P_4LRU: Towards An LRU Cache Entirely in Programmable Data Plane

Yikai Zhao[†]
Peking University
Beijing, China
zyk@pku.edu.cn

Wenrui Liu[†]
Peking University
Beijing, China
liuwenrui@pku.edu.cnn

Fenghao Dong[†]
Peking University
Beijing, China
dfh@pku.edu.cn

Tong Yang[†]
Peking University
Beijing, China
yangtongemail@gmail.com

Yuanpeng Li[†]
Peking University
Beijing, China
liyuanpeng@pku.edu.cn

Kaicheng Yang[†]
Peking University
Beijing, China
ykc@pku.edu.cn

Zirui Liu[†]
Peking University
Beijing, China
zirui.liu@pku.edu.cn

Zhengyi Jia[‡]
Huawei Cloud Computing
Technologies Co., Ltd.
Beijing, China
jiazhengyi@huawei.com

Yongqiang Yang[‡]
Huawei Cloud Computing
Technologies Co., Ltd.
Beijing, China
yangyongqiang@huawei.com

ABSTRACT

[*]The data plane cache, a critical functionality found in numerous network devices, such as programmable switches, intelligent NICs, and DPUs, is often subject to limitations in its programmability and memory access capacity. As a result, the majority of existing data plane caches rely on simple and inefficient replacement policies. This paper is set to introduce LRU, a near-optimal replacement policy, into the programmable data plane. We first explore the reasons why the traditional implementation of LRU is not suitable for deployment on the data plane. Consequently, we propose P_4LRU, a pipeline-optimized version of the LRU implementation. Building on P4LRU, we conceive three distinct in-network systems – LruTable, LruIndex, and LruMon, and successfully bring them to life on Tofino switches. Our thorough experimental trials establish that P_4LRU provides a significant performance boost over existing data plane caches in these three systems. We have open-sourced the source codes for the three systems on GitHub [1].

CCS CONCEPTS

• **Networks → Programmable networks**; *In-network processing*; Network monitoring; • **Theory of computation → Caching and paging algorithms**.

[*]Yikai Zhao and Wenrui Liu contribute equally to this paper. Tong Yang (yangtongemail@gmail.com) is the corresponding author.
[†]National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University, Beijing, China.
[‡]Huawei Cloud Computing Technologies Co., Ltd., Beijing, China.

ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09…$15.00
https://doi.org/10.1145/3603269.3604813

KEYWORDS

Data Plane Cache; Least Recently Used; Programmable Switches

ACM Reference Format:
Yikai Zhao, Wenrui Liu, Fenghao Dong, Tong Yang, Yuanpeng Li, Kaicheng Yang, Zirui Liu, Zhengyi Jia, and Yongqiang Yang. 2023. **P_4LRU: Towards An LRU Cache Entirely in Programmable Data Plane**. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3603269.3604813

1 INTRODUCTION

1.1 Background and Motivation

Caching is a vital and core component in computer science. The storage in most computer systems is organized into a hierarchical structure revolving around the processor, with the storage media closest to the processor delivering the highest performance. Therefore, positioning the cache at the apex of this hierarchical structure can markedly boost system performance. For switches offering network functions [31], such as forwarding, routing, and firewalling, the on-chip memory in the data plane is the closest to the packet processing logic, whereas the off-chip memory in the control plane augments the capacity. In the case of remote services with a client-server architecture, the on-chip memory in the switches around the client is closer to the user than the server memory, hence can yield higher performance. Therefore, establishing an efficient cache in the data plane of switches offers considerable benefits to network functions and remote services.

The most significant measure to evaluate cache is the hit rate, which is directly determined by the cache replacement policy. Over the years, various cache replacement policies have been proposed by researchers, including LRU (Least Recently Used) [40], LFU (Least Frequently Used) [48], LFRU, [8, 32] and others, with LRU being the most universal and tested policy. Multiple studies indicate that LRU performs exceptionally well in most scenarios, only slightly behind more complex replacement policies based on dedicated design or machine learning [5, 6, 20, 23]. As such, LRU is often considered the go-to choice. However, implementing an efficient and strict LRU is challenging, even on a CPU platform. To attain $O(1)$ complexity, the renowned Memcached [21] implementation uses a doubly linked-list to record entries in LRU order and employs a linked hash table to swiftly locate entries. Its successor, MemC3 [20], utilizes a cuckoo hash table [42] to improve the loading rate and applies the CLOCK algorithm [14] to approximate LRU, hence saving memory. Regrettably, neither the linked hash table nor the cuckoo hash table can be implemented in the data plane, and the scanning thread required by the CLOCK algorithm poses a considerable challenge for the data plane.

In the context of switches, data plane caching often serves not only as a read-cache but also as a write-cache. For instance, both Netseer [59] and Beaucoup [9] employ caching for flow-level information on the data plane. Consequently, each incoming packet alters the value of the corresponding cache entry, adding to the complexity of implementing the cache replacement policy. Despite these challenges, none of the existing works have successfully implemented the LRU policy in the data plane. Instead, they have primarily opted for two policies: the timeout policy and the LFU policy.

- The **timeout policy** involves using a hash table to log the cached entries, where each entry is linked with a timestamp noting the last access time. Beaucoup [9] is a typical example of this approach. In the event of a hash collision, they only replace the old entry with the incoming packet if the timestamp of the former has expired. The major drawback of this method is the need for careful timeout threshold setting; otherwise, the hit rate would significantly decrease.
- On the other hand, the **LFU policy** utilizes a multi-level hash table to log the cached entries, and each entry's access frequency is recorded. CocoSketch [58], Elastic [57], and HashPipe [51] are typical examples of this approach. In case of a hash collision, they employ different frequency-based replacement policies to decide whether to evict the old entry, aiming to cache the most frequently accessed flows. However, this method's downside is that entries hit many times tend to be cached for a long duration, even though there might not be any new access.

This paper, therefore, strives to achieve an approximate LRU replacement policy on the programmable data plane. The objectives are to (**R1**) meet the programming constraints of the data plane and ensure implementation on commercial programmable switches (*e.g.*, the Tofino switch), (**R2**) achieve cache performance nearly equivalent to an ideal LRU replacement policy, and (**R3**) utilize additional storage cost acceptably, without impacting the data plane's throughput.

1.2 Our Proposed Solution

The primary reason why standard LRU implementations cannot be applied in the programmable data plane is due to the strict data access requirements of the latter. The cached data must be segmented and positioned at various stages within the data plane. Each packet traversing through the data plane can only access a small block of data (*e.g.*, 8 bytes) at each stage and cannot repeatedly access the same data block across different stages. However, almost all conventional LRU implementations necessitate a second access to the same data (refer to § 2.1 for further details). To realize an approximate LRU replacement policy, we progressively investigate and propose the following essential techniques.

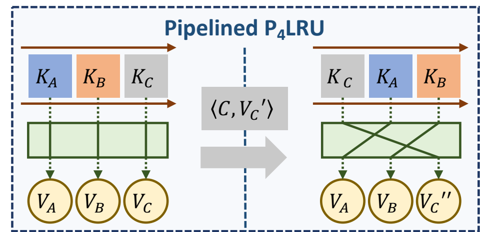


Figure 1: A simple example of our P_4LRU ($n = 3$).

Design of P_4LRU without Second Data Traversal: Conventional LRU implementations necessitate second data access due to their data placement scheme that stores keys and values together. We discovered that separating the keys and values eliminates this need. We designed P_4LRU, which keeps keys and values in different orders and maintains a Deterministic Finite Automaton (DFA) called cache state to denote the mapping relationship between keys and values. As illustrated in Figure 1, P_4LRU arranges keys in LRU order in the queue within the pipeline, but keeps the order of all values constant. For each incoming packet, P_4LRU places it at the head of the queue and modifies the cache state to maintain the correct mapping relationship, thereby avoiding a second access to the same data block.

Design of Stateful ALU-Based Cache State DFA: Given the limited programming model of the current programmable data plane, implementing a DFA poses a significant challenge. Specifically, the P_4LRU cache that records n entries has $n!$ states, and each state has n transitions. This implies that we need n tables, each with a size of $n!$, to log all transitions. However, on the current programmable data plane, *when operating the stateful register*, we can only access a tiny table[1]. Luckily, when the cache has fewer entries, we can encode the cache states into numbers and define the state transition of the DFA through arithmetic logic. For example, using a well-designed coding scheme, we can depict 18 transitions of the DFA of P_4LRU cache that records 3 entries ($n = 3$) with only five simple numerical operations, implemented with three stateful arithmetic logic units (ALUs).

