P_4LRU: Towards An LRU Cache Entirely in Programmable Data Plane

Yikai Zhao[†] Peking University Beijing, China
Wenrui Liu[†] Peking University Beijing, China
Fenghao Dong[†] Peking University Beijing, China
Tong Yang[†] Peking University Beijing, China
Yuanpeng Li[†] Peking University Beijing, China
Kaicheng Yang[†] Peking University Beijing, China
Zirui Liu[†] Peking University Beijing, China
Zhengyi Jia[‡] Huawei Cloud Computing Technologies Co Ltd Beijing, China
Yongqiang Yang[‡] Huawei Cloud Computing Technologies Co Ltd Beijing, China

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 P_4LRU, 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**.

KEYWORDS

Data Plane Cache; Least Recently Used; Programmable Switches

*Yikai Zhao (zyk@pku.edu.cn) and Wenrui Liu (liuwenrui@pku.edu.cn) 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

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, 14 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 [32], 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) [41], LFU (Least Frequently Used) [49], LFRU, [8, 33] 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, 24]. 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 [43] 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 [61] 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 [59], Elastic [58], and HashPipe [52] 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.

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

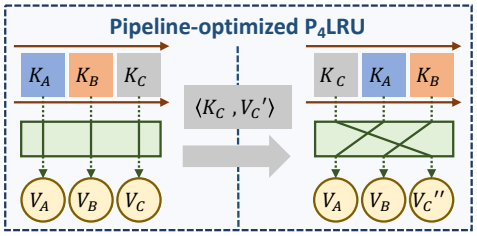


Figure 1: A simple example of our P_4LRU ($n = 3$).

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 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.

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

(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$_4$LRU: (1) **LruTable**, a Network Address Translation (NAT) system [32] that employs P$_4$LRU 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$_4$LRU 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 [57] to filter minor flows and employs P$_4$LRU 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$_4$LRU, that complies with pipeline programming constraints.
- We investigate the actual deployment of P$_4$LRU 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$_4$LRU 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$_4$LRU

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 $\langle k, v, t \rangle$. 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.

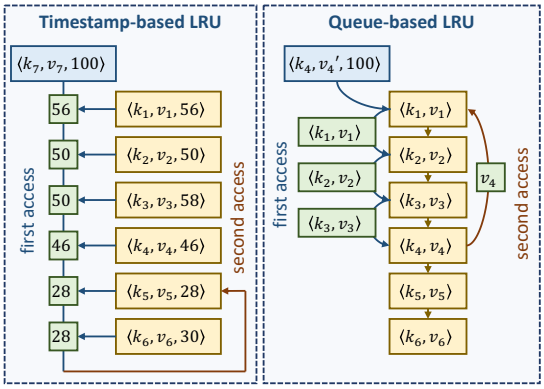


Figure 2: Examples of two LRU implementations.

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 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

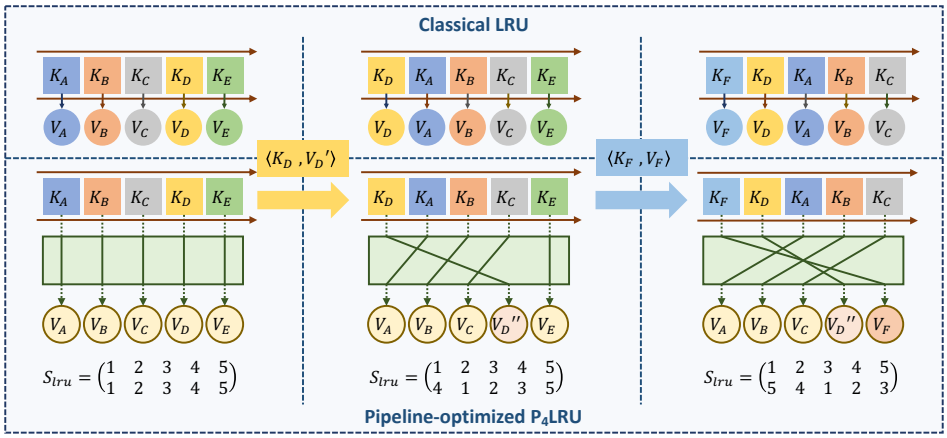


Figure 3: An example of our in-network P$_4$LRU algorithm.

