EFRC is Adapted to Access-rate Imbalance: A Class of Efficient and Fair Replacement Algorithms for Cache Sharing

LIANG AI, Sun Yat-Sen University, Guangzhou, China
PENGFEI CHEN, Sun Yat-Sen University, Guangzhou, China
GUANGBA YU, Sun Yat-Sen University, Guangzhou, China
YUHUI DENG, Jinan University, Guangzhou, China
XIAO QIN, Auburn University, Auburn, The United States of America

Co-located workloads demand fine-grained management of caching control for enhanced deterministic performance. Traditionally, minimizing cache interference has relied on hardware mechanisms that offer coarse cache resource allocation across cores or the last-level cache. More often than not, existing cache replacement algorithms aim to optimize cache hit efficiency depending on single workload access characteristics. In this article, we propose a series of replacement algorithms for multi-workload shared caches, radically enhancing the deterministic performance of co-located workloads. We devise a class of replacement algorithms for cache sharing, referred to as Efficient and Fair Replacement Cache or EFRC. To strick a balance between performance and complexity, we divide the class of algorithms into Type I and Type II according to the data access characteristics. The common idea of our EFRC-I and EFRC-II algorithms is to judiciously adjust resources for a diversity of workloads or data characteristics online by triggering various resource allocation policies through a corresponding ghost cache preceptor. Our extensive experiments unveil that the EFRC-I and EFRC-II algorithms exhibit a handful of advantages. Firstly, the algorithms support multiple workloads sharing cache—and are compatible with single workload scenarios. Secondly, the cache-hit performance of our proposed EFRC-I and EFRC-II algorithms is superior to that of the existing ARC, LIRS, CACHEUS, and SIEVE algorithms in multi-workload shared caches. Importantly, our technique enhances the fairness of co-located workloads. In particular, when access rates of co-located workloads are imbalanced, our method demonstrates clear superiority over the baseline and the aforementioned state-of-the-art algorithms.

CCS Concepts: • **Information systems** → *Hierarchical storage management;*

Additional Key Words and Phrases: Cache, cache sharing, cache replacement policy, fairness

The research is supported by the National Natural Science Foundation of China under Grant No.62272495, the Guangdong Basic and Applied Basic Research Foundation under Grant No.2023B1515020054, and the International Cooperation Project of Guangdong Province under Grant No.2025A0505020058. This research is also sponsored by Tencent.
Authors' Contact Information: Liang Ai, Sun Yat-Sen University, Guangzhou, Guangdong, China; e-mail: ailiang3@mail2.sysu.edu.cn; Pengfei Chen (corresponding author), Sun Yat-Sen University, Guangzhou, Guangdong, China; e-mail: chenpf7@mail.sysu.edu.cn; Guangba Yu, Sun Yat-Sen University, Guangzhou, Guangdong, China; e-mail: yugb5@mail2.sysu.edu.cn; Yuhui Deng, Jinan University, Guangzhou, Guangdong, China; e-mail: tyhdeng@jnu.edu.cn; Xiao Qin, Auburn University, Auburn, Alabama, The United States of America; e-mail: xqin@auburn.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).
ACM 1553-3077/2026/11-ART42
https://doi.org/10.1145/3799981

ACM Reference Format:
Liang Ai, Pengfei Chen, Guangba Yu, Yuhui Deng, and Xiao Qin. 2026. EFRC is Adapted to Access-rate Imbalance: A Class of Efficient and Fair Replacement Algorithms for Cache Sharing. *ACM Trans. Storage* 22, 4, Article 42 (November 2026), 35 pages. https://doi.org/10.1145/3799981

1 Introduction

Performance optimization and resource management for co-located workloads are common and important problems in computing [20, 30]. The cache is a crucial bottleneck resource that has a significant impact on computational performance. Co-locating workloads to enhance system resource efficiency may result in cache interference issues [58]. Most existing approaches to reducing cache interference or enhancing determinism are anchored on system-level hardware mechanisms. To divide portions of way-based **last-level cache** (**LLC**) resources among tasks [7, 52], one may either adjust CPU core resource allocation for co-located workloads [12] or utilize Intel's **Cache Allocation Technology** (**CAT**). In addition, a series of innovative cache allocation mechanisms is proposed based on the co-located workloads' behaviors observed in simulations [25, 28, 40].

While these methods may partially mitigate cache interference, they are also flawed in two ways. Firstly, these approaches are inherently based on partitioning techniques and can be limited by hardware cache resource management granularity, such as the commonly used way-based CAT technology. The fine-grained resource management approach not only helps to improve the resource utilization efficiency of the cache but also improves the control of the cache to better meet the needs of software services. In the context of elevated performance criteria for software services, such as microsecond **quality of service** (**QoS**) targets [15, 16], which fundamentally necessitate near-uniform processing times for each individual request [8, 33, 34], the cache requires enhanced control. Indeed, the increasing trend of more detailed performance monitoring and resource management at the hardware level is in line with this [44, 56]. Secondly, these approaches translate the problem of mitigating cache interference or contention into the problem of how to dynamically adjust workload-based resource allocation, but neglect the study of synergy with cache replacement strategies. On the one hand, most of these methods default to using algorithms such as **Least Recently Used** (**LRU**) [14], which solely take into account the allocation of resources for the cache, without factoring in the cache replacement policy. On the other hand, there has also been a trend in the research of cache replacement strategies based on data access characteristics [31, 53]. This is essentially a resource allocation problem based on data characteristics. For example, the ARC algorithm selects the data blocks to be replaced in the cache by dynamically allocating and adjusting the size of two cache queues based on recency and frequency characteristics [31]. Therefore, given that it is also a resource allocation issue, is there a possibility of a convergent strategy?

When considered as a whole, these factors indicate a fine-grained, convergent approach to management that caching necessitates. In this regard, we propose a special cache replacement algorithm for handling multiple workloads. This algorithm enhances the overall cache hit rate while ensuring that the hit rates of each workload are met (i.e., preventing interference). We present this scenario in Figure 1. It should be noted that, although the research problem of this article is partially abstracted from Figure 1(a), given that it is still a cache replacement algorithm, the research approach of this article is mainly based on Figure 1(b). The objective is to validate the feasibility of Figure 1(c) at the algorithmic level. Nevertheless, it is believed that the algorithm still possesses some applicability at the software level, such as memory cache.

As described in a previous study [18], existing cache replacement algorithms that are designed to achieve a single objective of enhancing hit rates are insufficient in scenarios where multiple workloads share caches. Furthermore, as emphasized in [39], when workloads access the shared

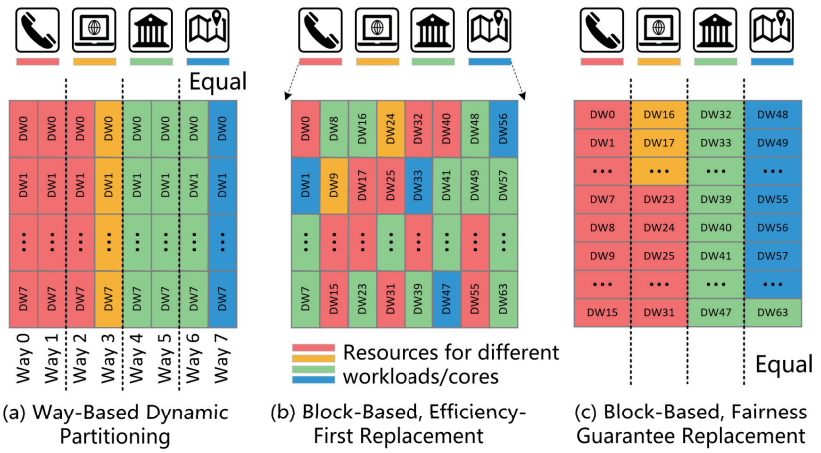


Fig. 1. Different schemes. (a) Common cache partitioning scheme: good cache efficiency and fairness, not involved in fine-grained cache replacement management. (b) General cache replacement scheme: excellent cache efficiency, but not aware of the presence of upper-level workloads and not provide fairness. (c) The scheme in this article: fine-grained cache resource management that balances cache efficiency and workload fairness.

cache at random access rates, especially under highly imbalanced access rate ratios, such policies are unable to guarantee fair resource allocation. Even workloads that demonstrate exceptional hit performance in single workload scenarios may experience substantial performance degradation in shared environments if their relative access rates are low. Additionally, although the objectives of cache hit efficiency and workload fairness appear to be inherently conflicting, it is demonstrated in this article that these goals can be effectively balanced at the block granularity level of caching.

In this article, we develop the EFRC-I and EFRC-II algorithms. The difference in the ordinal numbers is the number of data features utilized in each algorithm. Indeed, the EFRC multi-workload algorithmic framework is extensible, thereby allowing for the design of alternative algorithms based on different data feature classifications. The EFRC-I algorithm is based only on the recency feature, and its design was primarily driven by the principles of simplicity and high cost-effectiveness. In contrast, EFRC-II incorporates both recency and frequency characteristics, increasing its complexity. Nevertheless, it is able to achieve greater performance gains through the leveraging of richer data access information. Despite the differences in the dimensionality of the information utilized, both EFRC-I and EFRC-II algorithms have both efficiency and fairness resource strategies. These strategies are consistent across different data feature dimensions, thereby ensuring the success of the algorithms in achieving their objectives during operation.

We conduct an array of experiments to validate the effectiveness of the EFRC-I and EFRC-II algorithms. In the scenarios where multi-workload run with various access rates, our EFRC-I and EFRC-II algorithms have been demonstrated to exhibit high hit performance, accompanied by enhanced fairness. The results exhibit the anti-cheating characteristics of our EFRC-I and EFRC-II, which resist the effects of access rate imbalance. Furthermore, our algorithm seamlessly scales and adapts to a variety of cache sharing scenarios regardless of the number of shared workloads. The experimental results also reveal that our EFRC-I and EFRC-II algorithms have a low overhead.

In summary, we make the following contributions.

— We propose an efficient and fair replacement cache to enhance the determinism of shared workloads' hit performance.

— We present a comprehensive set of metrics gauging fairness measures and outline the fundamental objectives and criteria that shared replacement caching algorithms must meet.
— We design the class of EFRC algorithms to address challenging issues that must be overcome in shared caching algorithms.
— Our extensive experiments demonstrate the effectiveness of our EFRC-I and EFRC-II algorithms.

The rest of this article is organized as follows. Section 2 describes the background and motivation of this article. Section 3 presents the proposed EFRC-I and EFRC-II algorithms. Section 4 discusses our experimental evaluation results. Finally, in Section 6, we present the conclusions of this study.

2 Background and Motivation

2.1 Fairness Metric

There are various definitions of fairness metrics [2, 13, 23, 37], such as Jain's Fairness Index [19] based on the degree of "similarity" in the performances of multiple workloads. Due to the objective differences in hit rates between workloads, aiming for all workloads to achieve the same rate is impractical, akin to striving for everyone to have Warren Buffett's wealth. Therefore, the fairness is defined as allowing each workload to achieve its highest potential hit rate. Put simply, the marginal benefit of each workload's hit rate is the same under a given resource allocation. Alternatively, this can be explained using the **miss rate curves** (**MRC**) theory, which entails equalizing the derivatives of the MRC across all workloads, as detailed in Section 5.2.

We use the harmonic mean of individual workload hits as a fairness metric [11, 51]. Harmonic mean is used for fairness because it penalizes imbalance; it drops sharply when any one value is low, even if the others are high. The Harmonic Fairness is expressed as follows:

$$Harmonic\ Fairness = \frac{N}{\sum_i \frac{Hits_i^{Equal}}{Hits_i^{Share}}}, i \in \{1, 2, \cdots, N\}, \tag{1}$$

where we assume that the cache is shared by N workloads in which $Hits_i^{Equal}$ denotes the number of hits for workload i when the cache resource is equally shared among N workloads, and $Hits_i^{Share}$ is the number of hits for workload i when the cache resource is shared by N workloads in an arbitrary way. It should be noted that the terms of "Equal" and "Share" in this context emphasize the proportion of resources allocated to each workload. In this sense, "Equal" refers to an equal share, emphasizing the even distribution of resources. "Share" simply highlights that the resource is being shared, without specifying the method of distribution.

To compare the performance across multiple algorithms, we apply the Equal-LRU as a baseline. The cache resource of the Equal-LRU algorithm is divided equally among N workloads,—and each workload adopts the LRU algorithm. Therefore, the fairness is formalized as

$$Harmonic\ Fairness = \frac{N}{\sum_i \frac{Hits_i^{Equal-LRU}}{Hits_i^{Share}}}, i \in \{1, 2, \cdots, N\}. \tag{2}$$

The subsequent fairness metric is defined by Equation (2), which can effectively evaluate the algorithm's ability to balance cache hit efficiency and workload performance. In Equation (2), Equal-LRU is a baseline algorithm, which is fixed in the numerator. Other algorithms need to be incorporated into the denominator to calculate the Harmonic Fairness based on Equation (2). Therefore, the fairness value of Equal-LRU under the definition in Equation (2) is 1. For convenience, we will occasionally use EQ-LRU to refer to the Equal-LRU algorithm in the following sections.

Here is an example of the calculation. Suppose there are two workloads, w_1 and w_2, sharing a cache. w_1 accesses the cache 10 times, and w_2 accesses the cache 20 times. We now want to calculate the Harmonic Fairness using the ARC algorithm in the above experimental setup. First, we need to record the cache hits using the EQ-LRU algorithm. Assume that there are 5 cache hits for w_1 and 10 cache hits for w_2. With the ARC algorithm, there are 2 cache hits for w_1 and 16 cache hits for w_2. The Harmonic Fairness of the ARC algorithm, as calculated based on the definition in Equation (2), should be 0.64. According to the definition in Equation (2), the Harmonic Fairness of the ARC algorithm is calculated as

$$\frac{2}{\left(\frac{5}{2}+\frac{10}{16}\right)} = 0.64.$$

This value is lower than the Harmonic Fairness value of 1 for the EQ-LRU algorithm. However, the cache hit rate for the EQ-LRU algorithm is 50%, while the cache hit rate for the ARC algorithm is 60%.