The Series & Parallel Connection Technique of P_4LRU Units. The P_4LRU scheme can maintain a strict LRU cache with fewer entries. To scale the cache capacity and adhere more closely to the ideal LRU when the capacity is large, we propose a technique involving the serial and parallel linking of P_4LRU units. The Parallel Connection Technique substitutes the buckets of a hash table with P_4LRU cache units of $n = 2$ or $n = 3$ to accomplish arbitrary cache capacity. The Series Connection Technique links multiple P_4LRU cache units in series to construct a deeper, albeit approximate, LRU structure. Parallel connection is always advantageous, but serial connection may introduce duplicate entries, thus preventing the achievement of higher cache performance. However, in certain

[1]Of course, the programmable data plane allows us to access sufficiently large flow tables either *before* or *after* operating the register.

scenarios, we have found opportunities to avoid duplicate entries. Whenever each key requires twice the access to the data plane (*e.g.*, round trip), we can separate the cache query and update, avoid entry duplication, and make the cache utilizing the series connection technique closer to the ideal LRU.

We categorize three types of data plane caches that can be served by P_4LRU: (1) **Local Read-Cache**: It caches the most accessed entries in the large-scale flow table stored in control plane memory onto the data plane, accelerating packet forwarding. (2) **Remote Read-Cache**: It caches the most accessed data from a remote memory server on the data plane, thereby speeding up remote queries. (3) **Remote Write-Cache**: It buffers flow-level information destined for a remote server on the data plane, conserving upload or transmission bandwidth. For these cache types, we have developed three prototype systems to assess the performance of our P_4LRU: (1) **LruTable**, a Network Address Translation (NAT) system [31] that employs P_4LRU to cache fast path table entries on the data plane, achieving up to a 35% reduction in additional latency compared to the baseline. (2) **LruIndex**, an in-network query acceleration system that uses the P_4LRU with the series connection technique to cache database indexes. It can increase the throughput speedup by up to 8% compared to the baseline. (3) **LruMon**, a network telemetry system that uses TowerSketches [56] to filter minor flows and employs P_4LRU to aggregate and measure major flows on the data plane. This can reduce the upload or transmission volume of the telemetry system by up to 35%.

1.3 Key Contributions

- We explore the reasons why typical LRU implementations can't be deployed on a programmable data plane, and we introduce a novel LRU implementation, P_4LRU, that complies with pipeline programming constraints.
- We investigate the actual deployment of P_4LRU units encompassing two or three entries on the current programmable data plane, and put forth the series connection technique to further enhance cache performance in specific scenarios.
- We leverage the P_4LRU cache to construct three practical data plane systems – LruTable, LruIndex and LruMon on the programmable switch, and we evaluate the performance and scalability of the P4LRU cache through comprehensive experiments.

Ethics: This work does not raise any ethical issue.

2 IN-NETWORK P_4LRU

In this section, we first explore how to create an LRU cache within a network in a pipelined fashion. Then, we elaborate on the process of setting up an LRU cache that accommodates two or three key-value pairs on the current programmable data plane.

2.1 Limitations on Implementing LRU

LRU cache has two prevalent implementation methods: timestamp-based LRU cache and queue-based LRU cache. First, we introduce these two implementation methods, followed by an explanation as to why they cannot be implemented in a pipelined manner.

Timestamp-based LRU Cache. Figure 2 illustrates the use of a bucket array $C[1 \cdots n]$ of width n to store cache entries in a timestamp-based LRU. Each bucket includes three fields $\langle k, v, t\rangle$.

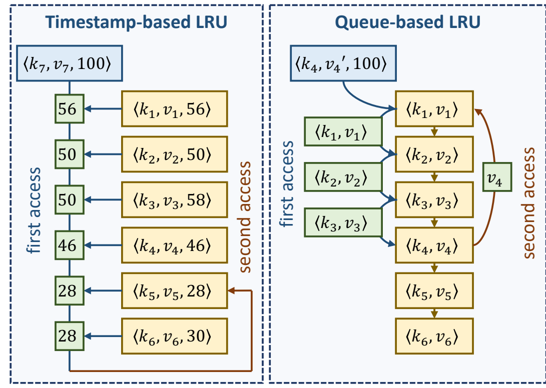


Figure 2: Examples of two LRU implementations.

For a bucket $C[i]$ that stores an entry, $C[i].k$ represents the key, $C[i].v$ represents the value, and $C[i].t$ represents the most recent access time of the entry. For a new item $\langle k', v'\rangle$, we traverse through n buckets. If there exists a bucket $C[i]$ that fulfills $C[i].k = k'$, we update the value $C[i].v$ and timestamp $C[i].t$. Otherwise, if there are available empty buckets, we select one to store the key k' and value v'. If there are no empty buckets, we eliminate the bucket with the oldest timestamp to free up space.

Limitations: In the worst-case scenario for implementing the timestamp-based LRU cache, we must traverse the bucket array C twice. When an incoming key isn't found in any bucket and no bucket is empty, we locate the bucket with the oldest timestamp in the first pass. Then, in the second pass, we alter the stored entry in that bucket. However, this necessity to *access the same data twice* contravenes the principles of pipeline programming. Using P4 language as an example, the data plane pipeline consists of multiple stages. Each stage can only access a restricted number of data blocks with a confined bit width (*e.g.*, 64 bits). The program is not allowed to access the same data block across different stages.

Queue-based LRU Cache. As illustrated in Figure 2, the queue-based LRU cache utilizes a queue Q, with a capacity of n, to store entries. Every entry $\langle k, v\rangle$ within the queue only keeps a record of the key k and value v, without accounting for any timestamp. For any incoming item $\langle k', v'\rangle$, we scan the entirety of the queue. If we find an existing entry $\langle k, v\rangle$ that fulfills $k = k'$, we proceed to update the value v and shift this entry to the beginning of the queue. In contrast, if the current number of entries registered in queue Q has not reached its limit n, we generate a new entry at the front of the queue, logging the key k' and value v' within. Lastly, if the queue Q is at full capacity, we expunge the final entry at the tail end of the queue.

Limitations: The implementation of the queue-based LRU cache still confronts the issue of *accessing the same data twice.* The queue needs to be configured in the pipeline in a sequenced manner. The front of the queue is placed at the first stage; the second entry is placed in the subsequent stage, and this continues down the line. For an incoming key, we have to store the key at the head of the queue, and the initial head entry is displaced to the second stage, with subsequent entries shifting down the pipeline. However, during this process, if we encounter an entry that contains the same key

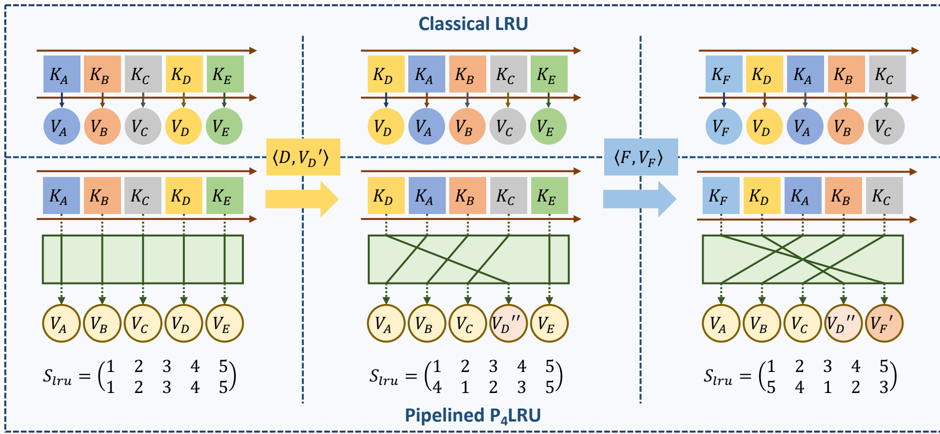


Figure 3: An example of our in-network LRU algorithm.

as the incoming key, we need to update the value of the entry at the queue's head with this entry's value. This requirement results in a second access to the queue's head.

2.2 Implementing P$_4$LRU in the Pipeline

Rationale: Our understanding is that the fundamental obstruction preventing the implementation of LRU cache methods, as discussed in Section 2.1, in a pipelined manner, originates from the fact that *both methods locate the key and value of an entry together.* As seen in Figure 2, this prohibits us from obtaining the position of the oldest key (in the case of the timestamp-based LRU) or the initial value of the incoming key (in the case of the queue-based LRU) during the first access to the LRU cache. However, storing keys and values separately could allow for the implementation of LRU in a pipelined manner. *We keep all keys arranged in the LRU order, while maintaining a consistent order of values.* Instead of altering the order of values, we discern the position of the value to be modified in each operation through a deterministic finite automaton (DFA) that represents the current key-value mapping state of the LRU cache.

Data Structure of P$_4$LRU: As illustrated in Figure 3, our P$_4$LRU deviates from the typical LRU cache implementation by storing keys and values separately. For an LRU cache with a capacity of n, P$_4$LRU incorporates a key array $\texttt{key}[1\cdots n]$ with a width of n (stored across n stages), a value array $\texttt{val}[1\cdots n]$ with a width of n, and a DFA state S_{lru} referred to as the cache state. $S_{lru} = \begin{pmatrix} 1 & \cdots & n \\ p_1 & \cdots & p_n \end{pmatrix}$ fundamentally represents a permutation, documenting the mapping relationship between positions in the key array and those in the value array. For instance, when $n = 3$ and the cache state is $S_{lru} = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}$, the key in $\texttt{key}[1]$ aligns with the value in $\texttt{val}[2]$, the key in $\texttt{key}[2]$ aligns with the value in $\texttt{val}[1]$, and the key in $\texttt{key}[3]$ aligns with the value in $\texttt{val}[3]$. In essence, this P$_4$LRU cache currently documents the three key-value pairs: $\langle\texttt{key}[1], \texttt{val}[2]\rangle$, $\langle\texttt{key}[2], \texttt{val}[1]\rangle$, and $\langle\texttt{key}[3], \texttt{val}[3]\rangle$.

