, 2, \dots, |C_i|\}$, we have

$$r_{m+j} = f_{a+j}. \tag{3}$$

We will write $\overline{Q}_i$ as $q_1, q_2, \dots, q_m$ and C_i as $t_{a+1}, \dots, t_b$. Notice that f_{a+j} expands to

$$f_{a+j} = |\{t_{\text{prev}(a+j)+1}, \dots, t_{a+j}\}| \tag{4}$$

If $\text{prev}(a+j) \ge a+1$, then we will trivially have $r_{m+j} = d_{a+j}$. On the other hand, if $\text{prev}(a+j) \le a$, then r_{m+j} will behave as follows. Define $\overline{\text{prev}}(j)$ to be 0 if $t_j \notin \overline{Q}_i$, and to satisfy $q_{\overline{\text{prev}}(j)} = t_j$ otherwise. If $\text{prev}(a+j) \le a$, then

$$r_{m+j} = |\{q_{\overline{\text{prev}}(j)}, \dots, q_m\} \cup \{t_{a+1}, \dots, t_{a+j}\}|. \tag{5}$$

We wish to show that (4) and (5) evaluate to the same quantities.

Critically, by the definition of Q_i, we have that

$$\{q_{\overline{\text{prev}}(j)}, \dots, q_m\} = \{t_{\text{prev}(a+j)}, \dots, t_a\}. \tag{6}$$

This is because Q_i can be obtained from $t_1, \dots, t_a$ by removing any t_r, $r \le a$, such that t_r appears again in the sub-trace $t_{r+1}, \dots, t_a$.

By (6), we can conclude that the right sides of (4) and (5) are equal, meaning that $f_{a+j} = r_{m+j}$, as desired.

Finally, we must consider the case where $|Q_i| > k+1$, meaning that $\overline{Q}_i$ consists of the final $k+1$ elements of Q_i. Notice that this distinction only matters for f_{a+j} where $f_{a+j} \ge k+1$. Moreover, the distinction preserves the fact that $r_{|\overline{Q}_i|+j}$ (which would have been $r_{|Q_i|+j}$ if we set $\overline{Q}_i = Q_i$) is at least $k+1$. Thus $\min(k+1, r_{|\overline{Q}_i|+j}) = \min(k+1, f_{a+j})$, as desired. □

Bounded–Increment–and–Freeze works as follows: We process the chunks $C_1, C_2, \dots$ (each of which are size $O(k)$) one after another. To process a chunk $C_i = [t_{a+1}, t_b]$, we run Increment–and–Freeze on $\overline{Q}_i \cdot C_i$ and use Lemma 7.1 to recover $\min(f_{a+1}, k+1), \dots, \min(f_b, k+1)$. While processing C_i, we also compute $\overline{Q}_{i+1}$ (this is straightforward to do in time $O(k)$ using $\overline{Q}_i$ and C_i). This means that, once it is time to process C_{i+1}, we already have $\overline{Q}_{i+1}$.

Since $|\overline{Q}_i \cdot C_i| = O(k)$, the time to run Increment–and–Freeze on $\overline{Q}_i \cdot C_i$ is $O(k \log k)$. Bounded–Increment–and–Freeze therefore takes total time $O(n \log k)$. Similarly, if $k \ge B$, then the external-memory complexity of the algorithm is $O((n/B) \log k)$. Additionally, since all requests but $\overline{Q}_{i+1}$ can be discarded after processing C_i, we require only $O(k)$ memory.

The algorithm produces the truncated forward distance vector $\langle \min(f_1, k{+}1), \dots, \min(f_n, k{+}1) \rangle$, which can be then be used to compute the first k entries of the LRU hit-rate curve, while preserving the algorithm's complexity in every model (see discussion in Section 3). One subtlety here is that we wish to translate$\langle \min(f_1, k+1), \dots, \min(f_n, k+1) \rangle$ into a k-truncated LRU hit-rate curve in time $O(n \log k)$, rather than $O(n \log n)$. To do this, we can compute separate k-truncated hit-rate curves for each C_i (each takes time $O((n/k) \log k)$), and then sum the curves together.