Nonetheless, the Harmonic Fairness metric does not intuitively reflect the loss of each workload. If several algorithms can attain the same fairness value, it remains unclear which one is the winner. Moreover, an algorithm could have multiple tradeoffs between cache hit efficiency and workload performance with the same fairness value. Therefore, it is ambiguous which of these balances is optimal. We define the `MaxLoss` metric in Equation (1) below as a supplementary means to measure the loss of shared workload.

$$MaxLoss = max_i \frac{Hits_i^{Equal-LRU} - Hits_i^{Share}}{Hits_i^{Equal-LRU}}, i \in \{1, 2, \cdots, N\}. \tag{3}$$

In summary, an algorithm delivers prominent performance in a multi-workload sharing scenario when the algorithm has a large value of cache hit efficiency and fairness,—as well as a small maxloss value.

2.2 Caching Algorithms

Caching algorithms are block-based management granularities that achieve high caching efficiency by determining which blocks remain in a limited cache space. Since the introduction of the LRU algorithm in the early 1980s, the field has produced a rich literature [4, 10, 22, 36, 38, 46, 53, 62]. In this subsection, we provide a concise overview of select state-of-the-art or recently published algorithms.

LRU: LRU [14] is a relatively simple and widely used caching algorithm. As the algorithm's design is based on the recency feature, the workloads that perform optimally on it are also referred to as `LRU-friendly` workloads. In contrast, the workloads that perform optimally on the **Least Frequently Used (LFU)** algorithm [6], which is based on the frequency feature, are classified as `LFU-friendly` workloads. A notable limitation of the LRU algorithm is its suboptimal performance on `scan` workloads. In a `scan` workload, blocks are accessed only once, which results in the eviction of meaningful blocks from the LRU cache.

Adaptive Replacement Cache (ARC): ARC [31] is an algorithm that adaptively recognizes both recency and frequency of access. The ARC algorithm employs two LRU queues, designated T_1 and T_2. The T_1 queue represents the recency feature, which stores blocks accessed only once recently, while the T_2 queue represents the frequency feature, which stores blocks accessed at least twice recently. ARC is capable of dynamically adjusting the size of the T_1 and T_2 queues in accordance with the access characteristics of the workload. As a consequence of the fact that the blocks accessed only once in the `scan` workload are stored solely in the T_1 queue, the pollution of the T_2 queue is effectively avoided.

Low Interference Recency Set (LIRS): The LIRS [21] algorithm is based on **inter-reference recency** (**IRR**), also referred to as reuse distance, which is defined as the number of distinct blocks between two consecutive accesses to the same block in a sequence of requests. The LIRS algorithm maintains a **Low Inter-reference Recency** (**LIR**) set and a **High Inter-reference Recency** (**HIR**) set. The LIR set contains blocks that have a short reuse distance for recent accesses, while the HIR set stores the accessed blocks with a larger reuse distance, where the proportion of HIR blocks in the cache is very small. The fundamental principle of the LIRS algorithm is to retain the blocks with smaller IRR in the cache. Consequently, the blocks accessed only once in the `scan` workload do not pollute the LIR set.

Cacheus: CACHEUS [42] is an adaptive, learning-from-multiple-experts algorithm that performs well on four workloads: `LRU-friendly`, `LFU-friendly`, `Scan`, and `Churn`. The `churn` workload refers to a large number of newly accessed blocks that are repeatedly accessed with equal probability. The ARC and LIRS algorithms are unable to differentiate between the hotness of accessed blocks in `churn` workloads based on recent access frequency and IRR information. Consequently, these algorithms are unsuitable for use in `churn` workloads. In contrast, the CACHEUS experts comprise a **scan-resistant version of the LRU** (**SR-LRU**) and a **churn-resistant version of the LFU** (**CR-LFU**) algorithm, enabling the algorithm to perform optimally across a range of workloads.

Sieve: SIEVE [57] is an efficient turn-key eviction algorithm. The two fundamental principles of the algorithm are lazy promotion and quick demotion. Lazy promotion means that blocks are promoted only at eviction time, i.e., minimal effort is put to retain hot blocks. Quick demotion denotes that the most accessed blocks are removed quickly after they are inserted into the cache. The SIEVE requires only a **FIFO** queue and a pointer (referred to as the "hand") to rapidly identify and remove unpopular blocks from the cache. In comparison to the aforementioned algorithms, SIEVE is an algorithm that is simpler than LRU and provides outstanding cache efficiency and scalability.

2.3 Unfairness of Caching Algorithms

Despite the caching algorithms are continually optimized based on the workload properties and adapted to diversified types of workloads, we observe that their limitations exist in scenarios where multiple workloads share caches.

Table 1 presents a comprehensive range of real-world workload sets, collected from a variety of functions within data centers and provided by Microsoft Research Cambridge (MSR) [35]. The block-level workload traces were obtained by means of monitoring 36 volumes over the course of one week in an enterprise data center, encompassing 179 disks across 13 servers. We selected these traces due to the implementation of the write off-loading technique, which allows write operations to be temporarily redirected to alternative storage devices. This results in fewer disk spin-ups, leading to energy conservation. Given the characteristics of write off-loading, we posit that these workload traces are suitable for cache sharing, as the technique effectively manages I/O loads, minimizes unnecessary disk activity, and maintains data consistency. Consequently, we have selected a subset of workload traces, encompassing all the different function types, to conduct the experiments and observations on cache sharing across different workloads.

Figure 2 illustrates the performance of various cache replacement algorithms when sharing a cache between the *Rsrch*_0 and *Prxy*_0 workloads. The experiment evaluates how changing the access-rate ratio magnitudes between the specified workloads affects key metrics, including hit rate, average cache usage, harmonic fairness, and maxloss. This leads us to a critical observation, which we present below.

Observation 1. *Workload access rates can impact algorithms, leading to the creation or aggravation of unfairness.*

Table 1. Microsoft Research Cambridge Workload Description

Name	Function	Footprint
Hm_0	Hardware monitoring	193 MB
Mds_0	Media server	61 MB
Prn_0	Print server	279 MB
Proj_0	Project directories	223 MB
Prxy_0	Firewall/web proxy	629 MB
Rsrch_0	Research projects	75 MB
Src1_0	Source control	2 GB
Src2_0	Source control	79 MB
Stg_0	Web staging	101 MB
Ts_0	Terminal server	89 MB
Usr_0	User home directories	113 MB
Wdev_0	Test web server	58 MB
Web_0	Web/SQL server	103 MB

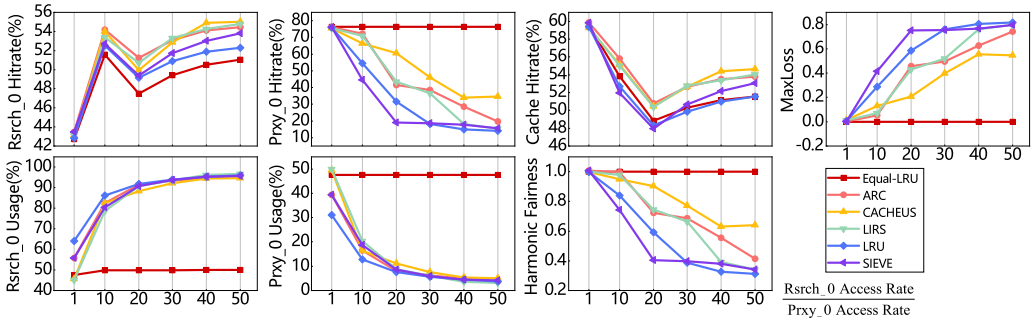


Fig. 2. Performance comparison of different cache replacement algorithms under varying access-rate ratio magnitudes in a shared cache environment. The workloads used are *Rsrch*_0 and *Prxy*_0 from the MSR trace collection. The *x*-axis represents the access-rate ratio magnitude between the two workloads, ranging from 1 to 50. The figures show several performance metrics, including hit rate, average cache usage, harmonic fairness, and maxloss.

Our experimental results reveal that workload access rates significantly affect the performance of cache replacement algorithms. With the exception of the EQ-LRU algorithm, which equally divides cache resources between workloads, all other algorithms exhibited a noticeable trend: when the access rates of *Rsrch*_0 and *Prxy*_0 are initially equal, *Rsrch*_0's cache hit rate is lower than *Prxy*_0's (as observed from the vertical axis in the graphs). However, as the access-rate ratio magnitude increases, *Rsrch*_0's hit rate improves while *Prxy*_0's hit rate decreases. This tendency is further evidenced by the harmonic fairness and maxloss metrics, wherein as the access-rate ratio magnitude rises, the harmonic fairness metric typically declines, while maxloss rises for all algorithms except for EQ-LRU.

The root cause of this phenomenon can be traced to resource usage. As shown in the Figure 2, increasing the access rate of *Rsrch*_0 results in a corresponding increase in its cache resource allocation. However, this comes at the expense of *Prxy*_0, whose cache resources diminish. With the exception of EQ-LRU, the resource allocation for *Prxy*_0 consistently declines as the access-rate ratio magnitude increases, leading to a reduction in its hit rate. This demonstrates that as workload

access rates become more disproportionate, the cache algorithms cannot maintain fairness in resource distribution.

Considering these observations, we conclude that cache replacement algorithms must effectively address the asymmetry in workload access rates to prevent unfair resource allocation, sometimes referred to as "cache cheating." To streamline the presentation and analysis, we primarily focus on overall cache hit rates, harmonic fairness, and maxloss metrics in subsequent sections, rather than detailing metrics for each workload. These selected metrics provide sufficient insight into the fairness and performance of the algorithms. Specifically, in cases where only two workloads share the cache, we omit maxloss due to its inverse correlation with harmonic fairness. However, in scenarios involving more than two workloads, we present both metrics to provide a comprehensive view of performance and fairness.

2.4 Integration Fairness and Cache Hit Efficiency

This subsection presents the rationale for integrating fairness and cache hit efficiency into the algorithm design when co-located workloads share the cache at the granularity of blocks. Firstly, we provide a proof sketch to address the problem of workload-based cache resource allocation, which is a necessary step in improving cache hit efficiency.

Intuition. When multiple workloads share caches at the block level, the cache replacement algorithm design must incorporate workload-based cache resource allocation to achieve optimal efficiency. (This implies that when multiple workloads are cached and the granularity of workload-based resource allocation is identical to that of the cache replacement algorithm; it is no longer feasible to devise the cache replacement algorithm on the basis of a single workload with a view to enhancing the efficiency of cache hits.)

Discussion. Let us posit the existence of a cache replacement algorithm, designated X, which is not required to consider the problem of workload-based cache resource allocation, and which achieves the maximum number of cache hits. In the following, we consider a simple scenario in which a block-based cache is shared by two different workloads, each of which employs the X algorithm. If the decrease in the number of possible cache hits due to a decrease in the allocated resources for one workload is not equal to the increase in the number of possible cache hits due to a corresponding increase in the allocated resources for another workload, then there exist a variety of workload-based cache resource allocation schemes that correspond to different numbers of cache hits. It thus follows that the X algorithm for block-based cache design is not the algorithm with the maximum number of cache hits, since it fails to solve the optimal workload-based cache resource allocation problem. In fact, it is difficult to guarantee that a decrease in the number of cache hits for one workload is equal to an increase in the number of cache hits for another workload due to changes in resource allocation between the two workloads. This is because the relationship between the number of cache hits and the size of the cache is not uniform and simply linear, even when considering a single workload [49].

In accordance with Intuition and Equation (2), it can be ascertained that the problem of workload-based resource allocation has an impact on both cache hit efficiency and workload fairness. Consequently, the algorithmic design proposed in this article seeks to integrate considerations of cache hit efficiency and fairness when co-located workloads share the cache at the granularity of a block.

2.5 The Problem and Challenges

Unlike existing cache replacement algorithms that primarily focus on maximizing cache hit efficiency, our algorithmic design seeks to address three critical objectives. First, we aim to integrate cache replacement and resource allocation processes at the block-granularity level in shared caches. Second, we strive to enhance both the cache hit efficiency and the fairness of resource allocation

among workloads in shared caches, as established as both necessary and feasible in Section 2.4. Third, we endeavor to improve the adaptability of the algorithm to workloads with varying access rate ratios in shared caches. Addressing these objectives requires overcoming the following four key challenges.

— **Cache Hit Efficiency.** The algorithm must ensure its cache hit performance surpasses or at least matches that of existing state-of-the-art cache algorithms, especially in multi-workload shared scenarios.
— **Fairness and Anti-cheating.** The algorithm must adapt to workloads with arbitrary access rate ratios in shared caches, ensuring fair resource distribution and preventing potential cheating behaviors.
— **Scalability.** The algorithm ought to support cache sharing for an arbitrary number of workloads and minimize the overhead.
— **Compatibility.** The algorithm should maintain consistent performance with existing cache replacement algorithms when there is only one workload.

3 Efficient and Fair Replacement Cache

This section presents two algorithms—EFRC-I and EFRC-II—designed to improve cache hit efficiency and fairness in shared cache environments. EFRC-I follows a straightforward approach inspired by algorithms like LRU, focusing on single features such as recency or frequency, ensuring a high cost-effectiveness. In contrast, EFRC-II integrates both features, thereby introducing increased complexity with the concomitant potential for enhanced performance. This added complexity comes with higher computational costs, representing a tradeoff between efficiency and overhead.

The design of both algorithms emphasizes flexibility and scalability, making it easier to incorporate additional features or factors in future improvements. Each algorithm will be explored in its respective subsection, covering its design, flow, adjustment strategies, and analysis to provide a comprehensive understanding.

3.1 The EFRC-I Algorithm

3.1.1 Architecture Design. The EFRC-I algorithm is designed for a cache-sharing environment with N workloads sharing a cache of size C. The algorithm's structure is illustrated in Figure 3. For the complete set of symbols and notation used in both EFRC-I and EFRC-II algorithms, refer to Table 2.

To ensure efficient utilization and fair resource distribution across workloads, the EFRC-I algorithm adheres to the following rules.