Update Operation of P$_4$LRU: As outlined in Algorithm 1, we update the P$_4$LRU cache in three steps for an incoming key-value pair $\langle k, v\rangle$. To elucidate the update process of the P$_4$LRU, we offer a detailed step-by-step depiction of the update process of the P$_4$LRU data structure using two examples shown in Figure 3. Assume a P$_4$LRU cache with $n = 5$ initially records five key-value pairs: $\langle K_A, V_A\rangle$, $\langle K_B, V_B\rangle$, $\langle K_C, V_C\rangle$, $\langle K_D, V_D\rangle$, and $\langle K_E, V_E\rangle$, with the cache state in the initial state, that is, $S_{lru} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}$.

Step 1: Maintain Key Array in LRU Order. Firstly, we compare the incoming key k with the most recently used key $\texttt{key}[1]$ in the key array. If these two keys are identical, the key array requires no operation and we end step 1. However, if they differ, we record key k at position $\texttt{key}[1]$ and consider the original $\texttt{key}[1]$ as the evicted key k_e. Subsequently, we compare k with $\texttt{key}[2]$ in the key array. If these keys are identical, we record the evicted key k_e at position $\texttt{key}[2]$ and conclude step 1. Otherwise, we swap k_e and $\texttt{key}[2]$, treating the original $\texttt{key}[2]$ as the new evicted key k_e. This operation continues until a $\texttt{key}[i]$ satisfying $\texttt{key}[i] = k$ is found, stopping step 1, or until the least recently used key $\texttt{key}[n]$ in the key array is evicted. Step 1 is represented in pseudo code on line 1 and lines 8-17 of Algorithm 1.

Example 1: Suppose the incoming packet carries the key-value pair $\langle K_D, V_D'\rangle$. During step 1, we record key K_D at position $\texttt{key}[1]$ and evict the original key K_A from $\texttt{key}[1]$ to $\texttt{key}[2]$. Subsequently, we evict key K_B from $\texttt{key}[2]$ to $\texttt{key}[3]$, evict key K_C from $\texttt{key}[3]$ to $\texttt{key}[4]$, and discover that the key K_D, evicted from $\texttt{key}[4]$, is identical to the incoming key. Following step 1, the key array is updated to $\{K_D, K_A, K_B, K_C, K_E\}$.

Example 2: Suppose the incoming packet carries the key-value pair $\langle K_F, V_F\rangle$. In step 1, we record key K_F at position $\texttt{key}[1]$ and evict the original key K_D from $\texttt{key}[1]$ to $\texttt{key}[2]$. Then, we evict key K_A from $\texttt{key}[2]$ to $\texttt{key}[3]$, evict key K_B from $\texttt{key}[3]$ to $\texttt{key}[4]$, evict key K_C from $\texttt{key}[4]$ to $\texttt{key}[5]$, and finally, we entirely evict key K_E initially recorded in $\texttt{key}[5]$ from the cache. Post step 1, the key array becomes $\{K_F, K_D, K_A, K_B, K_C\}$.

Step 2: update cache state to transition mapping relationship. If we find a $\texttt{key}[i] = k$ in step 1, then our operation on the key

Algorithm 1: Update operation of $\mathsf{P_4LRU}$.

Input: key array $\mathtt{key}[1\cdots n]$, value array $\mathtt{val}[1\cdots n]$, cache state $S_{lru} = \begin{pmatrix} 1 & \cdots & n \\ p_1 & \cdots & p_n \end{pmatrix}$, key-value pair $\langle k, v\rangle$ to be inserted.

1 $\langle k_e, i\rangle$ = `Update_Key_Array`(Key, k);
2 S_{lru} = `Update_Cache_State`(S_{lru}, i);
3 **if** $k_e = k$ **then**
4   `Update_Value`$(Val, S_{lru}(1), v)$;
5 **else**
6   `Replace_Value`$(Val, S_{lru}(1), v)$;
7 **end**
8 **Function** `Update_Key_Array`(Key, k)**:**
9     $k_e \leftarrow k$;
10   **for** $i = 1 \rightarrow n$ **do**
11   Swap$(k_e, \mathtt{key}[i])$;
12   **if** $k_e = k$ **then**
13   **return** $\langle k, i\rangle$;
14   **end**
15   **end**
16   **return** $\langle k_e, n\rangle$;
17 **end**
18 **Function** `Update_Cache_State`(S_{lru}, i)**:**
19   **return** $\begin{pmatrix} 1 & 2 & \cdots & i & i+1 & \cdots & n \\ i & 1 & \cdots & i-1 & i+1 & \cdots & n \end{pmatrix} \times S_{lru}$;
20 **end**
21 **Function** `Update_Value`(Val, i, v)**:**
22     $\mathtt{val}[i]$ =Update$(\mathtt{val}[i], v)$;
23 **end**
24 **Function** `Replace_Value`(Val, i, v)**:**
25     $\mathtt{val}[i] = v$;
26 **end**

array is equivalent to a rotation

$$R = \begin{pmatrix} 1 & 2 & \cdots & i-1 & i & i+1 & \cdots & n \\ 2 & 3 & \cdots & i & 1 & i+1 & \cdots & n \end{pmatrix}$$

on the key array. Accordingly, to update the cache state to describe the mapping relationship between the updated key array and value array, we need to pre-multiply[2] (left-multiply) the cache state by the inverse of rotation R, *i.e.*, update the cache state S_{lru} to

$$R^{-1} \times S_{lru} = \begin{pmatrix} 1 & 2 & \cdots & i & i+1 & \cdots & n \\ i & 1 & \cdots & i-1 & i+1 & \cdots & n \end{pmatrix} \times S_{lru}.$$

If we do not find k in the key array, then we will evict the least recently used key $\mathtt{key}[n]$ at the end of step 1. In this case, we want the incoming key k recorded in $\mathtt{key}[1]$ to reuse the position of the value corresponding to the least recently used key. At this time, we let

$$R = \begin{pmatrix} 1 & \cdots & n-1 & n \\ 2 & \cdots & n & 1 \end{pmatrix}.$$

The pseudo code of step 2 is shown on line 2 and lines 18-20 of Algorithm 1.

[2]Permutation multiplication:

$$\begin{pmatrix} 1 & \cdots & n \\ p_1 & \cdots & p_n \end{pmatrix} \times \begin{pmatrix} 1 & \cdots & n \\ q_1 & \cdots & q_n \end{pmatrix} = \begin{pmatrix} 1 & \cdots & n \\ q_{p_1} & \cdots & q_{p_n} \end{pmatrix}$$

Example 1: After step 1,

$$R = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 1 & 5 \end{pmatrix}.$$

In step 2, we use R^{-1} to update the cache state to

$$S_{lru} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 2 & 3 & 5 \end{pmatrix} \times \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 2 & 3 & 5 \end{pmatrix}.$$

Example 2: After step 1,

$$R = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 1 \end{pmatrix}.$$

In step 2, we use R^{-1} to update the cache state to

$$S_{lru} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 1 & 2 & 3 & 4 \end{pmatrix} \times \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 2 & 3 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 4 & 1 & 2 & 3 \end{pmatrix}.$$

Step 3: find and update the value through the cache state. Whether k is originally recorded in the key array or not, k must be recorded at position $\mathtt{key}[1]$ as the most recently used key after step 1. If the cache state updated in step 2 is $S_{lru} = \begin{pmatrix} 1 & \cdots & n \\ p_1 & \cdots & p_n \end{pmatrix}$, the position of the value corresponding to $\mathtt{key}[1]$ is $\mathtt{val}[p_1]$[3]. If we find a $\mathtt{key}[i] = k$ in step 1, which means the position $\mathtt{val}[p_1]$ records the original value of k, we update it to $\mathtt{OP}(\mathtt{val}[p_1], v)$ with the incoming value v; If we do not find k in the key array, then the position $\mathtt{val}[p_1]$ records the value of the evicted key, and we overwrite it with $\mathtt{OP}(\mathtt{GET}(k), v)$. Where operator $\mathtt{OP}()$ and $\mathtt{Get}()$ are defined in Section 1.1. The pseudo code of step 3 is shown on lines 3-7 and lines 21-26 of Algorithm 1.

Example 1: In step 3, we use the current cache state to locate that the value V_D corresponding to key $\mathtt{key}[1]$ is at position $\mathtt{val}[4]$, and update it to $\mathtt{val}[4] = \mathtt{OP}(V_D, V_D') = V_D''$.

Example 2: In step 3, we use the current cache state to find that the value corresponding to key $\mathtt{key}[1]$ should be recorded at position $\mathtt{val}[5]$, and replace it with $\mathtt{val}[5] = \mathtt{OP}(\mathtt{GET}(F), V_F) = V_F'$.