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.

Algorithm 1: Update operation of P$_4$LRU.

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

```
1  ⟨k_e, i⟩ = 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 ← k;
10 |  for i = 1 → n do
11 |  |  Swap(k_e, key[i]);
12 |  |  if k_e = k then
13 |  |  |  return ⟨k, i⟩;
14 |  |  end
15 |  end
16 |  return ⟨k_e, n⟩;
17 end
18 Function Update_Cache_State(S_lru, i):
19 |  return (1 2 ··· i i+1 ··· n / i 1 ··· i−1 i+1 ··· n) × S_lru;
20 end
21 Function Update_Value(val[], i, v):
22 |  val[i] =Update(val[i], v);
23 end
24 Function Replace_Value(val[], i, v):
25 |  val[i] = v;
26 end
```

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 key[1] and evict the original key K_D from key[1] to key[2]. Then, we evict key K_A from key[2] to key[3], evict key K_B from key[3] to key[4], evict key K_C from key[4] to key[5], and finally, we entirely evict key K_E initially recorded in 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. During Step 1, if we find key$[i] = k$, our operations on the key array are equivalent to a rotation

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

of the key array. As a consequence, to update the cache state and depict the mapping relationship between the updated key and value arrays, we need to pre-multiply[2] (left-multiply) the cache state by the inverse of rotation R. That is, we 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 k is not found in the key array, then we will evict the least recently used key key$[n]$ at the end of Step 1. In this situation, we want the incoming key k recorded in key[1] to reuse the position of the value corresponding to the least recently used key. In such a case, 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.

Example 1: After completing Step 1, we find that

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

Proceeding to Step 2, we use the inverse of R to update the cache state as follows:

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

Example 2: After completing Step 1, we find that

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

Proceeding to Step 2, we use the inverse of R to update the cache state as follows:

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

Step 3: Find and Update the Value through the Cache State. Whether the key k was originally present in the key array or not, it will be recorded at position 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 key[1] is val$[p_1]$[3]. If we find a key$[i] = k$ in Step 1, indicating that the position val$[p_1]$ holds the original value of k, we update it to Update(val$[p_1], v$) with the incoming value v. If we don't find k in the key array, then the position val$[p_1]$ holds the value of the evicted key, and we overwrite it with v. The operator Update() is contingent on the specific usage scenario of the cache. It could be an operation that accumulates or aggregates two values (for a write-cache), or an operation that retains one of the values (for a read-cache). These scenarios are discussed in detail in Section 3. The pseudocode for 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 identify that the value V_D, which corresponds to the key key[1], is located at the position val[4]. We then update this to val[4] = UPDATE$(V_D, V'_D) = V''_D$.

Example 2: In Step 3, we use the current cache state to determine that the value corresponding to the key key[1] should be stored at position val[5]. We then replace this with val[5] = V_F.

2.3 Deploying P$_4$LRU on the Data Plane

Although the P$_4$LRU cache we proposed in Section 2.2 meets the constraints of pipeline programming, the limited computing resources of the current programmable data plane make its deployment on the data plane nontrivial. More specifically, computing the product of permutations on the data plane is virtually impossible, and to record all transitions of a cache state DFA with a parameter of n, we need to use n tables, each of size $n!$. However, considering the Tofino chip [2] – one of the most widely used programmable chips – as an example, we are required to store the cache state in registers. When operating registers through the stateful arithmetic logic unit (ALU), we can only access a table of size 16, which is typically used to support approximate floating-point operations. Although Tofino provides sufficiently large flow tables at each stage, it currently does not support the logic of "`read register - lookup table - write register`".

Fortunately, when n is small, such as $n = 2$ or $n = 3$, we can carefully encode the cache state as an integer and utilize the stateful ALU provided by the programmable data plane to transition the cache state. We will further detail how to encode the cache state and embed state transitions into arithmetic calculations. For simplicity of notation, we will refer to the P$_4$LRU cache of $n = 2$ and $n = 3$ as P$_4$LRU$_2$ and P$_4$LRU$_3$.