A.1 $\sum_{i=0}^{N-1} |T_i| = C, i \in \{0, 1, \cdots, N-1\}$.
A.2 $|B_0| = |B_1| \cdots = |B_i|, i \in \{0, 1, \cdots, N-1\}$.
A.3 $\sum ps = C$ and $ps[i] \geq 0, i \in \{0, 1, \cdots, N-1\}$.

(1) A.1: The sum of the resident queue sizes across all workloads must equal the total cache size C. Each workload has a resident queue, a designated queue storing resident blocks—data blocks that are currently used in the cache. This rule guarantees that the entire cache capacity is utilized without exceeding its limit, with resident blocks representing each workload's in-cache data.
(2) A.2: The history or metadata queue for each workload must maintain approximately equal size. This design choice allows the algorithm to gauge hit rates for each workload under a theoretically fair resource distribution. Although the effective resident queue sizes for workloads may vary dynamically, maintaining equal-sized history queues ensures that each workload has a similar opportunity to acquire the resident block resource. History queues

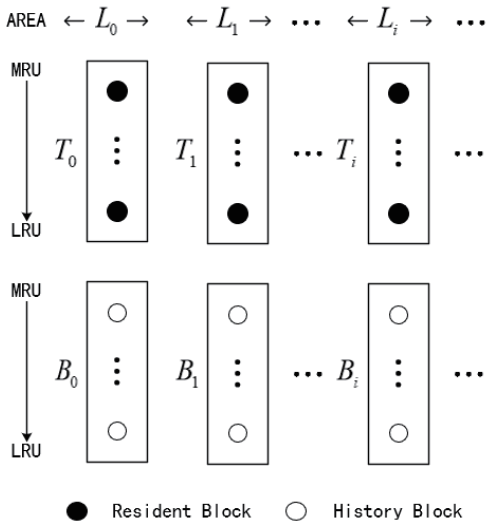


Fig. 3. General structure of the EFRC-I algorithm. The cache is partitioned into N regions, each of which is utilized by a workload represented by $L_i (i \in \{0, 1, \cdots, N-1\})$. The cache contains two types of data: resident blocks (which store block data) and history blocks (which only store metadata, such as block addressing). Each workload is assigned a resident queue T_i and a history queue B_i. The T_i and B_i queues refer to the top and bottom of L_i. The resident queue is used to store resident blocks. The history queue, also referred to as the ghost cache, is employed to register history blocks that have been evicted from the corresponding resident queue.

primarily store history or metadata blocks, containing address and auxiliary information that consume minimal space compared to resident blocks.

(3) A.3: The parameter $ps = [p_0, p_1, \ldots, p_i, \ldots, p_{N-1}]$ is a set of tunable parameters representing the target size of the cache resident queue vector $[|T_0|, |T_1|, \ldots, |T_i|, \ldots, |T_{N-1}|]$, where $ps[i]$ denotes the target size of the resident queue T_i for workload i. With N elements in ps (one for each workload), the vector's sum must satisfy A.1 by equaling C. The algorithm constantly and dynamically self-corrects the size of parameter ps, thus adjusting the size of each workload resident queue in the cache. This dynamic adjustment process for ps is further detailed in the **Adjustment Strategy** subsection.

3.1.2 Algorithm Procedure. The complete process of the EFRC-I algorithm is presented in Algorithms 1 and 2, each addressing distinct aspects of cache management. Below, we outline the roles of each algorithm and include additional details not explicitly covered in the pseudocode.

Algorithm 1 manages cache hits and misses for data blocks associated with each workload. The process is divided into three parts: *Input*, *Initialization*, and *Block Request Handling* for specific workloads.

(1) Input: We specify the structure of the data stream sequence as $x_1^0, x_2^0, x_3^{N-1}, x_4^1, \ldots, x_t^i, \ldots$, which allows for varied request orders and frequencies from different workloads. Definitions for all other symbols are provided in Table 2.

(2) Initialization: We introduce the equal-share parameter ES as an approximation of equal resource distribution across workloads, addressing cases where C may not divide evenly by

Table 2. Notation and Definitions

Symbol	Description
C	Size of the cache space.
N	Number of workloads sharing the cache.
L	Cache region associated with a specific workload or data access feature, represented as L_i or L_i^j, where $i \in \{0, 1, \ldots, N-1\}$ signifies the workload identifier, and $j \in \{0, 1\}$ denotes the data access feature identifier.
T	Resident space for storing resident blocks. Each resident block holds both block data and auxiliary information, specifically a visited bit in EFRC-I to facilitate SIEVE-based eviction, where T_i represents the resident space or queue allocated to a specific workload. In EFRC-II, represented as T_i^j, eviction uses the LRU policy, and no visited bit is utilized.
B	History space, also known as ghost cache, is used to store history blocks, which hold metadata, specifically block addresses and auxiliary information. In this context, B_i or B_i^j represents the space or queue allocated for a specific workload or access feature.
ES	A one-dimensional vector representing approximately equal resident resource allocation among N workloads.
ps	Target resource size for resident queues. In EFRC-I, it is represented as a one-dimensional vector $ps = [p_0, p_1, \ldots, p_{N-1}]$, a set of tunable parameters indicating the target size of the resident queue vector $[\lvert T_0\rvert, \lvert T_1\rvert, \ldots, \lvert T_{N-1}\rvert]$. In EFRC-II, it appears as a two-dimensional vector $ps = [[p_0^0, \ldots, p_{N-1}^0], [p_0^1, \ldots, p_{N-1}^1]]$ to gauge the target size of the cache resident queue vector $[[\lvert T_0^0\rvert, \ldots, \lvert T_{N-1}^0\rvert], [\lvert T_0^1\rvert, \ldots, \lvert T_{N-1}^1\rvert]]$.
$hands$	A vector of pointers, each tracking the next eviction candidate for a workload, moving from LRU to MRU in the resident queue. Used only in EFRC-I to facilitate resident block eviction via the SIEVE algorithm.
ξ	A buffer threshold triggering resource adjustment only when a workload's allocation deviates significantly from its average allocation, minimizing computation overhead and strategy switching.
$meta_w$	Coefficient determining the size of history queues. When $meta_w$ equals 1, the size limit matches the value of ES.

N and ensuring that the difference in allocation across workloads is at most one block. This principle applies not only to the initialization of resident queues (where $ps = ES$), but also to the history queues, as enforced by the pruning operations in Algorithm 2. The use of ES also permits flexibility for alternative weighted resource allocations, though such scenarios are outside the present scope.

(3) Block Request Handling: The algorithm differentiates three cases based on the structure: when a requested block is in the resident queue T, in the history queue B, or results in a miss, with each scenario managed according to the detailed steps in Algorithm 1.

Algorithm 2, implemented as the **Replace** function within itself, determines which resident block to evict when a cache miss occurs, and the resident space is fully utilized. Algorithm 2 consists of three main parts.

(1) The first part is the **GetReplaceWorkload** function, which identifies the workload from which a resident block will be evicted. This decision is based on solving a mathematical representation of the problem, as expressed in Equation (4). If multiple workloads satisfy the criteria, one is selected either randomly or by choosing the workload with the maximum or minimum $|T_i|$ for $i \in \{0, 1, \ldots, N-1\}$. In our experiments, the algorithm prioritizes the first

ALGORITHM 1: EFRC-I

INPUT: The request stream $x_1^0, x_2^0, x_3^{N-1}, x_4^1, \cdots, x_t^i, \cdots$, doubly-linked queues T_i, B_i ($i \in \{0, 1, \cdots, N-1\}$), cache size C, vector pointers *hands*, equal-share resource allocation *ES*, target resource size *ps*, buffer threshold ξ, and metadata coefficient *meta_w*.

INITIALIZATION: Set the initial value of parameter *ps* is *ES* and the queues T_i,B_i ($i \in \{0, 1, \cdots, N-1\}$) to empty.

BLOCK REQUEST HANDLING: For every request $t \geq 1$, workload $i \geq 0$, and any x_t^i, one and only one of the following three cases must occur.

Case I: x_t^i in T_i.

// A cache hit has occurred in EFRC-I.

$x_t^i.visited := 1$

Case II: x_t^i in B_i.

// A cache miss has occurred in EFRC-I.

if $|T_i| < C//N - \xi$ **then**

FairTune(i)

else if $|T_i| > C//N + \xi$ **then**

EffTune(i)

end if

Replace()

Move x_t^i to the MRU position in T_i.

Case III: x_t^i is not in $T_i \cup B_i$.

// A cache miss has occurred in EFRC-I.

if the cache is full, that is $\sum_{i=0}^{N-1} |T_i|$ = C **then**

Replace()

end if

$x_t^i.visited := 0$

Insert x_t^i as the MRU block in T_i.

ALGORITHM 2: Replace()

1: // u := GetReplaceWorkload().
2: $u := 0$
3: **for** $i := 0$ **to** $(N-1)$ **do**
4: **if** $|T_i| - ps[i] > |T_u| - ps[u]$ **then**
5: $u := i$
6: // Lines 7-17 remove a resident block using SIEVE's turn-key eviction policy.
7: $o := hands[u]$
8: **if** o is NULL **then**
9: o := LRU position of T_i
10: **while** $o.visited = 1$ **do**
11: $o.visited := 0$
12: // The direction of next is from LRU to MRU position.
13: $o := o.next$
14: **if** o is NULL **then**
15: $o :=$ LRU position of T_i
16: $hands[u] := o.next$
17: Move the block corresponding to o in the T_u to the MRU position in B_u.
18: // A pruning operation to ensure that rule A.2 is satisfied.
19: **if** $|B_u| > meta_weight * ES[u]$ **then**
20: Delete the LRU block in B_u.

ALGORITHM 3: EffTune(u)

1: **for** $i := 1$ to $(N-1)$ and $i \,!= u$ **do**
2: $\quad \delta := \mathbf{min}(ps[i],\ \mathbf{max}(1, |T_u| // \mathbf{max}(1, |T_i|)))$.
3: $\quad ps[i] := ps[i] - \delta$.
4: $\quad ps[u] := ps[u] + \delta$.

qualifying workload to ensure consistent performance.

$$\begin{aligned} & \underset{i}{argmax}(|T_i| - ps[i]) \\ s.t. \quad & |T_i| > 0, i \in \{0, 1, \cdots, N-1\} \end{aligned} . \tag{4}$$

(2) The second part, corresponding to lines 6–17 in the pseudocode, selects the specific block within the chosen workload's resident queue to be evicted, using SIEVE's turn-key eviction policy. In order to facilitate understanding, we first provide a brief overview of the principles behind the SIEVE algorithm. SIEVE employs a visited bit and a hand pointer to manage cache replacement. The visited bit is set to 0 for newly inserted blocks in SIEVE. When a cache hit occurs, the visited bit of the accessed block is set to 1, and if already set, no further action is taken. During a cache miss, the algorithm inspects the block indicated by the hand pointer. If the block's visited bit is set to 1, it is reset to 0, and the hand advances to the next block. This continues until a block with a visited bit of 0 is found, which is then chosen for eviction. The parameter *hands*, defined in Table 2, tracks pointers to the next eviction candidates, facilitating block eviction by moving from LRU to MRU within each resident queue.
(3) The third part of Algorithm 2 is a pruning operation to ensure compliance with rule A.2. This operation relies on the *meta_w* parameter, also defined in Table 2, which determines the size of each history queue. As this parameter influences metadata overhead, its effect on the algorithm will be assessed in the experimental Section 4.5.2.

3.1.3 Adjustment Strategy. In order to guarantee both hit efficiency and fairness in cache usage, the EFRC-I algorithm employs two learning-based adjustment strategies to dynamically tune the target parameter ps for each resident queue. These strategies are triggered through a perceptron mechanism, specifically a ghost cache B_i, which activates different adjustments based on the size of the corresponding resident space T_i. As outlined in Algorithm 1, the EFRC-I algorithm invokes the function **EffTune** to adjust the parameter ps when $|T_i| > C//N + \xi$ and the function **FairTune** when $|T_i| < C//N - \xi$. The parameter ξ denotes a hysteresis threshold parameter, designed to reduce computational overhead and prevent excessive strategy switching. The symbol ξ is defined in Table 2, and its impact on algorithm performance is discussed further in the experimental Section 4.5.1. Below, we explain each of these adjustment strategies in detail.

Efficiency Adjustment Strategy (EffTune): Algorithm 3 illustrates the **EffTune** function, which seeks to optimize hit efficiency. This adjustment draws on the "investment" intuition from the ARC algorithm, wherein a hit in B_u prompts an increase in the resident space T_u for that workload to enhance cache efficiency. Specifically, since p_u represents the target size of T_u, the algorithm expands p_u when there is a hit in B_u. To maintain the invariant property A.3 of the ps vector, the algorithm reduces p_i for each of the other workloads i.

This adjustment raises several important questions: the exact increase for p_u in the hit workload u, the selection of workloads from which to reduce target sizes, and the specific reductions to apply. When a hit occurs in B_u, as shown in Algorithm 3, the algorithm iterates through the other workloads i, incrementally reducing each p_i while increasing p_u step by step. The learning rate δ denotes the magnitude of each adjustment. Unlike the ARC algorithm's rate, which depends on ghost cache size, δ in EFRC-I depends on the resident queue size of the hit workload and the

ALGORITHM 4: FairTune(u)

```
1: δ := ES[u] − ps[u];
2: if δ > 0 then
3:     victims := [i := 0 → (N − 1) and i != u]
4:     while |victims| > 0 and δ > 0 do
5:         δ_v := max(1, δ // |victims|)
6:         for i in victims do
7:             δ_min := min(ps[i], δ_v)
8:             if δ_min = ps[i] then
9:                 victims.remove(i)
10:            ps[i] := ps[i] − δ_min
11:            ps[u] := ps[u] + δ_min
12:            δ := δ − δ_min
```

remaining workloads. If T_u is no greater than T_i, p_u is incremented by 1; otherwise, p_u is boosted by $|T_u|//|T_i|$. The learning rate δ is finally capped by p_i, allowing the size of T_u to govern the increment's magnitude.