2.3 Deploying P_4LRU on the Data Plane

Although the $\mathsf{P_4LRU}$ cache we proposed in Section 2.2 meets the limitation of pipeline programming, since the computing resources of the existing programmable data plane are still minimal, it is not possible to deploy the $\mathsf{P_4LRU}$ on the data plane with trivial methods. Specifically, it is almost impossible to calculate the product of permutations on the data plane, and it is necessary to use n tables with a size of $n!$ to record all transitions of a cache state DFA with a parameter of n. However, taking the Tofino [2] chip, one of the most widely used programmable chips, as an example, each stage only supports a table with a size of 16.

Fortunately, when n is small, such as $n = 2$ or $n = 3$, we can cautiously encode the cache state as an integer and use the stateful arithmetic logic unit (ALU) provided by the programmable data

[3]To simplify the notation, we also denote p_1 as $S_{lru}(1)$.

plane to transition the cache state. We describe below how to encode the cache state and embed state transitions into arithmetic calculations in detail. To simplify the notation, we refer to the P_4LRU cache of $n = 2$ and $n = 3$ as P_4LRU_2 and P_4LRU_3.

2.3.1 Details of Implementing P_4LRU_2. For P_4LRU_2 cache, there are only two possible cache states. We encode one state as 0 and the other as 1, *i.e.*,

$$S_{lru} = \begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix} \equiv 0 \qquad S_{lru} = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} \equiv 1.$$

Similarly, there are only two operations we can perform on the key array in step 1. For the incoming key k, if we find that it is the same as key[1], we do not change the cache state S_{lru}, *i.e.*,

$$S_{lru}^{new} = S_{lru};$$

If we find that it is the same as key[2], or that it is not in the cache, we change the cache state to another one, *i.e.*,

$$S_{lru}^{new} = S_{lru} \wedge 1.$$

In the data plane of the programmable switch, we can use registers to store cache states and use stateful ALU associated with the registers to achieve the above arithmetic logic of state transitions. Taking the Tofino chip as an example, each stateful ALU in the Tofino chip can support two arithmetic branches, so that one statefule ALU can support the arithmetic logic of P_4LRU_2 cache.

2.3.2 Details of Implementing P_4LRU_3. For P_4LRU_3 cache, the possible cache states increase to six, and our possible operations on the key array also increase to three. To enable state transitions to be embedded as arithmetic operators, we encode six states as shown in Table 1. The theory of permutation groups in abstract algebra guides our encoding scheme. For example, we encode even permutations as even numbers and odd permutations as odd numbers, which facilitates the embedding of state transitions. We discuss three operations on the key array in turn below.

Table 1: The cache state encoding scheme of P_4LRU_3 cache.

Cache state	Code	Cache state	Code
$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	4	$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}$	1
$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}$	5	$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}$	0
$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}$	2	$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$	3

Operation 1. For the incoming key k, if we find that it is the same as key[1], we do not change the cache state S_{lru}, *i.e.*,

$$S_{lru}^{new} = S_{lru}.$$

Operation 2. For the incoming key k, if we find that it is the same as key[2], we update the cache state with transitions shown in Figure 4. Following the encoding scheme in Table 1, the following arithmetic logic describes these state transitions:

$$S_{lru}^{new} = \begin{cases} S_{lru} \wedge 1 & S_{lru} \geqslant 4 \\ S_{lru} \wedge 3 & S_{lru} \leqslant 3 \end{cases}.$$

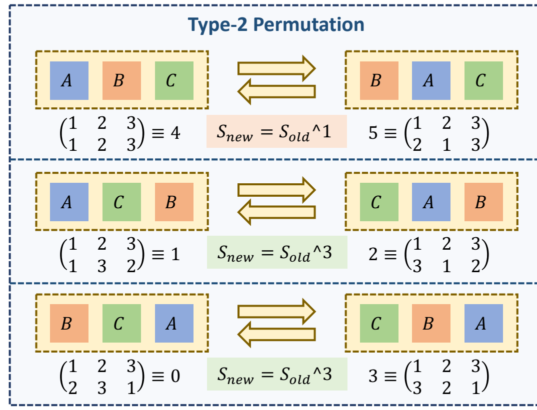


Figure 4: An example of P_4LRU algorithm.

Operation 3. For the incoming key k, if we find that it is the same as key[3], or that it is not in the cache, we switch the cache state with transitions shown in Figure 4. Following the encoding scheme in Table 1, the following arithmetic logic shows these state transitions:

$$S_{lru}^{new} = \begin{cases} S_{lru} - 2 & S_{lru} \geqslant 2 \\ S_{lru} + 4 & S_{lru} \leqslant 1 \end{cases}.$$

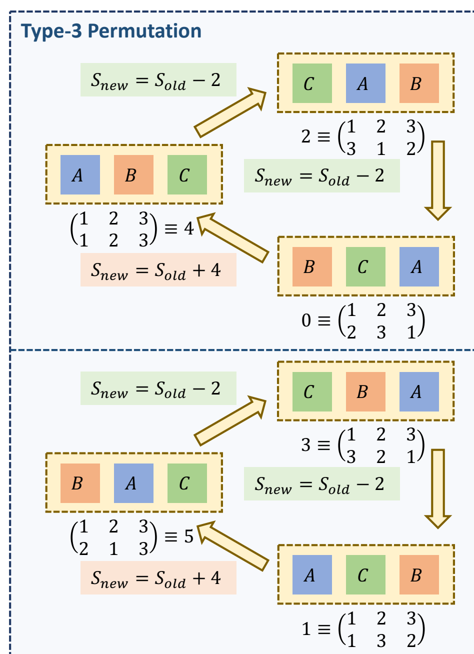


Figure 5: An example of P_4LRU algorithm.

We can use registers to store cache states and use stateful ALUs to achieve state transition as above. Still, take the Tofino chip as an example. Although we cannot use one stateful ALU to cover all five kinds of arithmetic logic due to the limitation of the number of arithmetic branches, we can use three stateful ALUs to implement the arithmetic logic corresponding to operations 1, 2, and 3, respectively. The number of stateful ALUs we use does not exceed the limit that the Tofino chip can support up to four stateful ALUs in one stage.

3 P$_4$LRU BASED IN-NETWORK SYSTEMS

Based on the data plane cache P$_4$LRU$_3$ we proposed in Section 2, we build three feasible in-network systems: LruTable, LruIndex, and LruMon. We introduce them separately in this section.

3.1 LruTable System

LruTable is a data plane network address translation (NAT) system, which translates the virtual destination address in each packet into the real destination address. As shown in Figure 6, LruTable uses a NAT table stored in the control plane to translate addresses and uses an array of 2^{16} P$_4$LRU$_3$ cache units on the data plane to cache some table entries. LruTable uses the following routines to process each packet.

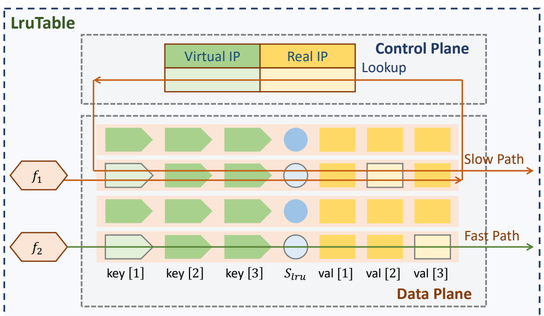


Figure 6: An example of LruTable system.

Processing routine: As shown in Figure 6, LruTable contains a hash function $h(\cdot)$ and an array $P[1\cdots 2^{16}]$ with 2^{16} units, each of which is a P$_4$LRU$_3$ cache unit. For each incoming packet with virtual address va, we use hash function $h(\cdot)$ to locate a cache unit $P[h(va)]$, and insert virtual address va into cache unit $P[h(va)]$.

- **Fast Path:** If the cache hits, *i.e.*, address va is the same as one of $P[h(va)]$.key[1], $P[h(va)]$.key[2], or $P[h(va)]$.key[3], then we update the cache state and take the real address ra corresponding to the virtual address va from position $P[h(va)]$.val$[P[h(va)].S_{lru}(1)]$ in the value array.
- **Slow Path:** If the cache misses, we update the cache state, record a `placeholder` (*e.g.*, `0x00000000` or `0xFFFFFFFF`) in the value array and send the packet to the control plane to lookup the complete NAT table and obtain the real address ra. The packet sent to the control plane carries the real address ra through the data plane again, updates the cache unit $P[h(va)]$, and replaces the `placeholder` previously written with the real address ra carried.

In addition, if the cache hits but the retrieved address is a `placeholder`, the packet still needs to get the real address from the control plane, but it does not need to go through the data plane cache again.

Resource usage: We completely implement LruTable system in the data plane of the Tofino programmable switching chip, and Table 2(a) shows the hardware resource usage of the system. The system takes up one of the four data plane pipelines in total.