Thus we have the following theorem:

Theorem 7.2. *Bounded–Increment–and–Freeze takes $O(n \log k)$ time and $O(k)$ memory to compute the first k entries of the LRU hit-rate curve on a trace of size n. Assuming $k = \Omega(B)$, the algorithm incurs $O((n/B) \log k)$ IOs in the external-memory model.*

We conclude the section by discussion external-memory and parallel versions of Bounded–Increment–and–Freeze. By using External–Increment–and–Freeze to process each chunk $\overline{Q}_i \cdot C_i$, we get an external-memory complexity of $O((n/B) \log_{M/B}(k/B))$.

Theorem 7.3. *Let $k, n \ge \Omega(B)$. External–Bounded–Increment–and–Freeze computes the first k entries of the LRU hit-rate curve of a trace of size n, and has external-memory complexity $O((n/B) \log_{M/B}(k/B))$.*

Parallelizing Bounded–Increment–and–Freeze requires a bit more effort. Naively, we get $\Theta(\log k)$-fold paralleism. Even if we use Parallel–Increment–and–Freeze to process each $\overline{Q}_i \cdot C_i$, we still get span $\Omega(\log^2 k \cdot n/k)$, where the n/k comes from the fact that we process the n/k chunks in serial.

To once again achieve span $O(\text{polylog}\, n)$, we need to compute $\overline{Q}_1, \dots, \overline{Q}_{n/k}$ in parallel, that way the chunks $\overline{Q}_1 \cdot C_1, \dots, \overline{Q}_{n/k} \cdot C_{n/k}$ can be processed in parallel.

Notice, however, that the $\overline{Q}_i$s can be obtained using a parallel-prefix-sum operation. Define $\overline{Q}(I)$ on any interval I to consist of the distinct addresses in I sorted in order of when they are last accessed in I (from least-recent final-access to most-recent final-access). Define the operator $\circ$ so that, for any two adjacent intervals I_1 and I_2 that collectively make up an interval I, we have that $\overline{Q}(I) = \overline{Q}(I_1) \circ \overline{Q}(I_2)$. It is straightforward to verify that $\circ$ is associative. It is also a straightforward exercise to compute $\overline{Q}(I_1) \circ \overline{Q}(I_2)$ with work $O(k \log k)$ and span polylog n; and to compute $\overline{Q}(I)$ for any interval I of size k with work $O(k \log k)$ and span polylog n. It follows that we can use a parallel prefix sum to compute $\overline{Q}_1, \overline{Q}_2, \dots, \overline{Q}_{n/k}$ with work $O(n \log k)$ and span $O(\text{polylog}\, n)$.

Thus we have the following theorem:

Theorem 7.4. *Parallel–Bounded–Increment–and–Freeze is a CREW PRAM algorithm that computes the first k entries of the LRU hit-rate curve of a trace of size n, while achieving work $O(n \log k)$ and span* polylog *n.*

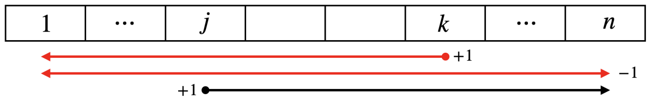

Figure 1: Prefix$(k, -1)$, in red, and Postfix$(j, 0)$, in black, are an efficient encoding of Increment$(j, k, 1)$ and Freeze(j). The increments outside of the range $[j, k]$ cancel and within the range they sum to 1 . The Postfix then freezes i.

Parallel–Bounded–Increment–and–Freeze pushes its parallelism beyond $\Theta(\log k)$ at the cost of additional memory consumption (as chunks must reside in memory simultaneously). Given a memory of size M, Parallel–Bounded–Increment–and–Freeze achieves parallelism $O((M/k) \log k)$. This is a factor $\log k$ improvement over using augmented-tree algorithms to process chunks in parallel. With M memory, this approach, i.e. PARDA [21], achieves $O(M/k)$ parallelism.