This efficiency adjustment strategy effectively reallocates cache resources to maximize hit potential for high-performing workloads. By continuously and adaptively updating the target sizes of each workload's resident queue, the algorithm dynamically adapts to workload access characteristics.

Fairness Adjustment Strategy (FairTune): Algorithm 4 demonstrates the **FairTune** function, which addresses fairness. Although each workload's actual resident resources may vary during execution, the size of each workload's history queue is determined by the parameter ES, representing an approximately equal distribution of resources. As a result, regardless of real-time allocation shifts, the history queue consistently reflects each workload's fair share based on an ideal distribution.

When a workload's history queue registers a hit and its resident stack size drops below $C/N - \xi$, the algorithm restores resources to the workload to mitigate hit loss and improve fairness. This restoration is achieved by resetting the target parameter $ps[i]$ for that workload to its fair allocation $ES[i]$.

As shown in Algorithm 4, **FairTune** addresses two key aspects of fairness adjustment. First, if the algorithm detects that a workload's hit loss could compromise its fairness, it increases $ps[i]$ with an increment defined by $\delta := ES[u] - ps[u]$. Second, the increased resources for this workload are distributed among all other workloads. Since some workloads may not support this redistribution, a similar Max-min allocation strategy is applied to guide the adjustment. This **FairTune** function provides a simple yet effective method for fair resource allocation. Additional fairness considerations, such as weighted fairness, are beyond the scope of this study [9].

3.2 The EFRC-II Algorithm

EFRC-II represents a hybrid strategy that integrates both recency and frequency features to optimize cache management. The development of this algorithm was driven by two overarching objectives. Firstly, the objective is to incorporate multiple workload and block access characteristics into a unified framework at the cache block granularity level. Secondly, the objective of this unified framework is to ascertain the latent performance potential of the algorithm.

3.2.1 Architecture Design. Figure 4 illustrates the general architecture of the EFRC-II algorithm, and the relevant symbol definitions that are listed in Table 2. Similarly, the EFRC-II algorithm aligns with the rules stipulated below.

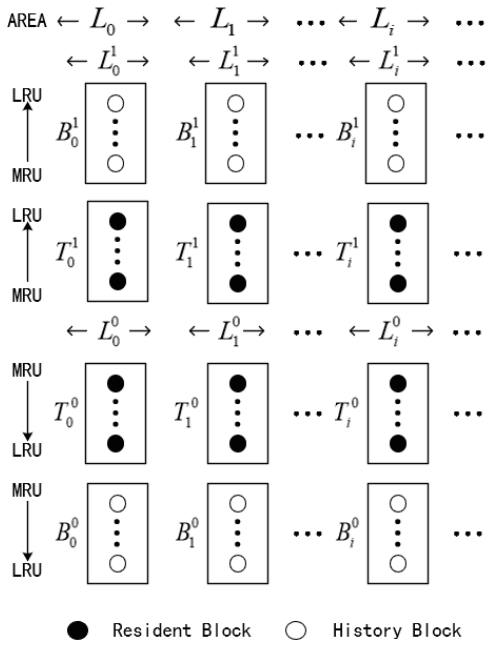


Fig. 4. General structure of the EFRC-II algorithm. Different from the EFRC-I structure, each workload region L_i is further divided into L_i^0 and L_i^1 according to the frequency of data access, where superscript 0 represents data accessed only once recently and superscript 1 represents data accessed multiple times recently. L_i^0 and L_i^1 have the same structure, both containing a resident queue and a history queue. Specifically, L_i^0 holds data that have been recently accessed only once, its resident queue and history queue are referred to as T_i^0 and B_i^0, respectively (T_i^0/B_i^0 refers to the top/bottom of L_i^0). L_i^1 holds data that has been accessed multiple times recently, its resident queue and history queue are referred to as T_i^1 and B_i^1, respectively (T_i^1/B_i^1 refers to the top/bottom of L_i^1).

B.1 $\sum_{j=0}^{1}\sum_{i=0}^{N-1}|T_i^j| = C, i \in \{0, 1, \cdots, N-1\}$.
B.2 Let $|B_0| = |B_1| \cdots = |B_i|$ and $|B_i^0| = |B_i^1|$,
where $|B_i| = |B_i^0| + |B_i^0|$ and $i \in \{0, 1, \cdots, N-1\}$.
B.3 Let $\sum\sum ps = C$ and $ps[j][i] \geq 0$,
where $i \in \{0, 1, \cdots, N-1\}$ $j \in \{0, 1\}$.

Following rules B.1-3, and compared with rules A.1-3, the EFRC-II algorithm extends the range of cache block access features, expanding j to take values in $\{0, 1\}$. Despite this extension, the rules governing resident block resources, history block resources, and target resource parameters remain consistent with those of EFRC-I. The only additional explanation required is for the constraint $|B_i^0| = |B_i^1|$ in rule B.2. Similar to the rule $|B_0| = |B_1| = \cdots = |B_i|$, this constraint not only provides each workload the opportunity to fairly obtain resident resources, but also ensures that blocks within a single workload, with different access characteristics, have an equal opportunity to compete for these resources.

3.2.2 Comparison of EFRC-II to EFRC-I. Algorithm 5 presents the complete process of the EFRC-II algorithm, while Algorithms 6 and 7 describe its adjustment strategies. Building upon

ALGORITHM 5: EFRC-II

INPUT: The request stream like $x_1^0, x_2^0, x_3^{N-1}, x_4^1, \cdots, x_t^i, \cdots$, doubly-linked queues T_i^j ($i \in \{0, 1, \cdots, N-1\}, j \in \{0, 1\}$), cache size C, equal-share resource allocation ES, target resource size ps, buffer threshold ξ, and metadata coefficient $meta_w$.

INITIALIZATION: Set $ps = [[0] \times N, ES]$ and the queues T_i^j,B_i^j ($i \in \{0, 1, \cdots, N-1\}, j \in \{0, 1\}$) to empty.

BLOCK REQUEST HANDLING: For every request $t \geq 1$, workload $i \geq 0$ and any x_t^i, one and only one of the following three cases must occur.

Case I: x_t^i in $T_i^0 \cup T_i^1$.
// A cache hit has occurred in EFRC-II.
Move x_t^i to the MRU position in T_i^1.

Case II: x_t^i in $B_i^0 \cup B_i^1$.
// A cache miss has occurred in EFRC-II.
// If x_t^i in B_i^0, j is 0; Else x_t^i in B_i^1, j is 1.
if $(|T_i^0| + |T_i^1|) < C//N - \xi$ **then**
    FairTune(j, i)
else if $(|T_i^0| + |T_i^1|) > C//N + \xi$ **then**
    EffTune(j, i)
end if
Replace()
Move x_t^i to the MRU position in T_i^1.

Case III: x_t^i is not in $T_i^0 \cup B_i^0 \cup T_i^1 \cup B_i^1$.
// A cache miss has occurred in EFRC-II.
If the cache is full, that is $\sum_{i=0}^{N-1}(|T_i^0| + |T_i^1|)$ = C **then**
    Replace()
end if
Finally, insert x_t^i as the MRU page in T_i^0.

Function Replace()
1: $f, u :=$ **GetReplaceWorkload**()
2: // Remove a resident block using LRU eviction policy.
3: Move the LRU block of T_u^f to the MRU position in B_u^f.
4: // A pruning operation to ensure that rule B.2 is satisfied.
5: **if** $|B_u^f| > meta_w * (ES[u]/2)$ **then**
6: Delete the LRU block in B_u^f.

ALGORITHM 6: EffTune(f, u)

1: **for** $j := 0 \rightarrow 1$ and $j != f$ **do**
2: **for** $i := 0 \rightarrow (N-1)$ and $i != u$ **do**
3: $\delta := \mathbf{min}(ps[j][i], \mathbf{max}(1, |T_u^f| // \mathbf{max}(1, |T_i^j|)))$.
4: $ps[j][i] := ps[j][i] - \delta$.
5: $ps[f][u] := ps[f][u] + \delta$.

EFRC-I, the EFRC-II algorithm introduces both recency and frequency features in classifying data block access patterns. As a result, the design of the two algorithms exhibits strong consistency, so that understanding the detailed design of EFRC-I makes the pseudocode of EFRC-II easy to follow. Therefore, rather than repeating the full design of EFRC-II, we focus on highlighting the key differences and similarities between EFRC-I and EFRC-II, particularly in terms of their modifications and objectives. We summarize the detailed differences between the EFRC-I and EFRC-II algorithms by answering three key questions.

ALGORITHM 7: FairTune(f, u)

```
δ := ES[u] − ps[0][u] − ps[1][u];
if δ > 0 then
    victims := [(j, i) := (0, 0) → (1, N − 1) and i != u]
    while |victims| > 0 and δ > 0 do
        δ_v := max(1, δ // |victims|)
        for (j, i) in victims do
            δ_min := min(ps[j][i], δ_v)
            if δ_min = ps[j][i] then
                victims.remove((j, i))
            ps[j][i] := ps[j][i] − δ_min
            ps[f][u] := ps[f][u] + δ_min
            δ := δ − δ_min
```

How does the extension of block access features affect the EFRC-II algorithm? The extension introduces four significant effects:

— The resident queue T_i for each workload is further divided into T_i^0 and T_i^1 queues based on block access frequency, leading to the formation of corresponding B_i^0 and B_i^1 queues. As a result, Algorithm 5 must consider more scenarios when the cache handles requests from various workloads. However, despite the distinction between T_i^0 and T_i^1, blocks that are re-accessed items in either queue are migrated to the MRU position in T_i^1. Therefore, the handling of hits for these two queues can be unified. A similar process occurs for the hit blocks in B_i^0 and B_i^1 queues for the same reason.

— The target resource parameter ps in Algorithm 5 is represented as a two-dimensional vector, as detailed in Table 2.

— When the cache reaches full resident capacity and a resident block needs to be evicted, the algorithm needs to determine both the workload, and the specific block access feature associated with the resident queue. Therefore, the **GetReplaceWorkload** function in Algorithm 5 returns two values. Solving this selection process is similar to solving a simple mathematical problem, as shown in mathematical form Equation (5).

— In the pruning operation within the ghost cache, it is necessary to ensure that B_i^0 and B_i^1 meet the conditions specified by rule B.2 in the EFRC-II algorithm.

$$\begin{aligned} & \underset{i,j}{argmax}(|T_i^j| - ps[j][i]) \\ s.t. \quad & |T_i^j| > 0, i \in \{0, 1, \cdots, N-1\}\, j \in \{0, 1\} \end{aligned} . \tag{5}$$

Why does EFRC-II use a different eviction strategy compared to EFRC-I? EFRC-I and EFRC-II employ different resident block eviction strategies to balance and optimize algorithmic performance and computational overhead, aligned with their architectural designs. In the EFRC-I algorithm, each workload has only one resident queue. The SIEVE algorithm, which was demonstrated in its original publication to be simpler and more efficient than LRU for single-queue caching, was chosen to reduce the time cost of moving blocks within the queue while preserving hit performance. In contrast, EFRC-II requires two resident queues per workload, where inter-queue block migration is necessary. Continuing to use the visited bit and hand pointer mechanisms in this scenario would add considerable complexity to the algorithm. Thus, LRU was selected for EFRC-II to manage block eviction more straightforwardly.

Why do EFRC-I and EFRC-II improve both cache hit efficiency and fairness? Both EFRC-II and EFRC-II achieve a balance between cache hit efficiency and fairness by continuously

adjusting the target parameter *ps* for resident resources. In both algorithms, the priority of cache hit efficiency is placed above that of fairness, meaning that the algorithm does not always ensure an equal allocation of resident resources to each workload or data type. Instead, it strategically sacrifices some resources from workloads or data types with lower cache efficiency, reallocating them to those with higher cache efficiency to enhance overall hit performance. Nevertheless, it is essential to recognize that fairness is also a critical factor in sustaining high cache hit efficiency over time. As workloads may exhibit dynamic and unpredictable access patterns or rates, the assumption of consistently high cache efficiency for specific workloads does not hold. Therefore, enhancing fairness and adaptability is a fundamental objective of the algorithm's design.

Informed by and further developing the ghost cache or history resource concept, this design extends beyond traditional usage. In the ARC algorithm, ghost cache design primarily aims to boost cache efficiency; however, in EFRC-I and EFRC-II, we reinterpret ghost cache as an opportunity for fair resource competition. While the algorithm may not maintain equal resident resources for each workload or data type at all times, it does strive to ensure fair access to history resources—providing each workload or data type with an equal opportunity. When the algorithm detects that a workload or data type has incurred a substantial hit loss that compromises its basic fairness guarantee, it reallocates the necessary resident resources to restore fairness. This ghost cache design enables each workload or data type to compete for more than its fair share of resident resources, without permanently losing its right to regain fair resource allocation.

The underlying "investment" philosophy remains consistent in both EFRC-I and EFRC-II. Algorithms 3 and 6 represent the respective efficiency adjustment strategies for EFRC-I and EFRC-II, differing primarily in their focus: EFRC-I reallocates cache resources based on the best-performing workload, while EFRC-I further refines this approach by investing resources in the best-performing data type. Similarly, Algorithms 4 and 7 implement the fairness adjustment strategies, with one ensuring workload-based fairness and the other further guaranteeing fairness based on data type.

4 Evaluation

We evaluate the EFRC class algorithms by answering the following questions:

- How do our algorithms perform in terms of cache hit performance and fairness in scenarios with shared cache workloads that exhibit imbalanced access rate ratios? (Section 4.2)
- By comparing the real-time performance and resource allocation processes of different algorithms, how can we understand and analyze why our algorithms can enhance cache hit performance, particularly in terms of fairness? (Section 4.3)
- How do our algorithms perform in terms of cache hit performance and fairness across scenarios with different quantities of shared cache workloads? (Section 4.4)
- What are the parameter sensitivity and overhead of our algorithms? (Section 4.5)
- What is the effectiveness of our algorithms, and what is their impact when used in more extensive testing? (Section 4.6)

4.1 Experimental Settings

Experimental Setup. We implement a trace-driven simulator to rigorously assess the efficacy of our algorithms. The simulator is derived from the open-source CACHEUS code, but we extensively rewrote the simulator code to support multi-workload shared cache scenarios. Our experiments are conducted on a server configured with Ubuntu 18.04 LTS and a Linux kernel version 4.15. The server's architecture comprises two sockets, each accommodating 16 cores, with two threads per core. The cache hierarchy consists of 32/48 KB split L1 caches (for instructions and data, respectively), a 1,280 KB L2 cache, and a 24 MB L3 cache.