3.2 LruIndex System

LruIndex is an in-network query acceleration system. Compared with NetCache [29], which directly caches key-value pairs, LruIndex caches the index (*i.e.*, the 48-bit address in memory) of the key in the database to support values of any length (64 bytes in our system). As shown in Figure 7, LruIndex uses four P$_4$LRU$_3$ cache arrays connected in series to cache indexes, each of which contains 2^{16} P$_4$LRU$_3$ cache units. LruIndex uses the following routines to process each packet.

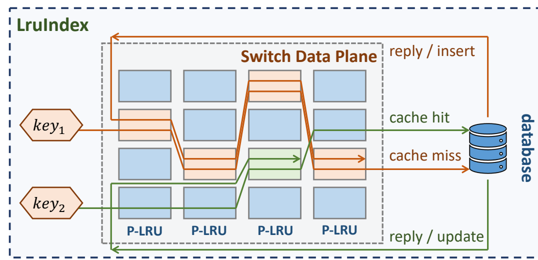


Figure 7: An example of LruIndex system.

Processing routine: As shown in Figure 7, each cache array $P_i[1\cdots 2^{17}]$ is associated with a hash function $h_i(\cdot)$, $1 \leqslant i \leqslant 4$. LruIndex processes query packets sent from the client to the database server and reply packets sent back from the database with different logic. Both packets enable two additional field `cached_flag` and `cached_index` in the packet header.

- **Query packet:** For each incoming query packet with the query key k, we access the four cache arrays in a read-only manner. For the i-th cache array, we use the hash function $h_i(\cdot)$ to locate a cache unit $P_i[h_i(k)]$ and lookup whether key k is cached in the unit. If the i-th array caches the query key k, we write i ($1 \leqslant i \leqslant 4$) to the field `cached_flag` in the packet header and write the index taken from cache unit $P_i[h_i(k)]$ to the field `cached_index`; Otherwise, we write 0 to the field `cached_flag`. The database server checks the field `cached_flag` after receiving each query packet. If the field `cached_flag` is 0, the database relies on a built-in index structure, such as B+ Tree [12], to find the index of key k; Otherwise, the database directly fetches the value of key k from address `cached_index`.
- **Reply packet:** For the query packet whose field `cached_flag` is 0, the database writes the index of key k obtained from the built-in index structure in the field `cached_index` of the reply packet. For each incoming reply packet, LruMon determines whether its field `cached_flag` is 0. If the field `cached_flag` is $i \neq 0$, *i.e.*, the key k was previously cached in the i-th array, then we insert the key k as the most recent used entry into the cache unit $P_i[h_i(k)]$ in the i-th array; Otherwise, if the field `cached_flag` is 0, *i.e.*, the key k is not cached, then we insert the key k and its index `cached_index` into the first array and insert the evicted pair into next three arrays in series: First, we insert the key k and its index into the cache unit $P_1[h_1(k)]$ in the first array, and evict the least recent used entry, key k_1 =

Table 2: Hardware resources used by P$_4$LRU systems.

Resource	Usage	Percentage
Hash Bits	377	7.55%
SRAM	108	11.25%
Map RAM	107	18.58%
TCAM	0	0%
Stateful ALU	7	14.58%
VLIW instr	24	6.25%

(a) LruTable system (one pipeline).

Resource	Usage	Percentage
Hash Bits	2160	10.82%
SRAM	541	14.09%
Map RAM	536	23.21%
TCAM	0	0%
Stateful ALU	40	20.83%
VLIW instr	102	6.64%

(b) LruIndex system (four pipelines).

Resource	Usage	Percentage
Hash Bits	396	3.97%
SRAM	478	24.90%
Map RAM	475	41.23%
TCAM	0	0%
Stateful ALU	17	17.71%
VLIW instr	32	4.17%

(c) LruMon system (two pipelines).

$P_1[h_1(k)]$.key[3] and its index. Then, we insert k_1 and its index as the least recent used entry into cache unit $P_2[h_2(k_1)]$ in the second array, *i.e.*, replace the key $k_2 = P_2[h_2(k_1)]$.key[3] and its index in cache unit $P_2[h_2(k_1)]$ with key k_1 and its index. Next, we replace the key $k_3 = P_3[h_3(k_2)]$.key[3] and its index in cache unit $P_3[h_3(k_2)]$ with key k_2 and its index. Finally, we replace the key $k_4 = P_4[h_4(k_3)]$.key[3] and its index in cache unit $P_4[h_4(k_3)]$ with key k_3 and its index. The key k_4 and its index are completely evicted from the data plane cache.

Series connection technique: LruIndex uses the series connection technique of P$_4$LRU cache arrays. This technique depends on the characteristics of the in-network query acceleration system itself, *i.e.*, each key carried by the packet passes through the data plane twice. This characteristic allows us to access multiple serial-connected cache arrays in a read-only manner and determine which array caches the query key. Our modification of the cache is completely written by the reply packet that accesses the data plane for the second time. Suppose we can only access the data plane once, and all query keys are inserted from the first cache array. In that case, the same key may be recorded in multiple arrays, thus reducing the cache utilization. LruIndex avoids this problem by delaying updating the cache by reply packets.

Resource usage: We completely implement LruIndex system in the data plane of the Tofino programmable switching chip, and Table 2(b) shows the hardware resource usage of the system. The system takes up all four of the four data plane pipelines in total, of which each P$_4$LRU$_3$ cache array takes up one pipeline.

3.3 LruMon System

LruMon is a network telemetry system that aims to classify as much traffic as possible through the data plane. In other words, it wants to measure the size of each flow and make the total size of all flows as large as possible without overestimating the size of any flow. This demand is especially important for traffic billing [18, 34]. As shown in Figure 8, LruMon contains two data structures. The first part is a TowerSketch [56], an enhanced version of the Count-Min sketch [15], which is used to filter the mouse flow. The second part is a P$_4$LRU$_3$ cache array, which contains 2^{17} P$_4$LRU$_3$ units, and uses 32-bit flow fingerprints and 32-bit flow lengths as keys and values. LruMon uses the following routines to process each packet.

Processing routine: As shown in Figure 8, LruMon uses a TowerSketch as a Tower filter, which contains two hash functions $g_1(\cdot)$ and $g_2(\cdot)$, and two counter arrays $C_1[1\cdots 2^{20}]$ and $C_2[1\cdots 2^{19}]$. Array C_1 contains 2^{20} 8-bit counters, and the other array C_2 contains 2^{19} 16-bit counters. Each counter is also associated with an 8-bit timestamp to reset the counter periodically (*e.g.*, every one millisecond). LruMon also contains a cache array $P[1\cdots 2^{17}]$ and a hash function $h(\cdot)$. For each incoming packet with its 5-tuple[4] f as the key and its packet length l as the value, we first use hash functions to locate two counters $C_1[h_1(f)]$ and $C_2[h_2(f)]$.

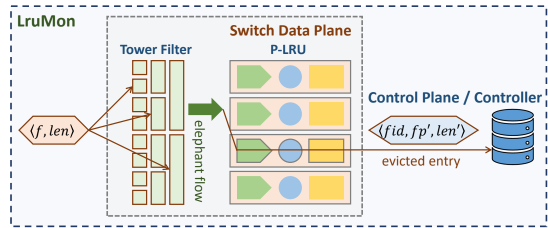
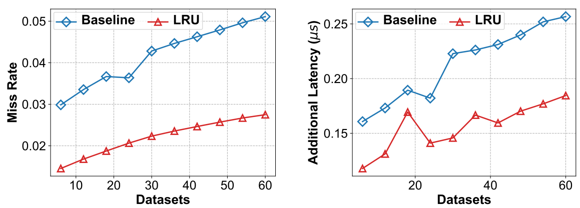


Figure 8: An example of LruMon system.

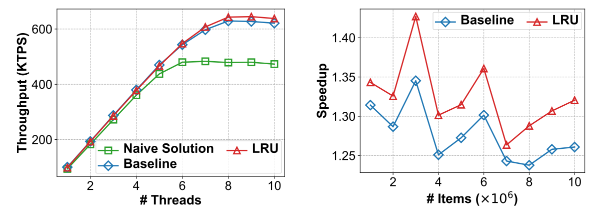
- **Tower filter:** For each counter, we update the timestamp and check whether the counter needs to be reset, and increase the counter according to the packet length *len*. We use
$$\hat{len} = \min\{C_1[h_1(f)], C_2[h_2(f)]\}$$
as the estimated total length of flow f in the current time period[5], and filter out packets belonging to mouse flows with $\hat{len} < L$, where L is a predefined threshold.
- **Cache array:** For packets belonging to elephant flows, we insert them into the cache array. Specifically, we use hash function $h(\cdot)$ to locate a cache unit $P[h(f)]$, use another hash function $fp(\cdot)$ to calculate the 32-bit fingerprint $fp(f)$ of flow f, and insert the fingerprint $fp(f)$ into the cache unit $P[h(f)]$. If the cache hits, *i.e.*, the fingerprint $fp(f)$ is cached in the unit, we update the cache state and increase the value corresponding to the key $fp(f)$ to $P[h(f)]$.val$[P[h(f)].S_{lru}(1)] + len$; If the cache misses, *i.e.*, the fingerprint $fp(f)$ is not cached in the unit, we update the cache state, set the value corresponding to key $fp(f)$ to len, and evict the original key $fp' = P[h(f)]$.key[3] and its length len'.
- **Remote analyzer:** When the cache misses, we also generate an entry $\langle f, fp', len' \rangle$ to upload to the remote analyzer. The analyzer maintains a table T_{fp} with 5-tuples and corresponding fingerprints, and a table T_{len} with 5-tuples and corresponding lengths. If the flow f is not recorded in table T_{fp} and table T_{len}, we add an entry $\langle f, fp(f) \rangle$ in table T_{fp} and an entry $\langle f, 0 \rangle$ in table

[4] ⟨source IP, source port, destination IP, destination port, protocol⟩
[5] For the operation details of TowerSketch, please refer to [56].