2.3.1 *Details of Implementing P$_4$LRU$_2$.*

For a P$_4$LRU$_2$ cache, there are only two possible cache states. We can encode one state as 0 and the other as 1. For instance,

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

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

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

[2]Permutation multiplication:

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

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

Table 1: Encoding scheme for the cache state of P_4LRU_3.

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

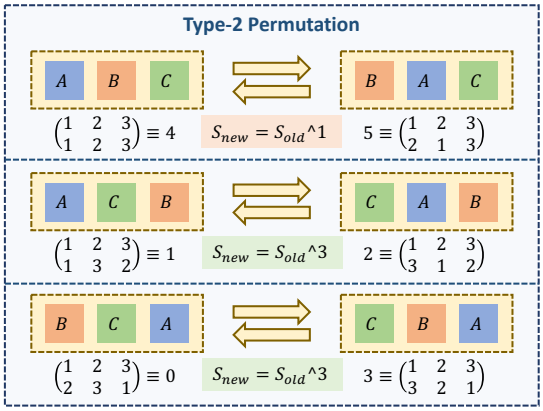


Figure 4: An example of P_4LRU algorithm.

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

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

On the programmable switch's data plane, we can use registers to store cache states and utilize the stateful ALU[4] associated with the registers to accomplish the aforementioned arithmetic logic of state transitions. For example, each stateful ALU in the Tofino chip can support two arithmetic branches, meaning that one stateful ALU can accommodate the arithmetic logic of a P_4LRU_2 cache.

2.3.2 *Details of Implementing P_4LRU_3.*

For a P_4LRU_3 cache, the potential cache states increase to six, and the possible operations on the key array also rise to three. To facilitate state transitions as arithmetic operations, we encode the six states as displayed in Table 1. The theory of permutation groups in abstract algebra informs our encoding scheme. For instance, we encode even permutations as even numbers and odd permutations as odd numbers, which simplifies the embedding of state transitions. Below, we discuss the three operations on the key array.

Operation 1. For the incoming key k, if we find that it matches key[1], we do not alter the cache state S_{lru}. That is,

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

Operation 2. For the incoming key k, if we find that it matches key[2], we modify the cache state according to the transitions shown in Figure 4. According to the encoding scheme in Table 1, the following arithmetic logic describes these state transitions:

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

[4]A stateful ALU can be shared by multiple cache states within a single stage.

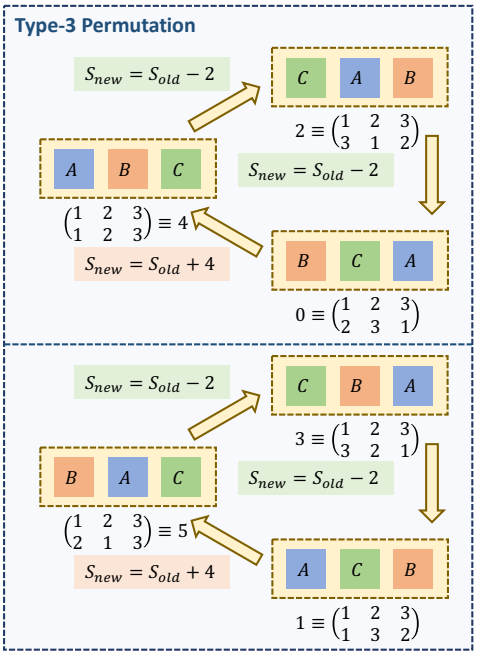


Figure 5: An example of P_4LRU algorithm.

Operation 3. For the incoming key k, if we find that it matches key[3], or that it is not in the cache, we switch the cache state according to the transitions shown in Figure 4. Following the encoding scheme in Table 1, the subsequent arithmetic logic demonstrates these state transitions:

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

We can use registers to store cache states and employ stateful ALUs to execute the above-mentioned state transitions. Let's continue to use the Tofino chip as an illustrative example. Although one stateful ALU cannot cover all five types of arithmetic logic due to the limitation on the number of arithmetic branches, we can utilize three stateful ALUs to implement the arithmetic logic corresponding to operations 1, 2, and 3 respectively. Importantly, the number of stateful ALUs employed does not exceed the maximum that the Tofino chip can support in one stage, which is up to four stateful ALUs.