Table 3. Average Gains in Harmonic Fairness and Cache Hit Rate for Various Algorithms in Two Workload-shared Cache Configurations at Different Access-rate Ratio Magnitudes, Relative to the EQ-LRU Algorithm

Algorithm	Average relative EQ-LRU performance gain(%) at different access-rate ratio magnitudes							
	Harmonic Fairness				*Cache Hit Rate*			
	1	10	20	30	1	10	20	30
EFRC-II	3.5	4.81	4.82	4.79	2.74	7.47	8.97	10.27
EFRC-I	-2.63	-1.86	-1.42	-1.57	-0.11	-0.14	1.13	2.22
SIEVE	-0.92	-8.28	-10.1	-13.6	0.12	-0.19	0.94	1.66
CACHEUS	-7.47	-5.59	-7.88	-8.65	-3.3	2.95	4.69	7.82
LIRS	-7.76	-13.6	-16.01	-16.96	-2.88	-1.03	0.01	2.32
ARC	1.4	-0.65	-3.01	-4.72	1.09	6.57	7.68	9.41
LRU	1.97	-2.27	-7.1	-11.92	-0.64	5.07	4.61	4.2

Cache Setup. In contrast to the approach taken by CACHEUS, whereby the cache size is determined by the footprint size of the workload, our experiments set the cache size at a uniform 16 MB to facilitate the management of large-scale tests. The preliminary tests indicate that the experimental conclusions remain unaffected by variations in cache size. The configuration is an appropriate cache size, while this choice facilitates focusing the analysis on key factors such as workload type, quantities, access speed, and algorithmic impact, thus avoiding unnecessary complexity in statistical understanding. Moreover, adhering to the MSR workload's I/O request specifications, the cache is structured with a basic cache line or data block size of 512 bytes and a history block size of 8 bytes for 64-bit addressing.

Algorithms. The simulator comprises various algorithms that are utilized for performance comparison. Some of these algorithms are the original algorithms—including LIRS and CACHEUS—shared by the authors. We compare our algorithm with the previously mentioned LRU, ARC, LIRS, CACHEUS, and SIEVE algorithms.

Workloads and Simulations. Table 1 summarizes the public workload datasets gleaned from the actual production systems. We construct a caching system handling these workloads in accordance with the dataset type. I/O traces for every workload are numbered and cached at a predetermined rate. In this manner, it is possible to simulate production system scenarios with different cache sizes, multiple workloads, and various read/write rates.

4.2 Comparison of Different Access-rate Ratios

To evaluate the adaptability of our algorithm under varying access rate ratios, we tested its performance in terms of harmonic fairness and cache hit rate across all pairwise combinations of 13 MSR workloads. As shown in Figure 2, when the access-rate ratio magnitude between two workloads exceeds 40, the cache resources of the slower workload are significantly dominated by the faster workload. Hence, we limited the access-rate ratio magnitudes to 1, 10, 20, and 30. Given the 13 workloads in the MSR dataset, there are 78 unique workload pairs. These combinations were numbered from 0 to 77 and divided into two groups: combinations 0–38 and 39–77. For the first group, we set the access-rate ratios to $1:1$, $1:10$, $1:20$, and $1:30$, while for the second group, the access-rate ratios were set to $1:1$, $10:1$, $20:1$, and $30:1$. The results of these tests are presented in Figure 5. Subsequently, we combined the results across different access-rate ratio magnitudes to compute the performance gains of each algorithm relative to the EQ-LRU algorithm. The average performance gains for each algorithm are summarized in Table 3.

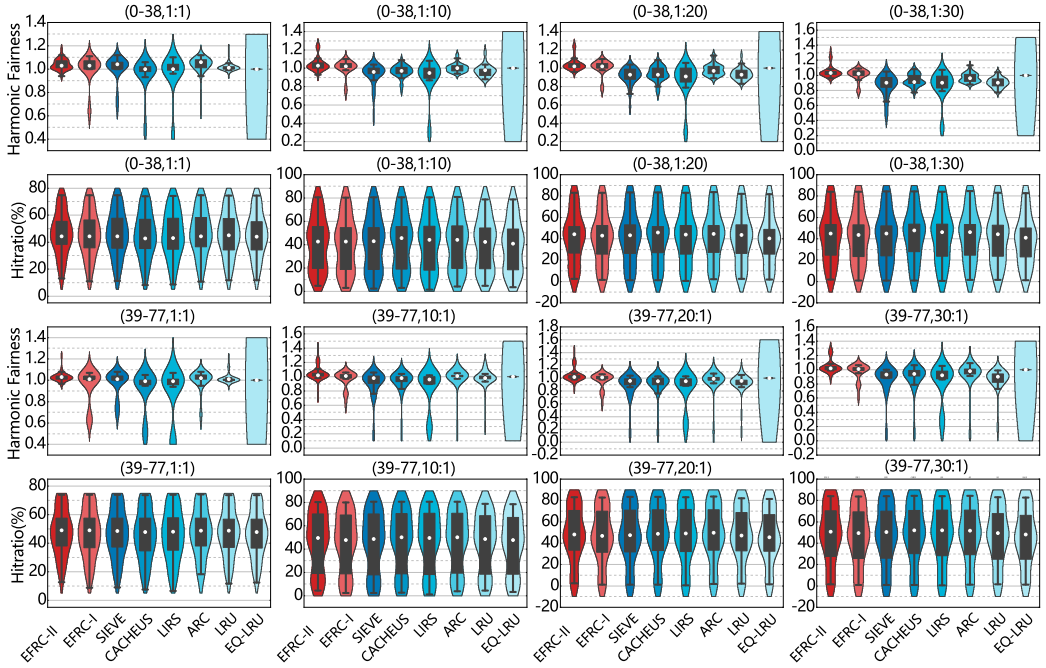
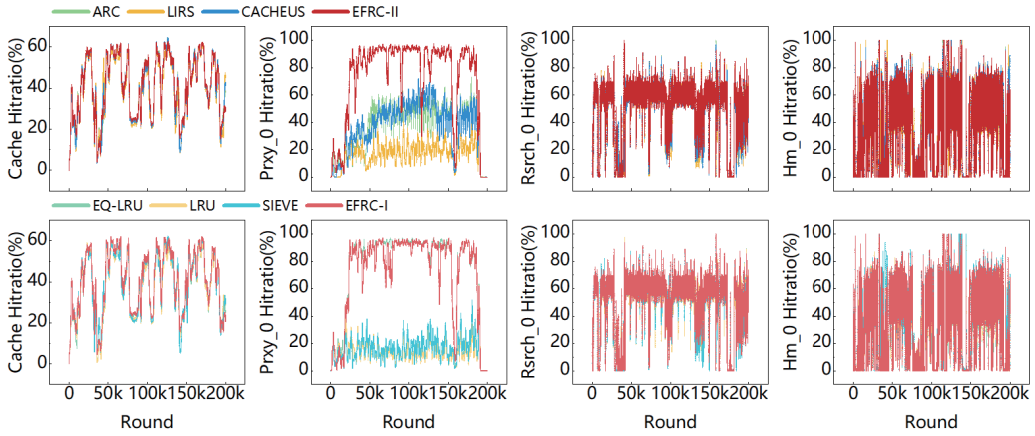


Fig. 5. Comparison of fairness and hit rate distributions for different algorithms across two workload-shared cache configurations at varied access-rate ratios. The title of the first subgraph (0-38, 1:1) signifies that the workload combination used is numbered from 0 to 38, and the two workloads within each combination access the cache at a ratio of 1 : 1.

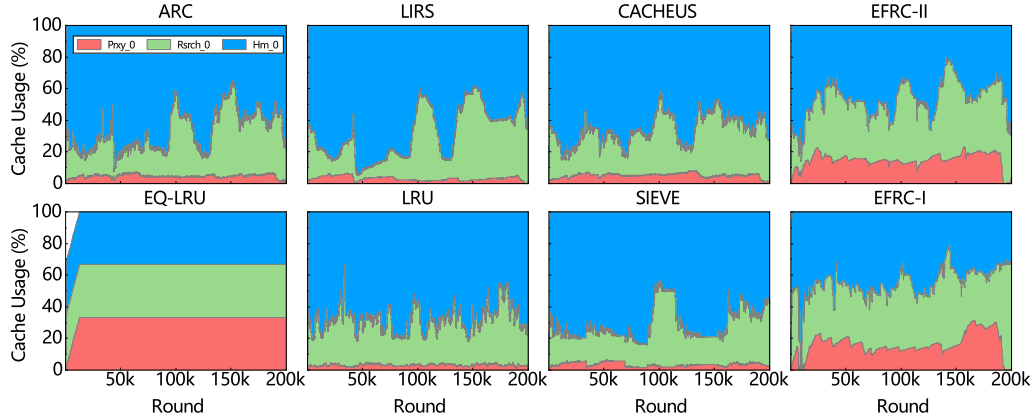
As shown in Figure 5 and Table 3, our algorithms significantly outperform others in terms of harmonic fairness and cache hit rate, with the advantages becoming more pronounced as the access rate ratio increases. The EQ-LRU algorithm serves as the baseline for assessing fairness. From Table 3, it can be observed that when the access-rate ratio magnitude exceeds 10, all algorithms except EFRC-II show fairness values lower than EQ-LRU. Although the fairness of the EFRC-I algorithm appears inferior to EQ-LRU in Table 3, this result is influenced by the workload *Src*1_0. As shown in Table 6, the cache hit rate of *Src*1_0 is less than 2%. In fact, when excluding combinations involving *Src*1_0, both EFRC-I and EFRC-II demonstrate significantly better fairness compared to other algorithms, including EQ-LRU. In terms of cache hit rate, when the access-rate ratio magnitude exceeds 10, the EFRC-I algorithm achieves better hit rates than the SIEVE algorithm. Furthermore, the EFRC-II algorithm consistently delivers the best hit rate performance among all algorithms tested.

4.3 Understanding EFRC in Real-time Process

To observe and analyze the real-time hit performance and resource allocation processes of our algorithms, we designed the experiment shown in Figure 6. In this experiment, the workloads *Prxy_0*, *Rsrch_0*, and *Hm_0* share a cache with access rate ratios of 1 : 15 : 30. This setup serves two purposes: (1) to further validate the algorithm's performance under scenarios similar to those shown in Figure 2 and (2) to introduce moderate experimental complexity without compromising the clarity of our observations. Next, we record the real-time processing results across 200,000 rounds governed by various algorithms. The size of the sliding window is set to 1% of the total number of rounds.



(a) Comparison of real-time cache hit rates and workload hit rates of different algorithms when workloads *Prxy_0*, *Rsrch_0*, and *Hm_0* share the cache at an access rate ratio of 1 : 15 : 30.



(b) Comparison of real-time cache resource allocation among shared workloads under different algorithms when workloads *Prxy_0*, *Rsrch_0*, and *Hm_0* share the cache at an access rate ratio of 1 : 15 : 30.

Fig. 6. Real-time comparison between EFRC and other state-of-the-art algorithms.

Figure 6(a) illustrates the real-time cache hit rates and workload hit rates under different algorithms, while Figure 6(b) shows the real-time proportion of cache resources allocated to each workload. Additionally, the overall hit rates and fairness metrics for this test are summarized in Table 4.

From Figure 6 and Table 4, it is evident that our algorithms fully utilize the hit potential of each workload, achieving superior real-time hit rates for all workloads compared to other algorithms. Consequently, our algorithms outperform others in both cache hit rate and fairness metrics. As shown in Table 6, the hit rate ranking for the three workloads is *Prxy_0* > *Rsrch_0* > *Hm_0*. However, due to the 1 : 15 : 30 access rate ratio, algorithms such as ARC, LIRS, CACHEUS, LRU, and SIEVE allocate minimal cache resources to *Prxy_0*, leading to significantly reduced hit rates for this workload. Only the EQ-LRU algorithm maintains an equal cache resource allocation among the three workloads throughout the process. In contrast, our EFRC-I and EFRC-II algorithms dynamically adjust cache resource allocation, ensuring appropriate distribution among the workloads. Even

Table 4. The Hit Rate and Fairness Metrics of Different Algorithms in a Shared Cache Scenario for Three Workloads *Prxy_0*, *Rsrch_0*, and *Hm_0* According to Access-rate Ratio 1 : 15 : 30

Algorithm	Hit Rate(%)				Fairness	MaxLoss
	Cache	*Prxy_0*	*Rsrch_0*	*Hm_0*		
EFRC-II	41.55	72.59	49.99	36.3	1.03	0.04
EFRC-I	40.94	72.61	49.83	35.44	1.02	0.04
SIEVE	39.62	16.45	47.39	36.51	0.46	0.78
CACHEUS	40.98	37.58	48.35	37.41	0.77	0.5
LIRS	39.95	17.05	47.74	36.82	0.47	0.77
ARC	41.36	36.38	49.03	37.68	0.76	0.52
LRU	38.72	13.71	48.3	34.76	0.4	0.82
EQ-LRU	38.79	75.51	48.59	32.67	1	0

Table 5. The Fairness, Hit Rate, and Maxloss of Different Algorithms in Shared Cache Scenarios across the MSR of 13 Workloads

Algorithm	*Harmonic Fairness*	*Cache Hit Rate(%)*	*MaxLoss*
EFRC-II	1.0	60.15	0.38
EFRC-I	0.95	60.32	0.49
SIEVE	0.84	59.08	0.58
CACHEUS	0.9	60.59	0.64
LIRS	0.86	59.98	0.66
ARC	0.9	60.85	0.64
LRU	0.82	57.89	0.46
EQ-LRU	1.0	41.67	0.0

The access rate ratio of these 13 workloads is 3 : 1 : 1 : 1 : 23 : 1 : 1 : 14 : 1 : 2 : 2 : 8 : 2.

when *Prxy_0* occasionally receives fewer resources, the algorithms effectively restore its allocation to a reasonable share in subsequent rounds.