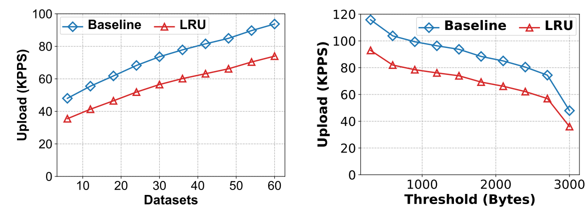
(a) Miss rate *v.s.* concurrency. (b) Latency *v.s.* concurrency.

Figure 9: Testbed experiment of LruTable system.



(a) Throughput *v.s.* # threads. (b) Speedup *v.s.* # items.

Figure 10: Testbed experiment of LruIndex system.



(a) Upload *v.s.* concurrency. (b) Upload *v.s.* threshold.

Figure 11: Testbed experiment of LruMon system.

T_{len}. We use fingerprint fp' to find the 5-tuple f' corresponding to the fp' and increase the length of flow f' in table T_{len} by len'.

All packets passing the Tower filter are buffered in the data plane cache and upload to the remote analyzer further, so that LruMon can measure these flows with no error. Although different data plane caches do not affect the measurement accuracy, *a better cache can reduce the number of entries uploaded from the data plane to the analyzer, thereby reducing the pressure on the analyzer.*

Resource usage: We completely implement LruMon system in the data plane of the Tofino programmable switching chip, and Table 2(c) shows the hardware resource usage of the system. The system takes up two of the four data plane pipelines in total, of which the Tower filter takes up one pipeline and the P$_4$LRU$_3$ cache array takes up one pipeline.

4 EVALUATION

In this section, we first use a Flnet S9280 Tofino switch with four pipelines to build a testbed and evaluate the three P$_4$LRU systems, respectively. Then we further analyze the performance of the three systems through simulation on the CPU platform.

Datasets: We use CAIDA 2018 [3], an anonymous IP trace dataset, to construct a synthetic dataset CAIDA$_n$: We divide the one-hour CAIDA 2018 trace into 60 one-minute datasets and take $\frac{1}{n}$ minutes from the first n datasets to form a synthetic dataset. The reason for using CAIDA$_n$ is to change the concurrency of the dataset. Each dataset contains about 2.6×10^7packets. From CAIDA$_1$ to CAIDA$_{60}$, the number of flows changes from 1.3×10^6 to 2.4×10^6, and the maximum number of concurrent flows changes from 1.5×10^5 to 5.8×10^5.

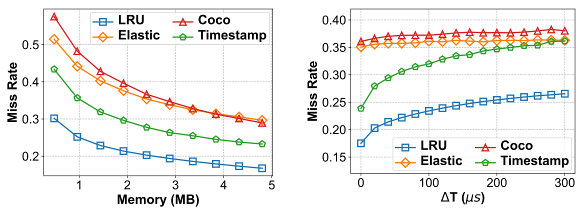
4.1 Testbed Experiments

In this section, we show the performance difference between using P$_4$LRU cache and hash table based cache in the three systems. In all figures, we use *LRU* to refer to the system using P$_4$LRU cache, *Baseline* to refer to the system using hash table based cache, and *Naive Solution* to the solution not using cache. We use about 6000 lines of P4 code to implement all systems.

LruTable system (Figure 9): We use the DPDK [4] driver to replay the CAIDA$_n$ datasets on the sender client. The packets pass through the Tofino switch, perform address translation, and arrive at the receiver client. We show the fast-path miss rate and additional end-to-end latency compared with direct forwarding without address translation. As shown in Figure 9(a), with the increase of traffic concurrency, the miss rate of the system using P$_4$LRU increases from 1.4% to 2.7%, and that of the system using the baseline solution increases from 3.0% to 5.1%. In summary, P$_4$LRU is 2.14× better than the baseline solution at most. As shown in Figure 9(b), with the increase of traffic concurrency, the additional latency of the system using P$_4$LRU increases from 0.11 μs to 0.18 μs, and that of the system using baseline solution increases from 0.16 μs to 0.26 μs. In summary, at most, P$_4$LRU is 1.35× better than the baseline solution.

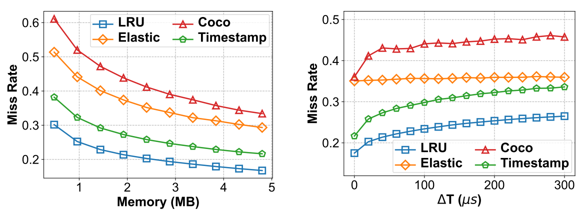
LruIndex system (Figure 10): We use the DPDK driver to implement the query thread and database server and use YCSB [13] benchmark to evaluate the database. We generate the query transaction set according to the Zipf distribution [44] of $\alpha = 0.9$, and evaluate the query performance of the database. As shown in Figure 10(a), when the database contains 1×10^6 items, with the increase of the number of query threads, the query throughput of the system using P$_4$LRU increases from 98.5 KTPS (Kilo Transactions Per Second) to 644.8 KTPS, and that of the system using baseline solution increases from 100.3KTPS to 629.2 KTPS. In summary, P$_4$LRU is 1.03× better than the baseline solution at most. As shown in Figure 10(b), when there are 8 query threads, with the increase of the database scale, the throughput speedup compared with naive solution of the system using P$_4$LRU varies from 1.26 to 1.36, and that of the system using baseline solution varies from 1.23 to 1.34. In summary, at most, P$_4$LRU is 1.08× better than the baseline solution.

LruMon system (Figure 11): We use the DPDK driver to replay the CAIDA$_n$ datasets at 10Gbps on the sender client. The packets are forwarded to the Tofino switch and measured, and the measurement entries generated by the data plane are forwarded to a remote analyzer. We show the generation rate of packets with entries uploaded to the analyzer from the data plane. As shown in Figure



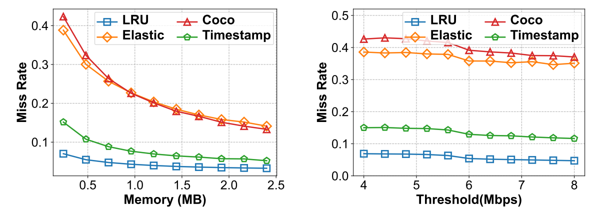
(a) Miss rate *v.s.* memory. (b) Miss rate *v.s.* ΔT.

Figure 12: Comparative experiment of LruTable.



(a) Miss rate *v.s.* memory. (b) Miss rate *v.s.* ΔT.

Figure 13: Comparative experiment of LruIndex.



(a) Miss rate *v.s.* memory. (b) Miss rate *v.s.* ΔT.

Figure 14: Comparative experiment of LruMon.

11(a), when the filter threshold is 1500 bytes and the reset period is 10 *ms*, with the increase of traffic concurrency, the upload rate of system using P_4LRU increases from 35.5 KPPS (Kilo Packets Per Second) to 74.0 KPPS, and that of system using baseline solution increases from 48.0 KPPS to 93.7 KPPS. In summary, P_4LRU is 1.35× better than the baseline solution at most. As shown in Figure 11(b), with the increase of filter threshold, the upload rate of system using P_4LRU decreases from 92.9 KPPS (Kilo Packets Per Second) to 36.0 KPPS, and that of the system using the baseline solution decreases from 115.8 KPPS to 47.9 KPPS. In summary, at most, P_4LRU is 1.33× better than the baseline solution.

Analysis: The reason why P_4LRU improves LruIndex less than the other two systems is that the query transaction set is randomly generated, so the continuity of the same query key in time is weaker than that of the CAIDA dataset.

4.2 Simulation Experiments

In this section, we use the CPU platform to simulate the performance of P_4LRU cache under more variable conditions. We use *LRU similarity* to evaluate the similarity between LRU replacement policy and other replacement policies: assuming that the cache capacity is n, for each entry evicted from the cache, if the ranking of its last access time is k, its relative ranking is $\frac{k}{n}$. For an ideal LRU cache, the relative ranking is always 1. We define LRU similarity as the average relative ranking of all evicted entries. We use $CAIDA_{60}$ dataset by default and rescale the time of the dataset to one second. In all figures, we use LRU_{IDEAL} to represent the ideal LRU cache, and P_4LRU_1 to represent the cache using the hash table, and P_4LRU_2 and P_4LRU_3 are defined in Section 2.3.