8 SYSTEMS ENGINEERING

In order for Increment–and–Freeze to be efficient in practice, there are several important engineering optimizations that need to be made. We describe here two optimizations that are especially impactful: (1) an alternative space-efficient encoding of the Increment and Freeze operations, and (2) an optimized approach to performing the partition routine. These optimizations are critical for minimizing memory usage. Collectively, they reduce memory usage by a factor of 6-12×. Roughly a factor of 4-6× comes from the encoding and roughly 1.5-2× comes from the improved partition. This memory reduction is, in turn, crucial for performance, as having a small memory footprint results in fewer cache misses.

An alternative encoding of Increment and Freeze. In our implementation of Increment–and–Freeze, we replace the Increment and Freeze operations with a different pair of operations, Prefix and Postfix. These operations implicitly operate on an interval $I = [a, b]$, which is the range covered by the current recursive subproblem:

- Prefix(t, r): This is equivalent to an Increment$(a, t, 1)$ followed by an Increment(a, b, r).
- Postfix(t, r): This is equivalent to an Increment$(t, b, 1)$, followed by a Freeze(t), and then an Increment(a, b, r).

Recall that, previously, our operation sequence $\mathcal{S}$ consisted of pairs of Increment and Freeze operations of the form Increment$(j, k, 1)$, Freeze(j). This can be replaced with the pair Prefix$(k, -1)$, Postfix$(j, 0)$ as shown in Figure 1. Here we are using the fact that incrementing the interval $[j, k]$ is equivalent to incrementing the prefix $(-\infty, k]$, decrementing $(-\infty, \infty)$, and then incrementing the suffix (k, ∞).

It is a straightforward exercise to express the projection of a Postfix/Prefix operation onto a smaller interval as a Postfix/Prefix operation on that interval. Moreover, whenever a Prefix operation effects an entire interval, it can be merged with the operation directly preceding it (*regardless of whether that operation is a Postfix or Prefix operation*). This replaces the merging of Increment operations used in the original Increment–and–Freeze algorithm.

One advantage of Postfix/Prefix operations is that they simplify the code. A second advantage is that, in a careful implementation, one can guarantee that the total number of operations at each level of recursion (across all subproblems) never exceeds $2n$.

An efficient implementation of the partition routine. We now turn our focus to implementing the partition routine, which takes as input the shrunk projection $\mathcal{S}_I = \langle x_1, x_2, \ldots \rangle$ of $\mathcal{S}$ on interval I, and outputs the shrunk projections $\mathcal{S}_{I_1}$ and $\mathcal{S}_{I_2}$ onto I's constituent halves. As notation, let $I_1 = [a, b]$ and $I_2 = [b + 1, c]$.

It turns out that $\mathcal{S}_I$ has the following property: If the i-th operation x_i in $\mathcal{S}$ is Prefix(t, r) for some t, then all of $x_1, x_2, \ldots, x_i$ are Prefix/Postfix operations whose first arguments are at most t. Moreover, if $t \leq b$, then one can confirm that these operations $x_1, x_2, \ldots, x_i$ collectively have no effect on I_2.

Thus, we can evaluate the partition routine as follows: We process $\mathcal{S}_I$ from right to left, building the projections $\mathcal{S}_{I_1}$ and $\mathcal{S}_{I_2}$ as we go. Once we encounter some x_i = Prefix(t, r) satisfying $t \leq b$, we complete the partition by simply adding $x_1, \ldots, x_i$ as a prefix of $\mathcal{S}_{I_1}$. In fact, with a careful implementation, one can even reuse the memory storing $x_1, \ldots, x_i$ so that the partition routine never even needs to touch $x_1, \ldots, x_{i-1}$. This serves both as a time optimization (since we need not process $x_1, \ldots, x_{i-1}$) and as a space optimization.

9 EXPERIMENTS

In this section, we evaluate the performance of Increment–and–Freeze (IAF) and Bounded–Increment–and–Freeze (Bound–IAF) as compared to two augmented tree algorithms. The first of these is PARDA [21], which uses a splay tree. The second is our own implementation that uses a Weight-Balanced Order-Statistic-Tree (OST) [19]. To disambiguate between the single-threaded version of PARDA (which is simply a splay-tree version of the augmented-tree algorithm) versus the multi-threaded version of PARDA (which achieves parallelism across time chunks), we will refer to the former as SPLAY and the latter as PARDA.