2.3.3 *From the Perspective of Group Theory.*

It is established that all cache states of P_4LRU_n housing n key-value pairs effectively represent the n-element permutation group S_n. The multiplication operation, associated with the n possible transitions, is genuinely a subset of the group's multiplication operation. This raises an intriguing question: which groups and their corresponding operations are implementable on the programmable data plane? Considering that the data plane's registers can store integers and the affiliated stateful ALU is equipped to handle addition and subtraction, the n-element cyclic group C_n can indeed be materialized on the data plane. To achieve this, the set $\{0, 1, \cdots, n-1\}$ can be employed to represent the elements $\{e, g, \cdots, g^{n-1}\}$ of the group C_n, with integer addition symbolizing element multiplication.

Delving into more complex groups, if both groups H and K are encodable on the data plane, and (1) given that group $G = H \times K$,

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

Resource	Percentage		
	LruTable	LruIndex	LruMon
Hash Bits	7.55%	10.82%	3.97%
SRAM	11.25%	14.09%	24.90%
Map RAM	18.58%	23.21%	41.23%
TCAM	0%	0%	0%
Stateful ALU	14.58%	20.83%	17.71%
VLIW instr	6.25%	6.64%	4.17%

it's plausible to encode G onto the data plane. For an element $g = (h, k) \in G$, where $h \in H$ and $k \in K$, both h and k can be encoded separately. When multiplying elements $g_1 = (h_1, k_1)$ and $g_2 = (h_2, k_2)$ to derive $g_3 = g_1 \cdot g_2$, we can compute $h_3 = h_1 \cdot h_2$ and $k_3 = k_1 \cdot k_2$ independently. (2) If the quotient group G/H (with $H \trianglelefteq G$) is isomorphic to K, encoding on the data plane remains feasible. An element can be denoted by its mapping as $h \in H$ and $k \in K$ and each can be encoded individually. For element multiplication, while direct multiplication might not be feasible across both sections, more sophisticated techniques can achieve this. However, an exhaustive exploration of these techniques is beyond the current discourse.

Remarkably, the cache state utilized by P$_4$LRU$_3$ pertains to the group S_3, where S_3/C_3 is isomorphic to C_2. This insight steered our encoding strategy for cache states, and with S_4/V_4 (Klein four-group) isomorphic to S_3 and $V_4 = C_2 \times C_2$, it suggests that P$_4$LRU$_4$ could be implemented on the data plane. However, this would demand a more nuanced logic to store and manage the cache states.

3 P$_4$LRU BASED IN-NETWORK SYSTEMS

In alignment with the P$_4$LRU$_3$ data plane cache detailed in Section 2, we've developed three practical in-network systems: LruTable, LruIndex, and LruMon. In this section, we'll delve into each of these systems individually.

3.1 LruTable System

LruTable serves as a data plane network address translation (NAT) system. Its primary task is to convert a packet's virtual destination address into the corresponding real address. Figure 6 illustrates that LruTable utilizes a NAT table in the control plane for the address translation and has an array of 2^{16} P$_4$LRU$_3$ cache units on the data plane to cache specific table entries. The processing routine for LruTable is as follows:

Processing routine: LruTable incorporates a hash function $h(\cdot)$ and an array $P[1 \cdots 2^{16}]$. Each entry in this array is a P$_4$LRU$_3$ cache unit. For an incoming packet with virtual address va, the hash function $h(\cdot)$ determines the cache unit $P[h(va)]$. The virtual address va is then inserted into this cache unit.