4.2.1 Comparative Experiments

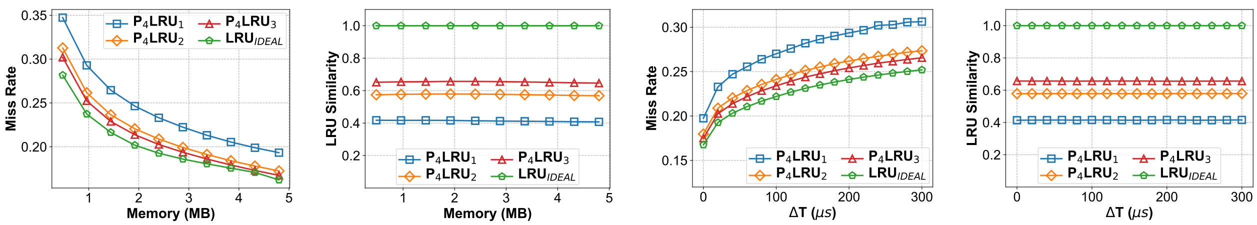
LruTable system (Figure 12): As shown in Figure 12(a), we change the memory used by the cache and evaluate the cache miss rate. The miss rate of using CocoSketch replacement policy and Elastic replacement policy is similar, while that of using our P_4LRU_3 can be reduced by up to 27.4% and 21.3%. As shown in Figure 12(b), we change the slow path latency ΔT, and evaluate the cache miss rate. The miss rate of using CocoSketch replacement policy and Elastic replacement policy is similar, while that of using our P_4LRU_3 can be reduced by up to 17.5% and 18.5%.

LruIndex system (Figure 13): As shown in Figure 13(a), we change the memory used by the cache and evaluate the cache miss rate. The miss rate of using CocoSketch replacement policy is higher than that of using Elastic replacement policy, while that of using our P_4LRU_3 can be reduced by up to 31.0% and 21.2%. As shown in Figure 13(b), we change the query latency ΔT, and evaluate the cache miss rate. The miss rate of using CocoSketch replacement policy is higher than that of using Elastic replacement policy, while that of using our P_4LRU_3 can be reduced by up to 17.6% and 18.6%.

4.2.2 Parameter Experiments

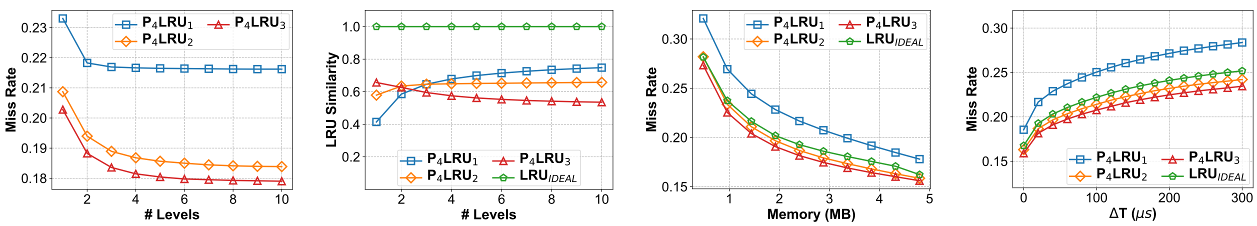
LruTable system (Figure 15): As shown in Figure 15(a) and 15(b), we change the memory used by the cache and evaluate the cache miss rate and LRU similarity. For the miss rate, P_4LRU_3 cache is always the closest to the ideal LRU cache. When the memory is considerable, P_4LRU_2, P_4LRU_3, and ideal LRU cache have similar performance. For the similarity, P_4LRU_3 cache is always the highest and almost does not change with the memory. As shown in Figure 15(c) and 15(d), we change the slow path latency ΔT, and evaluate the cache miss rate and LRU similarity. For the miss rate, P_4LRU_3 cache is always the closest to the ideal LRU cache. For the similarity, P_4LRU_3 cache is always the highest, and almost does not change with the latency.

LruIndex system (Figure 16): As shown in Figure 16(a) and 16(b), we change the number of connection levels used in serial connection technique, and evaluate the cache miss rate and LRU similarity. For the miss rate, P_4LRU_3 cache is always the lowest, and P_4LRU_2 and P_4LRU_3 are significantly better than P_4LRU_1. Interestingly, with the increase of the number of levels, the similarity of P_4LRU_2 and P_4LRU_1 are increasing, while the similarity of P_4LRU_3 is decreasing. Since LRU similarity often represents universality, it means that for CAIDA dataset, more levels mean better performance, but not necessarily for other datasets. Therefore, we use four levels as the default settings to achieve a low miss rate with sufficient LRU similarity. As shown in Figure 16(c) and 16(d), we change the memory and query latency ΔT of the database server, and evaluate



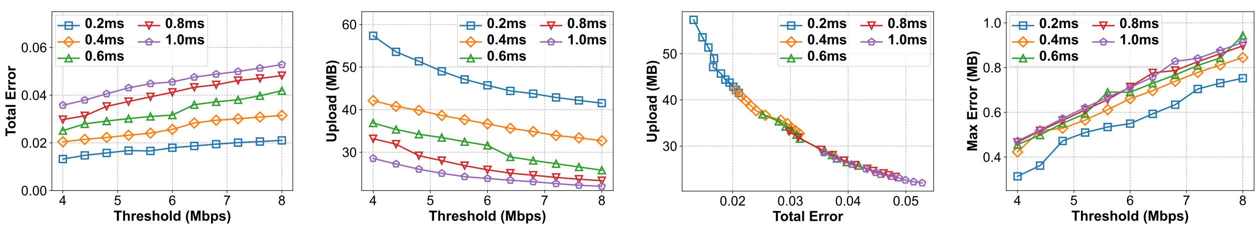
(a) Miss rate *v.s.* memory (b) Similarity *v.s.* memory (c) Miss rate *v.s.* ΔT (d) Similarity *v.s.* ΔT

Figure 15: Simulation experiment of LruTable system.



(a) Miss rate *v.s.* # connection levels (b) Similarity *v.s.* # connection levels (c) Miss rate *v.s.* memory (d) Miss rate *v.s.* ΔT

Figure 16: Simulation experiment of LruIndex system.



(a) Total error *v.s.* threshold (b) Upload *v.s.* threshold (c) Upload *v.s.* total error (d) Max error *v.s.* threshold

Figure 17: Simulation experiment of LruMon system.

the cache miss rate. The experimental results show that P$_4$LRU$_3$ cache is always the closest to the ideal LRU cache.

LruMon system (Figure 17): As shown in Figure 17(a) and 17(b), we use P$_4$LRU$_3$ and change the bandwidth threshold and reset period of the Tower filter, and evaluate the total error rate and upload volume. The bandwidth threshold is the ratio of the filter threshold to the reset period, and the total error rate is the ratio of the total underestimation error to the total number of bytes. The experimental results show that the shorter the reset period is, the lower the error is, and the larger the upload volume is. This is because using a larger reset period will result in filtering out burst traffic. Interestingly, as shown in Figure 17(c), no matter what the reset period is, when the total error is the same, the upload volume is almost the same. As shown in Figure 17(d), we also evaluate the maximum flow-level error. The experimental results show that, the maximum error will not exceed the filter threshold.

5 RELATED WORK

In this section, we first introduce typical cache replacement policies in § 5.1, then we summarize existing data plane cache solutions in § 5.2.

5.1 Cache Replacement Policies

The ideal cache replacement algorithm would always be to discard the item that will not be needed for the longest time in the future, which is impossible to implement because we cannot predict the future. Fortunately, many algorithms have been invented to approach the ideal solution. We roughly summarized existing algorithms into three categories: recency-based policies, frequency-based policies, and hybrid policies.

Recency-based policies discard an item based on its latest reference within its lifetime. For example, the well-known LRU policy evicts the least recently used items. Typical variants of LRU include Early Eviction LRU (EELRU) [52], Segmented LRU (Seg-LRU) [22], RRIP [26], and more [17, 27, 45, 54]. By contrast, the MRU (Most Recently Used) policy [10] evicts most recently used items. It was reported that for random access patterns and cyclic access patterns, MRU policies have more hits than LRU policies due to their tendency to retain older data [16]. In addition, there is also a particular class of recency-based policies [25, 47] that artificially extends the lifetimes of some items by storing them in an auxiliary buffer.

Frequency-based policies use access frequency to replace items, where items accessed more frequently are preferentially cached over items accessed less frequently. LFU (Least Frequently Used) [11] is the most strightforward frequency-based policy, which associates a frequency counter to each item. Unfortunately, frequency-based policies can not always catch up with the changes in application phases: an item with high frequency from a previous phase will remain cached in the new phase for a while even if the item is no longer being accessed. Many solutions have been proposed to address this issue. The key idea of these solutions is to combine frequency information with recency information to age old items. Typical solutions include FBR [49], LRFU [32], and more [19, 41, 50].