IAF achieves a speedup of 3-8× relative to SPLAY and uses 2-31× as much memory. Bound–IAF achieves a speedup of 3-6× relative to SPLAY and uses 0.9-1.32× as much memory. The main advantage of Bound-IAF is that it limits memory blowup in cases where the trace length n is significantly larger than the number of unique addresses u.

Both PARDA and Bound–IAF offer support for a user provided cache limit. Our baseline experiments do not impose a limit. For completeness, in Subsection 9.3, we also evaluate the effect of imposing cache limit.

PARDA, IAF, and Bound–IAF are multithreaded. We evaluate the parallelism of each algorithm by comparing it's serial running time to its parallel running time on multiple threads. By this metric, IAF and Bound–IAF achieve comparable speedups to PARDA, while using significantly less memory on large numbers of threads. Because IAF and Bound-IAF are faster than PARDA in serial, they continue to be faster in parallel.

On our largest workload, serial Bound–IAF achieves a speedup of nearly 10× while consuming only 1.33× the memory when compared with the Order-Statistic-Tree. In absolute terms, finding the hit-rate curve on a trace of this size took OST over 13 hours, whereas

Name	Requests	IDs	Requests per ID
Tiny	4e+7	2e+5	200
Small	1e+8	4e+6	25
Medium	5e+8	2e+7	25
Large	1e+9	1.6e+8	6.25
Huge	1e+10	2.68e+8	37.25

Table 1: We evaluate Increment–and–Freeze, OST, and PARDA on these synthetic workloads.

Bound–IAF took only an hour and 24 minutes. Computing the hit-rate curve in parallel with 48 threads increases the memory usage of Bound–IAF to 1.41× that of the Order-Statistic-Tree while increasing the speedup to over 66×, for a final running time of only 12 minutes.

9.1 Experimental Setup

Machine. We implemented IAF, Bound–IAF, and OST as C++17 libraries. We ran our experiments on a Dell Precision 7820 with 24-core 2-way hyperthreaded Intel(R) Xeon(R) Gold 5220R CPU @ 2.20GHz, and 64GB 4x16GB DDR4 2933MHz RDIMM ECC Memory.

Workloads. To test the systems we run them on a variety of synthetic traces. We use the following distributions: uniformly random and Zipfian distributions with α values of 0.1, 0.2, 0.4, 0.6, and 0.8. We use a trace from each distribution at the sizes in Table 1.

For each size, we report the average runtime and memory usage to compute the hit-rate curve of each trace. For each algorithm, we observe a deviation in runtime of 5-20% and in memory usage of 0-17% when comparing traces of the same size drawn from different distributions. PARDA has the highest deviation in runtime, likely due to its splay tree performing better on biased distributions. The trace with distribution Zipfian $\alpha = 0.8$ is the quickest and has the lowest memory usage across all algorithms.

If each address corresponds to a 4KiB page, then our workload sizes represent address spaces of size 781MiB for Tiny and 1TiB for the Huge workload. In reality, addresses may correspond to smaller or significantly larger objects. We remark that Increment–and–Freeze can be augmented to support objects of varying size.

Running Experiments. We tested the performance of OST and Bound–IAF at every input size. We tested IAF on each input size except for Huge because its memory consumption would have exceeded that of our machine. In our experiments, PARDA experienced segmentation faults on traces of size greater than Medium. When reporting running time results for PARDA, we exclude the time to pre-partition the trace into a disjoint trace for each thread.

9.2 Serial Performance

Runtime. The average runtime of computing the hit-rate curve for each input size and each algorithm is given in Table 2a. Both variants of IAF significantly outpace the augmented-tree approaches. IAF outperforms SPLAY by a factor of 3-8×, OST by 4-10×, and Bound–IAF by 1.08-1.29×. IAF outperforms Bound–IAF because although Bound–IAF has (slightly) better asymptotic performance it also pays higher constants due to reprocessing $\overline{Q}_i$ in each chunk.

Our results also support the conclusion that splay-trees offer better performance than standard augmented tree approaches. SPLAY outperformed OST by 10-30%, although, the gap in performance shrank as the size of the trace grew.