4.4 Comparison of Different Shared Quantities

4.4.1 Scalability. To evaluate the scalability of our algorithms, we conducted an experiment where all 13 MSR workloads shared the cache, as detailed in Table 5. The access rate ratios for these workloads were randomly generated using a Python script, ensuring that the sum of the rates ranged between 50 and 80 to maintain a moderate level of complexity. As shown in Table 5, our algorithms achieve a strong balance between cache hit performance and fairness compared to other algorithms. While the harmonic fairness of the EQ-LRU algorithm is 1, its cache hit rate is significantly lower than that of other algorithms. Conversely, although ARC and CACHEUS exhibit slightly higher cache hit rates than our algorithms, their fairness metrics are noticeably inferior.

4.4.2 Compatibility. To evaluate the compatibility of our algorithms, we tested the cache hit rate of various algorithms on each MSR workload, as shown in Table 6. From the results in Table 6, it is evident that the cache hit rates of EFRC-I and SIEVE are identical. This is because EFRC-I was explicitly designed to be compatible with SIEVE in single-workload scenarios. The only difference lies in the addition of a ghost cache in EFRC-I. While EFRC-I could also be made compatible with LRU or other algorithms, SIEVE was chosen due to its simplicity and efficiency over LRU, aligning with EFRC-I's emphasis on cost-effectiveness. In contrast, the design of EFRC-II in single-workload scenarios shares some similarities with ARC, though the two algorithms differ significantly. As

Table 6. Cache Hit Rates for Different Algorithms on a Single Workload

Workload	Cache Hit Rate(%) at different algorithms						
	EFRC-II	EFRC-I	SIEVE	CACHEUS	LIRS	ARC	LRU
Hm_0	38.27	37.28	37.28	37.63	38.2	38.97	35.09
Mds_0	64.94	64.62	64.62	65.78	64.83	64.94	62.57
Prn_0	15.45	14.71	14.71	14.72	14.39	15.24	13.59
Proj_0	22.4	22.28	22.28	24.27	21.65	22.81	22.02
Prxy_0	84.2	84.21	84.21	84.27	84.06	84.43	83.1
Rsrch_0	52.1	49.79	49.79	51.08	52.2	51.64	49.31
Src1_0	1.34	0.69	0.69	0.69	0.19	1.35	1.64
Src2_0	71.27	71.09	71.09	71.48	71.95	71.53	69.05
Stg_0	46.7	44.49	44.49	46.77	46.18	46.83	45.35
Ts_0	44.18	42.34	42.34	43.5	43.73	43.18	39.1
Usr_0	30.59	27.59	27.59	31.04	29.49	30.58	27.74
Wdev_0	54.09	52.66	52.66	52.19	52.88	51.78	52.95
Web_0	9.53	9.05	9.05	9.55	9.33	9.72	9.35

shown in Table 6, EFRC-II achieves a higher cache hit rate than ARC for some workloads. The primary differences between EFRC-II and ARC can be summarized in two key points. First, the sizes of the history queues corresponding to different data access features in EFRC-II are fixed and equal, whereas ARC dynamically adjusts the sizes of its ghost cache queues during runtime. EFRC-II adopts this equal-sized history queue design to uphold the principle of opportunity fairness. Second, EFRC-II adjusts its target resource parameters based on the sizes of resident queues, whereas ARC bases its adjustments on the sizes of ghost cache queues.

4.5 Sensitivity/Overhead Analysis

4.5.1 Sensitivity of Parameter ξ. The parameter ξ, as defined in Table 2, is introduced in our algorithms to prevent frequent switching between adjustment strategies near the average resource allocation of a workload and to reduce computational overhead. In this study, ξ is expressed as $\xi = K \cdot N$, where N represents the number of workloads sharing the cache, and K is a tunable multiplier. Thus, configuring ξ effectively reduces to identifying an appropriate value for K. For this experiment, we used the experimental setup from Table 5, where 13 MSR workloads share the cache with access rate ratios of $3:1:1:1:23:1:1:14:1:2:2:8:2$. The impact of K on algorithm performance was evaluated by varying K across the values 0, 1, 2, 4, and 8, and the results are illustrated in Figure 7.

As shown in Figure 7, the performance of the EFRC-I algorithm is largely insensitive to the value of K. In contrast, the performance of the EFRC-II algorithm exhibits variability when $K > 2$. Specifically, while larger values of K improve fairness, they tend to reduce cache hit rates. Given that one of the primary purposes of K is to minimize computational overhead, we recommend setting K to 1 or 2. For all other experiments in this article, K is fixed at 1.

4.5.2 Sensitivity of Parameter meta_w. The parameter *meta_w*, as defined in Table 2, determines the size of the history queues in the cache. Since the pruning operations on the history queues in our algorithm depend on this parameter, *meta_w* directly influences the size of the history space, or ghost cache, within the system. Although the size of a resident block is typically 512 bytes, modern systems often use a block size of 4 KB. In comparison, the size of a 64-bit address history block is only 8 bytes. Consequently, the storage resources occupied by history space are relatively small compared to the resident space. Nevertheless, it remains necessary to minimize the size of the history space and

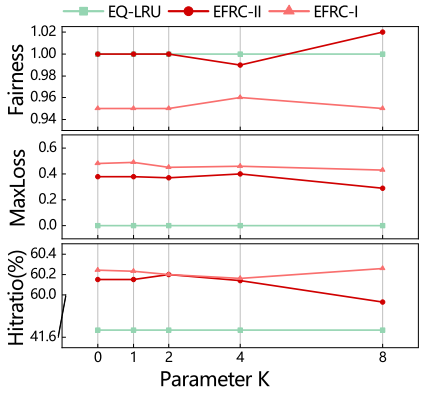


Fig. 7. Sensitivity of the parameter K.

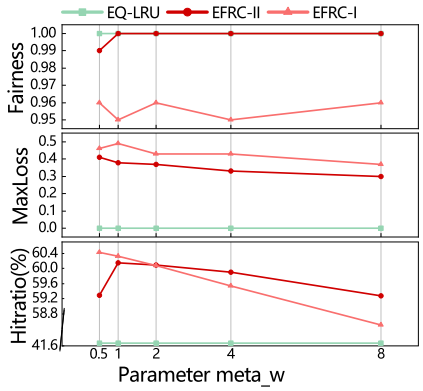


Fig. 8. Sensitivity of the parameter *meta_w*.

identify an optimal value for *meta_w* to balance algorithm performance and storage efficiency. In this experiment, we adopted the configuration outlined in Table 5, where a total of 13 MSR workloads shared the cache with access rate ratios set to $3:1:1:1:23:1:1:14:1:2:2:8:2$. To evaluate the effect of *meta_w* on algorithm performance, we tested the parameter using values 0.5, 1, 2, 4, and 8. The corresponding results, highlighting the relationship between *meta_w* and algorithm behavior, are presented in Figure 8.

As shown in Figure 8, the cache hit rates of both EFRC-I and EFRC-II are significantly influenced by *meta_w*, generally exhibiting an inverse relationship. For EFRC-I, the highest cache hit rate is achieved when $meta_w = 0.5$, corresponding to a ghost cache size of $0.5C$, with fairness also reaching its peak. For EFRC-II, the maximum cache hit rate occurs when $meta_w = 1$, corresponding to a ghost cache size of C, and fairness is similarly optimized. Based on these results, we recommend setting *meta_w* to 0.5 for EFRC-I and 1 for EFRC-II in practical applications.

4.5.3 Space Overhead Analysis. The storage cost for resident data blocks in our algorithms is C, which is consistent with the limitations imposed by any cache algorithm. Therefore, we focus only on the metadata or history block overhead in this analysis.

As discussed in Section 4.5.2, when the parameters *meta_w* are set to 0.5 and 1 for EFRC-I and EFRC-II, respectively, the corresponding metadata overheads are $0.5C$ and C, respectively.

Table 7. Time-consuming for Different Algorithms with Different Numbers of Shared Workloads and 100,000 Accesses per Workload

Number of shares	Time-consuming(seconds) under different algorithms							
	EQ-LRU	LRU	ARC	LIRS	CACHEUS	SIEVE	EFRC-I	EFRC-II
1	17.07	17.02	17.17	17.53	20.14	17.04	17.65	18.6
3	84.12	83.96	84.8	85.35	93.32	83.44	87.02	89.65
5	193.41	193.19	194.34	195.34	207.26	191.86	196.99	202.1
7	353.11	353.02	354.07	358.83	376.91	353.31	361.52	370.26
9	558.2	557.78	558.34	560.24	588.92	556.77	570.18	582.51
11	817.26	826.6	819.65	830.37	856.57	816.77	833.82	857.92
13	1,138.14	1,127.74	1,128.85	1,132.49	1,171.85	1,124.99	1,144.42	1,170.33

In comparison, the metadata overhead for LRU and SIEVE algorithms is 0, while ARC incurs an overhead of C, and LIRS and CACHEUS typically have metadata overheads not less than C. Considering that the size of a history block is much smaller than that of a resident data block, the metadata overhead of EFRC-I and EFRC-II is reasonable.

4.5.4 Time Overhead Analysis. The time complexity of the EFRC-I and EFRC-II algorithms is $O(m \cdot n)$, where m is the number of data access features and n is the number of workloads in the shared cache. Thus, EFRC-I has a time complexity of $O(n)$, while EFRC-I has a time complexity of $O(2n)$. If n is a small constant, the time complexity of both algorithms reduces to $O(1)$.

To validate this analysis, we conducted the experiment shown in Table 7. In this experiment, we measured the runtime of different algorithms under varying numbers of workloads sharing the cache, with each workload issuing 100, 000 requests. This request size was chosen to ensure that each algorithm could operate effectively within a sufficiently full cache. As shown in Table 7, the runtime of our algorithms across different numbers of shared workloads remains comparable to that of other algorithms, demonstrating consistent efficiency.

4.6 Supplementary Analysis

4.6.1 Supplementary Comparison. To validate the performance of our algorithms across a broader range of workloads and to compare them with other high-quality caching algorithms, we supplemented this experiment. The X (formerly known as Twitter) workload we selected is an object-based key-value storage workload, which is different from block-based MSR workloads. As with the approach taken in the S3-FIFO experiment, we ignored the size of the objects in the simulator because most production systems use slab storage (i.e., objects of a similar size) to implement cache replacement [55]. Within X's workload concentration, there are 54 distinct workloads categorized into three types: Computation, Storage, and Transient [54]. We selected 21 of these workloads based on statistical metrics such as production miss ratio and request rate, as well as considering script processing convenience. These are summarized in Table 8. Then we designed six scenarios for shared caching between different workload types: **Single-type Computation** (**SC**), **Single-type Storage** (**SS**), **Single-type Transient** (**ST**), **Mixed-type Computation** (**MC**), **Mixed-type Storage** (**MS**), and **Mixed-type Transient** (**MT**), as shown in Table 9. The access rate ratios of the workloads in these scenarios were configured according to the request rates listed in Table 8. It is worth noting that, although the access rate ratio between the *cluster*27 and *cluster*17 workloads exceeds 70 times, as shown in Table 8; these ratios were intentionally constrained to shorten the overall testing duration.

At the same time, we reproduced several well-known caching algorithms, such as S3-FIFO, LIRS2, DLIRS, CAR, and CLOCK, for comparative analysis [3, 29, 55, 59]. We also used the EFRC-I

Table 8. X Workload Description

Workload category	Cluster	Production Miss Ratio	Request Rate (KQPS)	WSS in MB (Working Set Size)	Footprint (MB)
Computation	*cluster*6	0.0268	12.61	128	459
	*cluster*12	0.0058	4.36	986	793
	*cluster*38	0.0022	2.52	21	247
	*cluster*40	0.1843	6.26	1,021	370
	*cluster*41	0.1831	5.18	849	302
	*cluster*43	0.1263	14.54	1,391	868
	*cluster*46	0.6854	10.47	16,437	826
Storage	*cluster*11	0.0047	4.16	1,751	165
	*cluster*14	0.1218	5.81	8,732	267
	*cluster*17	0.0002	0.35	1,162	392
	*cluster*24	0.0087	5.29	5,071	165
	*cluster*27	0.8061	24.55	4,372	1,229
	*cluster*29	0.2242	10.73	891	505
	*cluster*36	0.0375	9.08	2,696	298
Transient	*cluster*15	0.065	9.02	15	609
	*cluster*16	0.0728	17.51	18	557
	*cluster*20	0.0079	5.63	2,147	136
	*cluster*22	0.875	3.24	1,682	217
	*cluster*23	0.0259	8.69	91	309
	*cluster*30	0.0083	3.85	775	127
	*cluster*44	0.0083	9.41	536	331

Table 9. Scenario Description

Scene Name	Cluster combination	Access Rate Ratio
SC	6, 12, 38, 40, 41, 43, 46	13:5:3:7:6:15:11
SS	11, 14, 17, 24, 27, 29, 36	5:6:1:6:25:11:10
ST	15, 16, 20, 22, 23, 30, 44	10:18:6:4:9:4:10
MC	11, 17, 40, 43, 46, 22, 30	5:1:7:15:11:4:4
MS	14, 27, 29, 12, 38, 22, 30	6:25:11:5:3:4:4
MT	11, 17, 12, 38, 15, 16, 23	5:1:5:3:10:18:9

algorithmic framework to design the EFRC-I-SF algorithm based on S3-FIFO during this process. If you are familiar with the EFRC-I and S3-FIFO algorithms, designing the EFRC-I-SF algorithm should be relatively straightforward. The only potential difficulty is that the size of the ghost queue G in the S3-FIFO algorithm is the same as the size of the main queue M parameter. For each workload in the EFRC-I-SF algorithm, therefore, we have designed two historical cache queues, one to satisfy the requirements of the EFRC-I algorithm and one to satisfy the requirements of the S3-FIFO algorithm. The detailed experimental results for these algorithms are presented in Table 10.