- **Fast Path:** If a cache hit occurs, meaning the address va matches one of $P[h(va)]$.key[1], $P[h(va)]$.key[2], or $P[h(va)]$.key[3], the cache state is updated. We then fetch the real address ra corresponding to the virtual address va from the position
$$P[h(va)].\texttt{val}[P[h(va)].S_{lru}(1)]$$
within the value array.
- **Slow Path:** In case of a cache miss, we update the cache state and mark a `placeholder` (e.g., `0x00000000` or `0xFFFFFFFF`) in the value array. The packet is then forwarded to the control plane to consult the full NAT table and determine the real address ra. Upon retrieval, this packet carries the real address ra through the data plane once more, updating the cache unit $P[h(va)]$ and replacing the prior `placeholder` with the newly obtained real address ra.

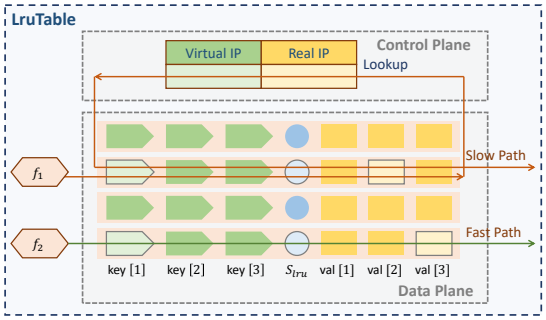


Figure 6: An example of LruTable system.

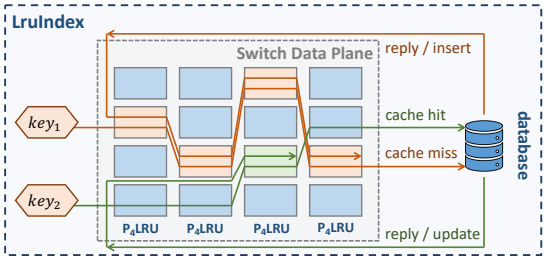


Figure 7: An example of LruIndex system.

Furthermore, if a cache hit occurs but returns a `placeholder`, the packet still requires the real address from the control plane. However, it won't process through the data plane cache again.

Resource Usage: LruTable is fully realized within the data plane of the Tofino programmable switch chip. The hardware resource consumption of this system is detailed in Table 2(a). Notably, the system occupies one out of the four available data plane pipelines.

3.2 LruIndex System

LruIndex serves as an in-network query acceleration system. Unlike NetCache [30], which caches key-value pairs directly, LruIndex caches the index (specifically, the 48-bit memory address) of the key in the database. This design facilitates supporting values of variable lengths (64 bytes in our configuration). As depicted in Figure 7, LruIndex deploys four series-connected P$_4$LRU$_3$ cache arrays to store indexes. Each array encompasses 2^{16} P$_4$LRU$_3$ cache units. The system follows the routines described below to process each incoming packet.

Processing routine: As depicted in Figure 7, each cache array $P_i[1 \cdots 2^{16}]$ pairs with a hash function $h_i(\cdot)$, for $1 \leqslant i \leqslant 4$. LruIndex treats query packets, which originate from the client and are destined for the database server, differently from reply packets dispatched from the server. Both packet types incorporate two supplementary fields in their header: `cached_flag` and `cached_index`.

- **Query packet:** For each incoming query packet containing the key k, the system consults all four cache arrays in a read-only fashion. In the context of the i-th cache array, the hash function $h_i(\cdot)$ determines the cache unit $P_i[h_i(k)]$ where a check is made

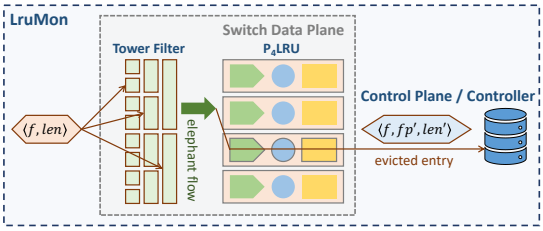


Figure 8: An example of LruMon system.

to ascertain if the key k is cached. If the i-th array retains the key k, the packet header's `cached_flag` is populated with i and `cached_index` is filled with the index sourced from $P_i[h_i(k)]$. If absent, `cached_flag` is set to 0. On receipt, the database server inspects `cached_flag`. If it reads 0, the server invokes built-in indexing, like the B+ Tree [12], to pinpoint key k's index; otherwise, it straightforwardly pulls the value linked to the key from `cached_index`.