Memory consumption. The memory required to compute the hit-rate curve for each input size and algorithm is given by Table 2b. IAF uses significantly more memory than the augmented-tree approaches while Bound–IAF's memory consumption is much closer to that of SPLAY and OST.

The reason for the difference in memory usage between IAF and the other algorithms is that for the tested traces, $u < n$, often by more than an order of magnitude. As expected, we observe that

System	Tiny	Small	Medium	Large	Huge
SPLAY	31.6	198	1.33e+3	-	-
OST	45.0	229	1.45e+3	3.60e+3	4.75e+4
IAF	11.2	32.0	176	377	-
Bound-IAF	12.0	40.0	211	486	5.07e+3

(a) Average runtime in seconds.

System	Tiny	Small	Medium	Large	Huge
SPLAY	31.0	623	2.89e+3	-	-
OST	24.4	401	1.95e+3	1.55e+4	2.67e+4
IAF	957	2.68e+3	1.34e+4	2.85e+4	-
Bound-IAF	35.1	573	2.59e+3	1.80e+4	3.54e+4

(b) Average memory usage in Mebibytes.

Table 2: Increment–and–Freeze and Bounded–Increment–and–Freeze are faster than both SPLAY and OST by a factor of up to 9× with minimal memory overhead.

System	Tiny	Small	Medium	Large	Huge
PARDA	4.48	40.4	290	-	-
IAF	2.20	5.30	26.9	64.5	-
Bound-IAF	2.22	6.66	34.0	79.7	777

(a) Average runtime in seconds.

System	Tiny	Small	Medium	Large	Huge
PARDA	869	1.22e+4	6.08e+4	-	-
IAF	1090	3.02e+3	1.41e+4	2.97e+4	-
Bound-IAF	109	793	3.12e+3	1.92e+4	3.62e+4

(b) Average memory usage in Mebibytes.

Table 3: Increment–and–Freeze and Bounded–Increment–and–Freeze run up to 10× faster and use much less memory than PARDA when running with 16 threads.

IAF's relative memory usage is highly sensitive to the difference between n and u. For Tiny, where $n/u = 200$, IAF's memory usage is 30× that of SPLAY and 39× that of OST. Whereas for Large, where $n/u = 6.3$, its memory usage is only 1.8× that of OST.

We also observe that the Order-Statistic-Tree approach is the most memory efficient, generally using two-thirds of the memory SPLAY does. SPLAY's memory usage is in part a reflection of its treatment of addresses as strings rather than integers. This is also why Bound–IAF uses less memory than SPLAY on large inputs.

9.3 Cache Size Limit

We test the performance benefit of providing a cache size limit to PARDA and Bound–IAF for each trace size. These limits are 7.5e+4, 1.5e+6, 8e+6, 6.7e+7, and 6.7e+7 respectively. For objects of size 4KiB, these limits correspond to caches of size 293MiB for Tiny and up to 256GiB for Large and Huge.

Bound–IAF derives far more benefit from this cache limit than PARDA. For trace sizes of Tiny to Medium PARDA's runtime is reduced by 1%-2% and memory consumption by 0%-1.6% when supplied with these cache limits. Bound–IAF's runtime is reduced by 13%-21.1% and its memory consumption is reduced by 26%-60%. The variation in runtime reduction is dependent upon the trace size, and is likely a result of CPU cache behavior.

9.4 Parallel Performance

The result of using more threads to compute the hit-rate curve is summarized in Figure 2. To compute the self-relative speedup for an algorithm on a given number of threads and on a given input size, we divide the algorithm's serial time by its parallel time, and take the geometric mean across the input distribution types.

The absolute runtime and memory consumption with 16 threads is shown in Table 3. Here we can see the significance of PARDA's memory scaling linearly with the thread count. At 16 threads, processing a Tiny trace requires PARDA to use over 8× the memory as Bound–IAF. Since the IAF variants achieve similar self-relative speedup to PARDA, they remain faster than PARDA in our experiments as thread count varies. In fact, on traces of size Small or larger, PARDA's parallel performance never reaches that of either IAF variant's serial performance.