The combined test results in Table 10 show that our EFRC-I, EFRC-II, and EFRC-I-SF algorithms perform consistently better than other algorithms in terms of fairness metrics. In terms of hit rate metrics, our EFRC-I and EFRC-II algorithms perform equally well against these excellent cache replacement algorithms. Although the S3-FIFO algorithm has a clear advantage in the X workload,

Table 10. The Hit Rate and Fairness Metrics of Different Algorithms in a Shared Cache Scenario for Six Combinations of X Workloads: SC, SS, ST, MC, MS, and MT

Algorithm	*HitRate*(%)	*Fairness*	*MaxLoss*	*HitRate*(%)	*Fairness*	*MaxLoss*
	SC			SS		
EFRC-I-SF	79.63	1.06	0.0	64.52	1.04	0.01
EFRC-II	75.01	1.01	0.01	63.58	1.02	0.03
EFRC-I	74.64	0.99	0.08	63.59	1.03	0.0
SIEVE	74.28	1.0	0.02	64.24	1.0	0.12
S3-FIFO	75.88	1.02	0.03	65.76	1.03	0.08
CACHEUS	72.3	0.89	0.43	63.29	0.99	0.13
LIRS2	74.31	0.88	0.51	64.8	1.02	0.09
DLIRS	73.29	0.88	0.49	63.92	1.0	0.13
LIRS	73.65	0.91	0.43	64.0	1.0	0.11
CAR	73.33	0.85	0.55	64.22	1.0	0.12
ARC	73.23	0.88	0.48	63.88	1.0	0.13
CLOCK	73.41	0.99	0.03	62.29	0.97	0.14
Algorithm	ST			MC		
EFRC-I-SF	72.51	1.04	0.0	79.74	1.02	0.0
EFRC-II	69.5	1.0	0.01	78.34	1.01	0.02
EFRC-I	69.25	1.0	0.01	77.92	1.01	0.0
SIEVE	68.29	0.99	0.04	78.37	0.99	0.11
S3-FIFO	69.18	1.0	0.03	80.01	1.01	0.08
CACHEUS	67.62	0.97	0.06	76.63	0.97	0.12
LIRS2	68.38	0.98	0.08	78.76	0.98	0.1
DLIRS	67.97	0.98	0.04	78.22	0.98	0.12
LIRS	67.79	0.98	0.06	78.05	0.97	0.12
CAR	68.34	0.99	0.04	78.26	0.98	0.11
ARC	68.02	0.99	0.04	78.02	0.98	0.12
CLOCK	67.6	0.98	0.05	77.88	0.98	0.11
Algorithm	MS			MT		
EFRC-I-SF	60.59	1.11	0.01	59.63	1.06	0.00
EFRC-II	51.51	1.03	0.01	55.63	1.02	0.03
EFRC-I	50.54	0.99	0.18	54.91	1.0	0.06
SIEVE	51.42	1.01	0.05	53.89	0.98	0.1
S3-FIFO	52.24	1.02	0.0	55.11	1.0	0.06
CACHEUS	50.46	0.95	0.25	53.33	0.96	0.11
LIRS2	49.88	0.76	0.69	53.42	0.81	0.63
DLIRS	50.94	0.96	0.28	53.51	0.96	0.13
LIRS	49.91	0.85	0.56	52.55	0.86	0.51
CAR	51.4	0.98	0.2	53.95	0.98	0.1
ARC	50.97	0.96	0.27	53.57	0.97	0.12
CLOCK	49.82	1.0	0.06	52.99	0.97	0.14

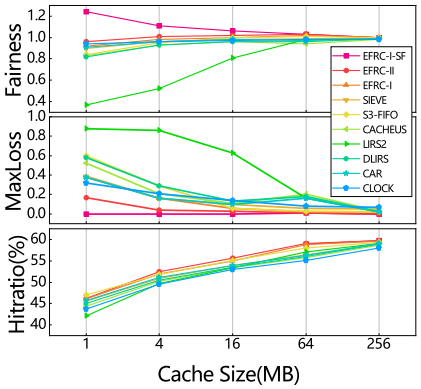


Fig. 9. Impact of the cache size C.

our newly designed EFRC-I-SF algorithm, based on S3-FIFO, achieves even more impressive results in terms of both hit rate and fairness metrics. These results unequivocally demonstrate the effectiveness of our EFRC algorithmic framework.

4.6.2 Impact of Cache Size. To investigate the impact of cache size on the performance of the algorithm, we conducted a supplementary experiment. We selected the MT workload scenario to verify whether our algorithms could maintain their performance advantage. As mentioned above, we did not consider object size within the simulator. To maintain compatibility with the cache size settings from the original MSR workload, we set the size of each object to that of a block in MSR, i.e., 512 bytes. We tested the algorithm's performance across cache sizes of 1 MB, 4 MB, 16 MB, 64 MB, and 256 MB and presented the results in Figure 9. To reduce testing time, we decreased the size of the dataset, which means that our experiments did not evaluate the performance of the algorithm at cache sizes in the gigabyte range. Consequently, Figure 9 shows that all performance metrics for the algorithm begin to converge at a cache size of 256 MB.

As can be seen in Figure 9, the hit rate and fairness metrics of each caching algorithm gradually improve as the cache size increases. The exception is the EFRC-I-SF algorithm, for which fairness gradually diminishes. This is because it performs excessively well at smaller cache sizes, resulting in a reduction in this advantage as the cache size increases. Meanwhile, our EFRC-I, EFRC-II, and EFRC-I-SF algorithms consistently outperform others across all cache sizes. Although performance convergence occurs among the algorithms at a cache size of 256 MB, ours still demonstrates superior performance at this point.

4.6.3 Special Comparison. To analyze the performance of our algorithm under specific workloads, such as *scan* and *churn* types, we supplemented this experiment. As the *Wdev_2* workload in the MSR dataset contains overlapping *churn* and *scan* workload segments; we selected it for testing the cache hit rate under single-workload conditions. The test results for these algorithms are presented in Table 11. As mentioned above, our EFRC-I algorithm exhibits behavior similar to that of the SIEVE algorithm under single-workload conditions. Despite employing a different resource adjustment strategy, the EFRC-II algorithm exhibits behavior similar to that of the ARC algorithm in single-workload scenarios. The ARC adjusts according to the size of the ghost queue, while the EFRC-II adjusts according to the size of the resident queue. However, when parameter $K = 0$, the EFRC-II algorithm ceases to execute its resource adjustment strategy. At this point, the frequency-based queue of EFRC-II occupies nearly all of the cache's resident resources. Meanwhile, our EFRC-I-SF algorithm exhibits behavior similar to the S3-FIFO algorithm under a single workload.

Table 11. Cache Hit Rates for Different Algorithms on the Workload *Wdev_2*

Workload	Cache Hit Rate(%) at different algorithms						
Wdev_2	LRU	CLOCK	CAR	LIRS	DLIRS	LIRS2	CACHEUS
	21.82	30.73	38.91	66.18	65.13	65.97	62.41
	ARC	S3-FIFO	SIEVE	EFRC-I	EFRC-II ($K = 1$)	EFRC-II ($K = 0$)	EFRC-I-SF (S3-FIFO-based)
	38.91	57.66	38.91	38.91	38.91	38.91	57.66

Table 12. The Hit Rate, Average Latency, and Fairness Metrics of Different Algorithms in a Shared Cache Scenario for Two Workloads *cluster*17 and *cluster*27 According to Access-rate Ratio 1 : 25

Algorithm	Hit Rate(%)			Avg.latency (µs)		Fairness	MaxLoss
	Cache	*cluster*17	*cluster*27	*cluster*17	*cluster*27		
EFRC-I-SF	38.19	97.85	35.8	1,200.67	6,810.5	1.06	0.0
EFRC-II	38.05	97.8	35.66	1,208.63	6,827.05	1.06	0.0
EFRC-I	37.93	97.91	35.53	1,194.08	6,826.87	1.06	0.0
SIEVE	38.2	88.49	36.19	2,043.33	6,753.99	1.01	0.1
S3-FIFO	38.82	91.97	36.7	1,729.77	6,711.72	1.04	0.06
CACHEUS	37.44	88.01	35.42	2,100.28	6,858.74	1.0	0.1
LIRS2	37.26	91.18	35.1	1,817.28	6,867.43	1.01	0.07
DLIRS	37.79	88.15	35.77	2,077.46	6,797.23	1.0	0.1
LIRS	36.95	89.0	34.87	2,001.79	6,878.92	0.99	0.09
CAR	38.31	88.86	36.28	2,010.43	6,750.2	1.01	0.09
ARC	37.81	88.19	35.8	2,072.97	6,794.33	1.0	0.1
CLOCK	36.77	85.9	34.8	2,275.82	6,878.74	0.97	0.12
LRU	36.2	85.34	34.23	2,328.06	6,930.18	0.96	0.13
EQ-LRU	34.22	98.06	31.67	1,183.03	7,160.9	1	0

As demonstrated in Table 11, our EFRC-I algorithm has hit rates comparable to the SIEVE algorithm; our EFRC-II algorithm has hit rates similar to those of the ARC and CAR algorithms; and our EFRC-I-SF algorithm has hit rates similar to those of the S3-FIFO algorithm. Although our algorithms have a higher hit rate than LRU and CLOCK, they still lag behind LIRS, DLIRS, LIRS2, and CACHEUS.

4.6.4 Higher-level Analysis. To conduct a more in-depth evaluation of the benefits that our algorithms yield in terms of the latency metric, we supplemented this experiment. By comprehensively comparing the test results across the six scenarios in Table 10, it is evident that our algorithm's performance in the SS scenario is relatively lacklustre. This has prompted us to conduct a deeper analysis of this particular scenario. To simplify the analysis, we selected two workloads, *cluster*17 and *cluster*27, for testing, setting their access rate ratio to 1:25 relative to the request rate. Since this is a simulation experiment, we have chosen average latency as the performance metric. In our experimental environment, we observed that the algorithmic processing time for a single request fluctuated within a range of several hundred microseconds. To mitigate such interference, we set the processing time for a request that is a hit to 1 millisecond plus the algorithmic processing time for that request, while the processing time for a request that is a miss is set to 10 milliseconds plus the algorithmic processing time for that request. We then calculate the final average latency metric by taking the average of all request processing times. Table 12 presents the measured hit rate, average latency, and fairness metrics.

As can be seen in Table 12, our EFRC-I, EFRC-II, and EFRC-I-SF algorithms outperform others in terms of fairness metrics. However, the SIEVE, S3-FIFO, and CAR algorithms demonstrate superiority in terms of cache hit rate. Further examination of the hit rates for the *cluster*17 and *cluster*27 workloads reveals that the SIEVE, S3-FIFO, and CAR algorithms enhance the *cluster*27 workload's hit rate at the significant expense of the *cluster*17 workload's hit rate. While the overall cache hit rate improves, the average latency for the *cluster*17 workload increases considerably. Compared to the EQ-LRU algorithm, our EFRC-I, EFRC-II, and EFRC-I-SF algorithms guarantee an average latency for *cluster*17, reduce the average latency for *cluster*27, and effectively enhance the overall cache hit rate.

5 Related Work

5.1 Cache Replacement Algorithms for a Single Workload

We classify existing cache replacement algorithms according to the motivations behind their research. **(1) Addressing new types of workloads or phenomena.** In Section 2.2, we described the `LRU-friendly`, `LFU-friendly`, `Scan`, and `Churn` workloads and their corresponding algorithmic designs. Furthermore, the LRU-K algorithm incorporates considerations for Correlated References and the Reference Retained Information Problem arising within access patterns [36]. The S3-FIFO algorithm addresses the one-hit wonder phenomenon observed in data access patterns [55]. **(2) Relying on new data metrics or algorithmic concepts.** The LIRS algorithm is designed based on the concept of reuse distance, while the LIRS2 algorithm extends this idea by considering two recent reuse distances instead of one [21, 59]. The LHD algorithm improves cache hit performance by maximizing hit density, and the EVA algorithm enhances cache efficiency by evaluating the economic value added of cache blocks [4, 5]. Furthermore, the GL-Cache algorithm optimizes overall cache performance through learning at the object group level [53]. **(3) Improving upon previous algorithms.** The Window-LFU algorithm addresses the limitation of the LFU algorithm, which requires maintaining access records for all data items [6, 24]. The LFU-Aging algorithm introduces an aging mechanism to alleviate cache pollution caused by long-lived hot data [1]. In addition, the 2Q algorithm achieves a low-overhead design inspired by the LRU-2 algorithm, thereby improving efficiency while retaining effectiveness [22, 36]. **(4) Integrating previous algorithms.** The ARC algorithm integrates both recency and frequency characteristics to improve adaptability [31]. Building on this idea, the CAR algorithm combines the CLOCK and ARC mechanisms, while the DLIRS algorithm merges the design principles of the LIRS and ARC algorithms [3, 29]. The CACHEUS algorithm further extends this line of work by adopting a multi-expert decision-making framework that synthesizes the designs of LRU and LFU [42]. **(5) Exhibiting unique insights** The ARC algorithm enhances the adaptability of caching strategies, demonstrating its ability to dynamically balance recency and frequency. The TinyLFU algorithm, on the other hand, introduces a novel mechanism that determines whether newly accessed data should trigger cache replacement, thereby improving decision precision and cache utilization [10]. **(6) Inspired by practical issues or newly observed phenomena.** The CLOCK algorithm addresses the lock contention problem arising from frequent cache page movements in multi-threaded environments when using the LRU algorithm. The SIEVE algorithm, in contrast, is inspired by the patterns and practices observed in production code written by developers [57]. **(7) Exploring the characteristics of an ideal algorithm.** The Hawkeye algorithm improves cache performance by learning from the optimal behavior defined by the Belady algorithm [17]. The MQ algorithm, on the other hand, is designed to incorporate three key properties of an effective cache replacement policy: minimal lifetime, frequency-based prioritization, and temporal frequency [62]. **(8) Introducing techniques from other research domains.** The Glider algorithm builds upon the recurrent neural network framework of **Long Short-Term Memory** (**LSTM**), leveraging its sequence-learning capability to enhance cache

management [45]. Similarly, the 3L-Cache algorithm integrates the **Gradient Boosting Machine** (**GBM**) method to improve predictive accuracy in replacement decisions [61].