- **Reply packet:** For query packets that read a `cached_flag` of 0, the database embeds the key k's index, derived from the inherent index structure, into the reply packet's `cached_index`. For each inbound reply packet, LruIndex examines its `cached_flag`. If it's non-zero (*i.e.*, $i \neq 0$), meaning the key was earlier cached in the i-th array, the key k gets prioritized as the most recent entry in the cache unit $P_i[h_i(k)]$ of the i-th array. In cases where the `cached_flag` field is set to 0, denoting that the key k is not in the cache, we undertake the following series of actions to insert the key k and its index `cached_index`: Firstly, we deposit the key k and its index into the cache unit $P_1[h_1(k)]$ of the first array. In doing this, the least recently used entry, which is the key $k_1 = P_1[h_1(k)]$.key[3] and its respective index, gets evicted. Subsequently, we place the evicted key k_1 and its index into the cache unit $P_2[h_2(k_1)]$ of the second array, *designating it as the least recently used entry*. During this step, the existing key $k_2 = P_2[h_2(k_1)]$.key[3] and its paired index in the cache unit $P_2[h_2(k_1)]$ are substituted by the key k_1 and its index. We then advance to the third array. Here, the key $k_3 = P_3[h_3(k_2)]$.key[3] and its index within the cache unit $P_3[h_3(k_2)]$ are replaced by the key k_2 and its index. In the final stage involving the fourth array, the key $k_4 = P_4[h_4(k_3)]$.key[3] and its associated index situated in the cache unit $P_4[h_4(k_3)]$ are switched out for the key k_3 and its index. By the culmination of this series of actions, the key k_4 along with its linked index are fully expelled from the data plane cache.

Series Connection Technique: LruIndex employs the series connection approach using P_4LRU cache arrays. This method leverages the inherent nature of the in-network query acceleration system: each key in the packet traverses the data plane twice. This behavior permits read-only access to multiple serially-connected cache arrays, enabling the determination of the cache array storing the query key. The cache's modification is solely executed by the reply packet when it accesses the data plane for its second round. Imagining a scenario where the data plane is accessed only once, and all query keys are injected from the first cache array, the same key might be logged in several arrays, leading to suboptimal cache utilization. LruIndex circumvents this issue by postponing cache updates until reply packets are received.

Resource Usage: LruIndex is fully implemented within the data plane of the Tofino programmable switching chip. Table 2(b) details the hardware resource consumption for the system. The system occupies all four available data plane pipelines. Each P_4LRU$_3$ cache array utilizes one pipeline. Put differently, we *"fold"* the switch's pipelines, turning parallel pipelines into a serial one, a trade-off where throughput is sacrificed for additional storage. LruIndex also supports versions employing either two or three pipelines.

3.3 LruMon System

LruMon is an advanced network telemetry system engineered to meticulously classify packets into their respective data flows via the data plane. The primary objective is to capture and measure the size of each flow, ensuring that the collective measurement across all flows is maximized, while simultaneously ensuring that no individual flow's size is overstated. Such precision is critical, particularly for tasks like traffic billing, as discussed in [18, 35]. Consulting Figure 8, we observe that LruMon is anchored on two foundational data structures. Firstly, it incorporates the TowerSketch [57] – a refined iteration of the Count-Min sketch [15], primarily harnessed for filtering mouse flows. The secondary structure is a P_4LRU$_3$ cache array that houses 2^{17} P_4LRU$_3$ units, employing both 32-bit flow fingerprints and 32-bit flow lengths as its keys and values, respectively. Detailing the operational logic, LruMon adopts the following procedural framework for packet processing:

Processing routine: Referring again to Figure 8, the system leverages the TowerSketch as its filtering mechanism. This filter is characterized by two distinct hash functions, $g_1(\cdot)$ and $g_2(\cdot)$, and is accompanied by two counter arrays, $C_1[1 \cdots 2^{20}]$ and $C_2[1 \cdots 2^{19}]$. Within these, the array C_1 integrates 2^{20} 8-bit counters, while its counterpart C_2 assimilates 2^{19} 16-bit counters. Notably, every counter is paired with an 8-bit timestamp, facilitating periodic counter resets, typically on a millisecond scale. Further, LruMon integrates a cache array denoted as $P[1 \cdots 2^{17}]$, paired with a hash function $h(\cdot)$. For every inbound packet characterized by its 5-tuple[5] – designated as f for key and its packet length len for value – the system initially employs hash functions to pinpoint two counters, specifically $C_1[g_1(f)]$ and $C_2[g_2(f)]$.