IAF's somewhat limited parallel scaling is not surprising as it has $O(\log n)$ parallelism. For our workloads, $O(\log n)$ tops out at roughly 30. It is likely that a well-engineered implementation of PARALLEL–INCREMENT–AND–FREEZE could achieve better scaling. This is left to future work.

9.5 64-Bit Addresses

In all our experiments it was sufficient for both variants of IAF to use 32-bit integers for storing addresses and counters. This is because the number of requests and addresses was less than 2^{32} for all inputs except Huge. For Huge, Bound–IAF's chunks contained roughly 1 billion requests, so they could be indexed by 32 bits.

If the number of addresses or the chunk size were to exceed 32 bits, then all of the algorithms (IAF, Bound-IAF, OST, PARDA, and SPLAY) would need to be converted to use 64-bit integers. For IAF and Bound-IAF executed in serial, switching to 64-bit integers

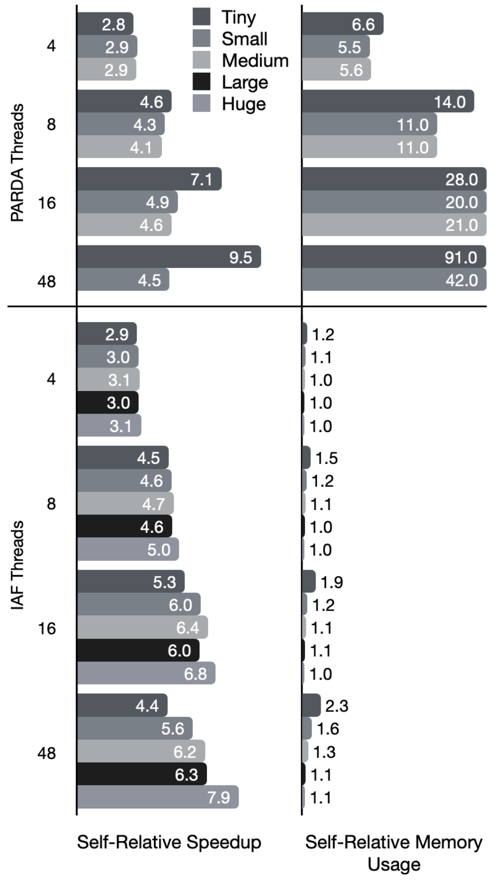

Figure 2: INCREMENT–AND–FREEZE achieves comparable speedups to PARDA with additional threads without experiencing a blowup in memory footprint.

resulted in a memory consumption increase of at most 2× and a runtime increase of at most 1.11×.

ACKNOWLEDGMENTS

Michael A. Bender, Daniel DeLayo, and Evan West are supported in part by NSF grants CCF-2247577, CCF-2118830, CCF-2106827, CSR-1763680, and CNS-1938709. Daniel DeLayo is supported in part by a GAANN fellowship. William Kuszmaul is funded by a Fannie and John Hertz Fellowship and an NSF GRFP Fellowship.

This research was also partially sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] Alok Aggarwal and S Vitter, Jeffrey. 1988. The input/output complexity of sorting and related problems. *Commun. ACM* 31, 9 (1988), 1116–1127.

[2] George Almási, Călin Caşcaval, and David A. Padua. 2002. Calculating stack distances efficiently. In *Proceedings of the 2002 Workshop on Memory System Performance (MSP)*. Berlin, Germany, 37–43. https://doi.org/10.1145/773146.773043

[3] Laszlo A. Bélády. 1966. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal* 5, 2 (1966), 78–101. https://doi.org/10.1147/sj.52.0078

[4] Laszlo A. Bélády and Frank P. Palermo. 1974. On-line measurement of paging behavior by the multivalued MIN algorithm. *IBM Journal of Research and Development* 18, 1 (Jan. 1974), 2–19. https://doi.org/10.1147/rd.181.0002

[5] Michael A. Bender, Daniel DeLayo, Bradley C. Kuszmaul, William Kuszmaul, and Evan West. 2022. Increment-and-Freeze source code. https://github.com/etwest/Increment-and-Freeze.

[6] B. T. Bennett and V. J. Kruskal. 1975. LRU stack processing. *IBM Journal of Research and Development* 19, 4 (July 1975), 353–357. https://doi.org/10.1147/rd.194.0353

[7] Daniel S Berger, Nathan Beckmann, and Mor Harchol-Balter. 2018. Practical bounds on optimal caching with variable object sizes. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)* 2, 2 (2018), 1–38.