Hybrid policies dynamically change the replacement algorithms according to the current working set. They need to accurately identify the optimal policy with low hardware overhead to manage multiple policies. Typical implementation of hybrid policies includes ARC (Adaptive Replacement Cache) [24, 39, 55], Set Dueling [45, 46], and more [28].

5.2 Data Plane Cache Solutions

Existing data plane cache solutions can be roughly divided into three categories: hash based solutions, frequency based solutions, and non-real time solutions. For other solutions, please refer to [30, 35, 37, 43, 53].

Hash based solutions record recent entries by maintaining a hash table on data plane. Typical work includes NetSeer [59], Pegasus [33], SpiderMon [36], and Jaqen [38]. For example, NetSeer [59] maintain a simple hash table in the data plane. When two items collide into the same entry, it discards the old entry to make room for the new item. SpiderMon [36] uses a similar technique to maintain telemetry information in data plane. The performance of these work is significantly influenced by hash collisions. When frequently accessed items collides into the same entry, the hit rate will drastically degrade.

Frequency based solutions maintain the information of frequent items on the data plane. These work devise elegant data structures to efficiently record large flows and filter small flows, and deploy the data structures on programmable data plane. Their design goal is to simplify the access to stateful memory to conform with RMT limitations and achieves high accuracy in finding elephant flows. Although they do not explicitly claim they are data plane caches, they can be treated as a cache that preserves frequent items and evicts infrequent items. Typical work includes HashPipe [51], PRECISION [7], CocoSketch [58], and Elastic sketch [57]. However, a significant drawback of frequency based solutions is that the frequent items will be kept for a long while, even though they are no longer being accessed.

Non-real time solutions update the cache in a delayed manner. For example, NetCache [29] maintains a table of frequent items on the data plane, and periodically exchanges frequent items into the cache through the control plane. BeauCoup [9] maintains the last accessed time of each entry, and periodically ages out the expired entries to make room for new items. A common shortcoming of non-real time solutions is that they have relatively long update latency, so they cannot catch up with the sudden changes of the workload.

6 CONCLUSION

In this paper, we first analyze why classical LRU cache cannot be implemented on the data plane of programmable switch. Then we propose a pipelined version of the LRU implementation, namely P_4LRU. In P_4LRU, we eliminate the circular access to data by introducing an deterministic finite automaton (DFA) named S_{lru}, and we discuss how to use the stateful arithmetic logical unit (ALU) of programmable ASICs to achieve the storage and state transition of DFA in detail. Based on P_4LRU cache, we design three kinds of in-network systems: a data plane network address translation (NAT) system LruTable, an in-network database query acceleration system LruIndex, and a data plane network telemetry system LruMon. We extensively evaluate the performance of the three systems. Experimental results show that compared to the baseline replacement policy, P_4LRU cache achieves an end-to-end performance improvement of up to 35%, 8.2%, and 35% on our three systems. All related codes of the three systems are available at GitHub [1].

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their valuable suggestions. This work is supported by National Key R&D Program of China (No. 2022YFB2901504), and National Natural Science Foundation of China (NSFC) (No. U20A20179).

REFERENCES

[1] All related codes of our three system. https://github.com/P4-LRU/P4-LRU.

[2] Barefoot tofino and tofino 2 switches. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[3] The caida anonymized internet traces. http://www.caida.org/data/overview/.

[4] Data plane development kit. http://doc.dpdk.org/guides-18.02/.

[5] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proceedings of the 42nd annual Southeast regional conference*, pages 267–272, 2004.

[6] Waleed Ali, Siti Mariyam Shamsuddin, Abdul Samad Ismail, et al. A survey of web caching and prefetching. *Int. J. Advance. Soft Comput. Appl*, 3(1):18–44, 2011.

[7] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 313–323. IEEE, 2018.

[8] Muhammad Bilal and Shin-Gak Kang. A cache management scheme for efficient content eviction and replication in cache networks. *IEEE Access*, 5:1692–1701, 2017.

[9] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 226–239, 2020.

[10] Hong-Tai Chou and David J DeWitt. An evaluation of buffer management strategies for relational database systems. *Algorithmica*, 1(1):311–336, 1986.

[11] Edward Grady Coffman and Peter J Denning. *Operating systems theory*, volume 973. prentice-Hall Englewood Cliffs, NJ, 1973.

[12] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[13] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[14] Fernando J Corbato. A paging experiment with the multics system. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.
[15] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.
[16] Shaul Dar, Michael J Franklin, Bjorn T Jonsson, Divesh Srivastava, Michael Tan, et al. Semantic data caching and replacement. In *VLDB*, volume 96, pages 330–341, 1996.
[17] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V Veidenbaum. Improving cache management policies using dynamic reuse distances. In *2012 45Th annual IEEE/ACM international symposium on microarchitecture*, pages 389–400. IEEE, 2012.
[18] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 323–336, 2002.
[19] Priyank Faldu and Boris Grot. Leeway: Addressing variability in dead-block prediction for last-level caches. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 180–193. IEEE, 2017.
[20] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 371–384, 2013.
[21] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.
[22] Hongliang Gao and Chris Wilkerson. A dueling segmented lru replacement algorithm with adaptive bypassing. In *JWAC 2010-1st JILP Worshop on Computer Architecture Competitions: cache replacement Championship*, 2010.
[23] Herodotos Herodotou. Autocache: employing machine learning to automate caching in distributed file systems. In *2019 IEEE 35th international conference on data engineering workshops (ICDEW)*, pages 133–139. IEEE, 2019.
[24] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. Improving flash-based disk cache with lazy adaptive replacement. *ACM Transactions on Storage (TOS)*, 12(2):1–24, 2016.
[25] Akanksha Jain and Calvin Lin. Back to the future: Leveraging belady's algorithm for improved cache replacement. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 78–89. IEEE, 2016.
[26] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). *ACM SIGARCH computer architecture news*, 38(3):60–71, 2010.
[27] Daniel A Jiménez. Insertion and promotion for tree-based pseudolru last-level caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 284–296, 2013.
[28] Daniel A Jiménez and Elvira Teran. Multiperspective reuse prediction. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 436–448. IEEE, 2017.
[29] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136, 2017.
[30] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.
[31] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 90–106, 2020.
[32] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, 50(12):1352–1361, 2001.
[33] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan RK Ports. Pegasus: Tolerating skewed workloads in distributed storage with {In-Network} coherence directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 387–406, 2020.
[34] Tao Li, Shigang Chen, and Yibei Ling. Per-flow traffic measurement through randomized counter sharing. *IEEE/ACM Transactions on Networking*, 20(5):1622–1634, 2012.
[35] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G Andersen, and Michael J Freedman. Be fast, cheap and in control with {SwitchKV}. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 31–44, 2016.
[36] Kang Ling, Yuntang Liu, Ke Sun, Wei Wang, Lei Xie, and Qing Gu. Spidermon: Towards using cell towers as illuminating sources for keystroke monitoring. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 666–675. IEEE, 2020.
[37] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. {DistCache}: Provable load balancing for {Large-Scale} storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, 2019.
[38] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. Jaqen: A {High-Performance} {Switch-Native} approach for detecting and mitigating volumetric {DDoS} attacks with programmable switches. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3829–3846, 2021.
[39] Nimrod Megiddo and Dharmendra S Modha. Outperforming lru with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, 2004.
[40] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.
[41] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.
[42] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.
[43] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open {vSwitch}. In *12th USENIX symposium on networked systems design and implementation (NSDI 15)*, pages 117–130, 2015.
[44] David MW Powers. Applications and explanations of zipf's law. In *New methods in language processing and computational natural language learning*, 1998.
[45] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News*, 35(2):381–391, 2007.
[46] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. Set-dueling-controlled adaptive insertion for high-performance caching. *IEEE micro*, 28(1):91–98, 2008.
[47] Kaushik Rajan and Govindarajan Ramaswamy. Emulating optimal replacement with a shepherd cache. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 445–454. IEEE, 2007.
[48] John T Robinson and Murthy V Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 134–142, 1990.
[49] John T Robinson and Murthy V Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 134–142, 1990.
[50] Dennis Shasha and T Johnson. 2q: A low overhead high performance buffer replacement algorithm. In *Very large database systems conference 1994, September 1994*, 1994.
[51] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, pages 164–176, 2017.
[52] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. Eelru: simple and effective adaptive page replacement. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):122–133, 1999.
[53] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with {* Flow}. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 823–835, 2018.
[54] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. Ship: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 430–441, 2011.
[55] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David HC Du. {AC-Key}: Adaptive caching for {LSM-based}{Key-Value} stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 603–615, 2020.
[56] Kaicheng Yang, Yuanpeng Li, Zirui Liu, Tong Yang, Yu Zhou, Jintao He, Tong Zhao, Zhengyi Jia, Yongqiang Yang, et al. Sketchint: Empowering int with towersketch for per-flow per-switch measurement. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–12. IEEE, 2021.
[57] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.
[58] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. Cocosketch: High-performance sketch-based measurement over arbitrary partial key query. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 207–222, 2021.
[59] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. Flow event telemetry on programmable data plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 76–89, 2020.