5.2 Cache Partitioning Methods for Multiple Workloads

The core objective of cache partitioning is to reduce performance interference among multiple workloads through rational allocation of shared cache resources. Most existing approaches are grounded in the theoretical foundation of MRCs, which describe the miss rate corresponding to a specific cache size. The central goal of such methods is typically to minimize the sum of the average miss rates across all workloads [49]. However, Rajkumar et al. theoretically proved that determining a globally optimal partitioning scheme under unconstrained conditions is an NP-hard problem, which cannot be solved in polynomial time [41].

To address this challenge, Stone et al. proposed a simplifying assumption and an approximate scheme. When the MRCs of all workloads exhibit convexity—that is, when the rate of decrease in miss rate diminishes as cache capacity increases, reflecting diminishing marginal benefits—the optimal partitioning can be achieved by equalizing the derivatives of all MRCs. At this point, each unit of cache resource contributes an equal reduction in miss rate across workloads, resulting in a globally minimal overall miss rate [49]. Owing to its simplicity and strong interpretability, this principle and its variants have been widely applied to multi-level cache systems, including CPU, main memory, and storage subsystems.

In CPU on-chip caches, Suh et al. developed a Way-level dynamic partitioning algorithm based on Stone's derivative-balance principle. They used per-Way hit counters to monitor, in real time, the marginal benefit of each workload and iteratively reallocated cache Ways to achieve derivative balance [50]. Qureshi and Patt later introduced an **Auxiliary Tag Directory** (**ATD**) to locally sample the MRC, allowing efficient derivative estimation without full-scale statistics and significantly reducing hardware storage overhead [40].

In main memory, Zhou et al. extended Stone's method to page cache partitioning. They dynamically tracked page-level MRCs and adjusted the cache quota of each process through page migration to maintain derivative balance, thus adapting to the access characteristics of main memory [60]. Soundararajan et al. further combined MRC derivatives with a memory access latency model, translating "miss rate reduction" into "latency optimization benefit," thereby enabling coordinated partitioning between main memory and CPU cache to improve overall system performance [47].

In SSD caches, Meng et al. adapted Stone's approach to the read–write asymmetry of SSDs by assigning higher weights to write miss rates when computing MRC derivatives, thus balancing read–write performance and prolonging SSD lifetime [32]. Although the Centaur scheme proposed by Koller et al. targets latency optimization rather than miss rate minimization, it still estimates the underlying "cache capacity–miss rate" relationship using MRC derivatives, providing a quantitative basis for latency reservation and adapting well to SSD I/O characteristics [27].

Beyond MRC-derivative-based techniques, a number of cache partitioning approaches have been proposed with alternative objectives, among which fairness is a major focus. Kim et al. designed a miss-rate-balanced partitioning scheme for CPU caches, which dynamically adjusts Way allocations to reduce miss-rate disparities among workloads and improve execution-time fairness [26]. Srikantaiah et al. introduced a PID-controller-based approach targeting fair speedup, which triggers derivative rebalancing when workload performance deviations exceed a given threshold, thereby maintaining a balance between fairness and throughput [48].

Other methods combine Way-level cache partitioning with resource management for CPU cores or DRAM bandwidth to achieve multi-objective optimization. For instance, SATORI explores multi-resource partitioning using Bayesian optimization to dynamically adjust the priorities between throughput and fairness, providing a practical solution for balancing these two objectives in

multi-core systems [43]. To address scalability and fairness challenges in large-scale multi-core environments, XChange models resource allocation as a distributed market and employs core bidding to achieve decentralized coordination, offering a new paradigm for balancing throughput, fairness, and scalability [51].

6 Conclusion

Current cache management methodologies often lack the necessary granularity in resource allocation and fail to provide the intelligent integrated management required for seamless collaboration among diverse workloads. In this study, we devised algorithms that thoroughly integrate cache replacement and fair resource sharing among varying co-located workloads. Our rigorous experiments demonstrate that the proposed EFRC-I and EFRC-II algorithms, bearing low overhead, possess remarkable cache efficiency, fairness, scalability, compatibility, and anti-cheating.

Acknowledgments

We greatly appreciate the insightful feedback from the anonymous reviewers.

References

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

[2] Ali Bakhoda, John Kim, and Tor M. Aamodt. 2010. Throughput-effective on-chip networks for manycore accelerators. In *Proceedings of the 43rd IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*. IEEE Computer Society, 421–432.

[3] Sorav Bansal and Dharmendra S. Modha. 2004. CAR: Clock with adaptive replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)*. USENIX, 187–200.

[4] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving cache hit rate by maximizing hit density. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*. USENIX Association, 389–403.

[5] Nathan Beckmann and Daniel Sánchez. 2017. Maximizing cache performance under uncertainty. In *Proceedings of the 23rd IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. IEEE Computer Society, 109–120.

[6] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. 1999. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the 18th IEEE International Conference on Computer Communications (INFOCOM'99)*. IEEE Computer Society, 126–134.

[7] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. ACM, 107–120.

[8] Christina Delimitrou and Christos Kozyrakis. 2018. Amdahl's law for tail latency. *Commun. ACM* 61, 8 (2018), 65–72.

[9] Alan J. Demers, Srinivasan Keshav, and Scott Shenker. 1989. Analysis and simulation of a fair queueing algorithm. In *Proceedings of the ACM SIGCOMM Conference on Communications Architectures, Protocols and Applications (SIGCOMM'89)*. ACM, 1–12.

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

[11] Stijn Eyerman and Lieven Eeckhout. 2008. System-level performance metrics for multiprogram workloads. *IEEE Micro* 28, 3 (2008), 42–53.

[12] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating interference at microsecond timescales. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, 281–297.

[13] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic scheduling in multi-resource clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, 65–80.

[14] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann.

[15] Rishabh R. Iyer, Musa Unal, Marios Kogias, and George Candea. 2023. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP'23)*. ACM, 466–481.

[16] Ravi R. Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Donald Newell, Yan Solihin, Lisa R. Hsu, and Steven K. Reinhardt. 2007. QoS policies and architecture for cache/memory in CMP platforms. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*. ACM, 25–36.

[17] Akanksha Jain and Calvin Lin. 2016. Back to the future: Leveraging Belady's algorithm for improved cache replacement. In *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'16)*. IEEE Computer Society, 78–89.

[18] Akanksha Jain and Calvin Lin. 2019. *Cache Replacement Policies*. Morgan and Claypool Publishers.

[19] Raj Jain, Arjan Durresi, and Gojko Babic. 1999. Throughput fairness index: An explanation. In *Proceedings of the ATM Forum Contribution*. 1–13.

[20] Congfeng Jiang, Yitao Qiu, Weisong Shi, Zhefeng Ge, Jiwei Wang, Shenglei Chen, Christophe Cérin, Zujie Ren, Guoyao Xu, and Jiangbin Lin. 2020. Characterizing co-located workloads in alibaba cloud datacenters. *IEEE Trans. Cloud Comput.* 10, 4 (2020), 2381–2397.

[21] Song Jiang and Xiaodong Zhang. 2002. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '02)*. ACM, 31–42.

[22] Theodore Johnson and Dennis E. Shasha. 1994. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*. Morgan Kaufmann, 439–450.

[23] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A. Kim. 2012. Measuring interference between live datacenter applications. In *Proceedings of the SC Conference on High Performance Computing Networking, Storage and Analysis (SC'12)*. IEEE/ACM, 1–12.

[24] George Karakostas and Dimitrios N. Serpanos. 2002. Exploitation of different types of locality for Web caches. In *Proceedings of the IEEE International Symposium on Computers and Communications (ISCC'02)*. IEEE Computer Society, 207–212.

[25] Harshad Kasture and Daniel Sánchez. 2014. Ubik: Efficient cache sharing with strict QoS for latency-critical workloads. *ACM SIGPLAN Notices* 49, 4 (2014), 729–742.

[26] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. 2004. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*. IEEE Computer Society, 111–122.

[27] Ricardo Koller, Ali José Mashtizadeh, and Raju Rangaswami. 2015. Centaur: Host-side SSD caching for storage performance control. In *Proceedings of the IEEE International Conference on Autonomic Computing, Grenoble*. IEEE Computer Society, 51–60.

[28] Neeraj Kulkarni, Gonzalo Gonzalez-Pumariega, Amulya Khurana, Christine A. Shoemaker, Christina Delimitrou, and David H. Albonesi. 2020. CuttleSys: Data-driven resource management for interactive services on reconfigurable multicores. In *Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*. IEEE, 650–664.

[29] Cong Li. 2018. DLIRS: Improving low inter-reference recency set cache replacement policy with dynamics. In *Proceedings of the 11th ACM International Systems and Storage Conference (SYSTOR'18)*. ACM, 59–64.

[30] Yi Liang, Shaokang Zeng, and Lei Wang. 2023. Quantifying resource contention of co-located workloads with the system-level entropy. *ACM Trans. Archit. Code Optim.* 20, 1 (2023), 1–25.

[31] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*. Jeff Chase (Ed.). USENIX.

[32] Fei Meng, Li Zhou, Xiaosong Ma, Sandeep Uttamchandani, and Deng Liu. 2014. vCacheShare: Automated server flash cache space management in a virtualization environment. In *Proceedings of the USENIX Annual Technical Conference (ATC'14)*. USENIX Association, 133–144.

[33] Amirhossein Mirhosseini and Thomas F. Wenisch. 2019. The queuing-first approach for tail management of interactive services. *IEEE Micro* 39, 4 (2019), 55–64.

[34] Amirhossein Mirhosseini, Brendan L. West, Geoffrey W. Blake, and Thomas F. Wenisch. 2020. Q-zilla: A scheduling framework and core microarchitecture for tail-tolerant microservices. In *Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. IEEE, 207–219.

[35] Dushyanth Narayanan, Austin Donnelly, and Antony I. T. Rowstron. 2008. Write off-loading: Practical power management for enterprise storage. *ACM Trans. Storage* 4, 3 (2008), 1–23.

[36] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'93)*. ACM Press, 297–306.

[37] Jinsu Park, Seongbeom Park, and Woongki Baek. 2019. Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys'19)*. ACM, 1–16.

[38] Sejin Park and Chanik Park. 2017. FRD: A filtering based buffer cache algorithm that considers both frequency and reuse distance. In *Proceedings of the 33rd IEEE International Conference on Massive Storage Systems and Technology (MSST'17)*, Vol. 6. 9–1.

[39] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. 2016. FairRide:Near-optimal, fair cache sharing. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*. USENIX Association, 393–406.

[40] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE Computer Society, 423–432.

[41] Ragunathan (Raj) Rajkumar, Chen Lee, John P. Lehoczky, and Daniel P. Siewiorek. 1998. Practical solutions for QoS-based resource allocation problems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*. IEEE, 296–306.

[42] Liana V. Rodriguez, Farzana Beente Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. 2021. Learning cache replacement with CACHEUS. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*. USENIX Association, 341–354.

[43] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2021. Satori: Efficient and fair resource partitioning by sacrificing short-term benefits for long-term gains. In *Proceedings of the 48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'21)*. IEEE, 292–305.

[44] Daniel Sánchez and Christos Kozyrakis. 2011. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA'11)*. ACM, 57–68.

[45] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*. ACM, 413–425.

[46] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. 2020. Learning relaxed belady for content distribution network caching. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. USENIX Association, 529–544.

[47] Gokul Soundararajan, Daniel Lupei, Saeed Ghanbari, Adrian Daniel Popescu, Jin Chen, and Cristiana Amza. 2009. Dynamic resource allocation for database servers running on virtual storage. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*. USENIX, 71–84.

[48] Shekhar Srikantaiah, Mahmut T. Kandemir, and Qian Wang. 2009. SHARP control: Controlled shared cache management in chip multiprocessors. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. ACM, 517–528.

[49] Harold S. Stone, John Turek, and Joel L. Wolf. 1992. Optimal partitioning of cache memory. *IEEE Trans. Computers* 41, 9 (1992), 1054–1068.

[50] G. Edward Suh, Larry Rudolph, and Srinivas Devadas. 2004. Dynamic partitioning of shared cache memory. *J. Supercomput.* 28, 1 (2004), 7–26.

[51] Xiaodong Wang and José F. Martínez. 2015. XChange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE Computer Society, 113–125.

[52] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley M. Felter, Juan Rubio, and Yang Li. 2018. dcat: Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys'18)*. ACM, 1–13.

[53] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. 2023. GL-Cache: Group-level learning for efficient and high-performance caching. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST'23)*. USENIX Association, 115–134.

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

[55] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. 2023. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*. ACM, 130–149.

[56] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. 2021. Don't forget the I/O when allocating your LLC. In *Proceedings of the 48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'21)*. IEEE, 112–125.

[57] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K. V. Rashmi. 2024. SIEVE is simpler than LRU: An efficient Turn-Key eviction algorithm for web caches. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*. USENIX Association, 1229–1246.

[58] Laiping Zhao, Yanan Yang, Kaixuan Zhang, Xiaobo Zhou, Tie Qiu, Keqiu Li, and Yungang Bao. 2020. Rhythm: Component-distinguishable workload deployment in datacenters. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys'20)*. ACM, 1–17.

[59] Chen Zhong, Xingsheng Zhao, and Song Jiang. 2021. LIRS2: An improved LIRS replacement algorithm. In *Proceedings of the 14th ACM International Conference on Systems and Storage (SYSTOR'21)*. ACM, 19:1–19:12.

[60] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. 2004. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*. ACM, 177–188.

[61] Wenbin Zhou, Zhixiong Niu, Yongqiang Xiong, Juan Fang, and Qian Wang. 2025. 3L-Cache: Low overhead and precise learning-based eviction policy for caches. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies (FAST'25)*. USENIX Association, 237–254.

[62] Yuanyuan Zhou, James Philbin, and Kai Li. 2001. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'01)*. USENIX, 91–104.

Received 20 April 2025; revised 18 October 2025; accepted 3 February 2026