[8] Gianfranco Bilardi, Kattamuri Ekanadham, and Pratap Pattnaik. 2011. Efficient stack distance computation for priority replacement policies. In *Proceedings of the 8th ACM International Conference on Computing Frontiers (CF)*. https://doi.org/10.1145/2016604.2016607

[9] Gianfranco Bilardi, Kattamuri Ekanadham, and Pratap Pattnaik. 2017. Optimal on-line computation of stack distances for MIN and OPT. In *Proceedings of the Computing Frontiers Conference (CF)*. 237–246. https://doi.org/10.1145/3075564.3075571

[10] Guy E Blelloch. 1993. Prefix sums and their applications. In *Synthesis of Parallel Algorithms*, John H Reif (Ed.). Morgan Kaufmann Publishers Inc.

[11] Guy E Blelloch, Phillip B Gibbons, and Harsha Vardhan Simhadri. 2010. Low depth cache-oblivious algorithms. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 189–199.

[12] Guy E Blelloch and Bruce M Maggs. 2010. Parallel algorithms. In *Algorithms and Theory of Computation Handbook: Special Topics and Techniques*, Mikhail J Atallah and Marina Blanton (Eds.). 25–25.

[13] Daniel Byrne. 2018. A survey of miss-ratio curve construction techniques. https://arxiv.org/pdf/1804.01972.pdf

[14] Zachary Drudi, Nicholas JA Harvey, Stephen Ingram, Andrew Warfield, and Jake Wires. 2015. Approximating hit rate curves using streaming algorithms. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[15] David Eklov and Erik Hagersten. 2010. StatStack: Efficient modeling of LRU caches. In *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 55–65.

[16] Changpeng Fang, S Can, Soner Onder, and Zhenlin Wang. 2005. Instruction based memory distance analysis and its application to optimization. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 27–37.

[17] Lulu He, Zhibin Yu, and Hai Jin. 2012. FractalMRC: online cache miss rate curve prediction on commodity systems. In *IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*. 1341–1351.

[18] Yul H Kim, Mark D Hill, and David A Wood. 1991. Implementing stack simulation for highly-associative memories. *ACM SIGMETRICS Performance Evaluation Review* 19, 1 (1991), 212–213.

[19] Charles Eric Leiserson, Ronald L Rivest, Thomas H Cormen, and Clifford Stein. 1994. *Introduction to Algorithms*. Vol. 3. MIT press.

[20] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. 1970. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal* 9, 2 (1970), 78–117. https://doi.org/10.1147/sj.92.0078

[21] Qingpeng Niu, James Dinan, Qingda Lu, and Ponnuswamy Sadayappan. 2012. PARDA: A fast parallel reuse distance analysis algorithm. In *IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*. 1284–1294.

[22] Frank Olken. 1981. *Efficient Methods for Calculating the Success Function of Fixed Space Replacement Policies*. Technical Report LBL-12370. Physics, Computer Science & Mathematics Division, Lawerence Berkeley Laboratory, University of California. M.S. thesis.

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

[24] Daniel D Sleator and Robert E Tarjan. 1985. Amortized efficiency of list update and paging rules. *Commun. ACM* 28, 2 (1985), 202–208.

[25] Rabin A. Sugumar. 1993. *Multi-configuration simulation algorithms for the evaluation of computer architecture designs*. Ph. D. Dissertation. University of Michigan.

[26] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, California, USA, 95–110. https://www.usenix.org/conference/fast15/technical-sessions/presentation/waldspurger

[27] Carl A Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. 2017. Cache Modeling and Optimization using Miniature Simulations. In *USENIX Annual Technical Conference (ATC)*. 487–498.

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

[29] Jiangwei Zhang and YC Tay. 2020. PG2S+: Stack distance construction using popularity, gap and machine learning. In *Proceedings of The Web Conference (WWW)*. 973–983.

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

[31] Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31, 6 (Aug. 2009). https://doi.org/10.1145/1552309.1552310 Article 20.