- **Tower Filter:** For each counter, the timestamp is updated. We check if there counter requires a reset, then increment the counter based on the packet length denoted by len. The estimated total length of flow f during the present interval is expressed as $\hat{len} = \min\{C_1[g_1(f)], C_2[g_2(f)]\}$[6]. Packets that belong to mouse flows and satisfy $\hat{len} < L$, where L is a predetermined threshold, are filtered out.
- **Cache Array:** For packets that are part of elephant flows, they are incorporated into the cache array. Specifically, the hash function $h(\cdot)$ is used to pinpoint a cache unit $P[h(f)]$. Another hash function $fp(\cdot)$ computes the 32-bit fingerprint $fp(f)$ of flow f, subsequently inserting the fingerprint $fp(f)$ into the cache unit $P[h(f)]$. If there's a cache hit – that is, fingerprint $fp(f)$ is found in the unit – we modify the cache state and augment the value linked to the key $fp(f)$ to $P[h(f)]$.val$[P[h(f)].S_{lru}(1)] + len$. In case of a cache miss, which means fingerprint $fp(f)$ isn't

[5] ⟨source IP, source port, destination IP, destination port, protocol⟩
[6] For more details on the operation of TowerSketch, refer to [57].

found in the unit, we refresh the cache state, adjust the value associated with key $fp(f)$ to len, and evict the existing key, which is $fp' = P[h(f)]$.key[3], and its length len'.

- **Remote Analyzer:** In instances of cache misses, an entry
$$\langle f, fp', len' \rangle$$
is also created and sent to the remote analyzer. This analyzer keeps a table T_{fp} with 5-tuples and their respective fingerprints, and another table T_{len} with 5-tuples and their respective lengths. If the flow f isn't listed in tables T_{fp} and T_{len}, an entry $\langle f, fp(f) \rangle$ is appended to table T_{fp} and an entry $\langle f, 0 \rangle$ to table T_{len}. Using the fingerprint fp', the 5-tuple f' corresponding to fp' is identified, and the length of flow f' in table T_{len} is increased by len'.

All packets that successfully pass through the Tower filter are temporarily stored in the data plane cache and subsequently uploaded to the remote analyzer. This ensures that LruMon measures these flows with impeccable accuracy. While different data plane caches don't compromise measurement precision, *a more efficient cache can diminish the number of entries transferred from the data plane to the analyzer, consequently alleviating the burden on the analyzer.*

Resource usage: We have successfully implemented the entire LruMon system within the data plane of the Tofino programmable switching chip. Table 2(c) presents a detailed overview of the system's hardware resource utilization. The system occupies two out of the four available data plane pipelines. Specifically, the Tower filter consumes one pipeline, while the P$_4$LRU$_3$ cache array utilizes another. Essentially, we've chosen to *"fold"* the switch's pipelines, a decision reflecting a trade-off: opting for enhanced logic capabilities at the expense of throughput. LruMon is also compatible with other sketches, such as the CM sketch [15] or the approximate CU sketch [60], when used as filters.

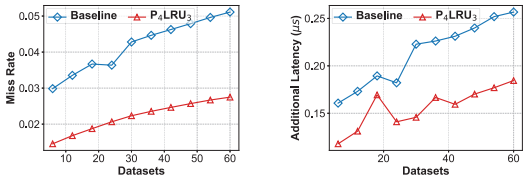
4 EVALUATION

In this section, we first utilize a Flnet S9280 Tofino switch with two pipelines to establish a testbed, upon which we evaluate the three P$_4$LRU systems. Following this, we deepen our performance analysis of the three systems by running simulations on a CPU platform.

Datasets: We employ the CAIDA 2018 [3] dataset, an anonymized IP trace, to create a synthetic dataset denoted as CAIDA$_n$. This is done by partitioning the one-hour CAIDA 2018 trace into 60 distinct one-minute datasets. From the initial n datasets, we extract $\frac{1}{n}$ minutes of data to craft our synthetic dataset. The intent behind the creation of CAIDA$_n$ is to vary the concurrency level of the dataset. Each of the datasets encompasses approximately 2.6×10^7 packets. Progressing from CAIDA$_1$ to CAIDA$_{60}$, the flow count varies from 1.3×10^6 to 2.4×10^6. Concurrently, the maximal count of concurrent flows escalates from 1.5×10^5 to 5.8×10^5.

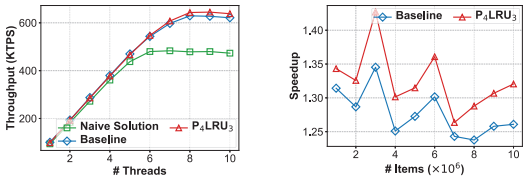
4.1 Testbed Experiments

In this section, we elucidate the performance variations discerned between the P$_4$LRU$_3$ cache and the hash table-based cache (analogous to P$_4$LRU$_1$) within the context of our three system prototypes. Within the depicted charts, the moniker *P$_4$LRU$_3$* signifies systems employing the P$_4$LRU$_3$ cache. In contrast, systems utilizing the hash table-based caching mechanism are denoted by the *Baseline* label. The *Naive Solution* label is reserved for configurations that do not utilize any caching mechanisms. We've deployed approximately 6000 lines of P4 code to implement all these systems.



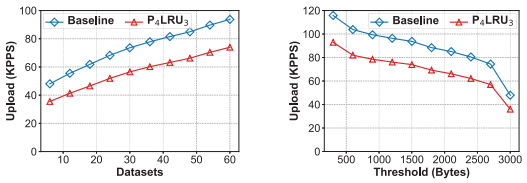
(a) Miss rate *v.s.* concurrency. (b) Latency *v.s.* concurrency.

Figure 9: Testbed experiment of LruTable system.



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

Figure 10: Testbed experiment of LruIndex system.

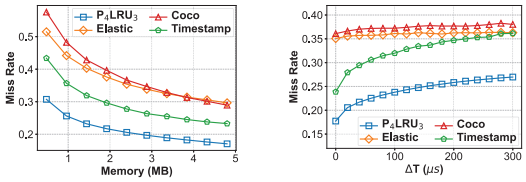


(a) Upload *v.s.* concurrency. (b) Upload *v.s.* threshold.

Figure 11: Testbed experiment of LruMon system.

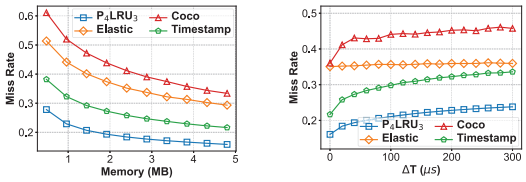
LruTable system (Figure 9): For our simulation, we employed the DPDK [4] driver to iterate and replay the CAIDA$_n$ datasets originating from our sender client. These packets traverse the Tofino switch, undergoing address translation in the process, before being relayed to the receiving client. Our observations, especially regarding the fast-path miss rate and the supplementary latency relative to direct forwarding without address translation, are illuminating. As illustrated in Figure 9(a), with increasing traffic concurrency, the miss rate of the system incorporating the P$_4$LRU$_3$ cache rises from a baseline of 1.4% to a zenith of 2.7%. Conversely, the systems leveraging the baseline solution display a steeper incline in the miss rate, from 3.0% to 5.1%. In comparative terms, the P$_4$LRU$_3$ manifests a performance that is, at its apex, 2.14× more robust than its baseline counterpart. Further insight is gleaned from Figure 9(b), detailing latency metrics. As traffic concurrency intensifies, the additional latency introduced by the P$_4$LRU$_3$ system elevates from an initial 0.11 μs to 0.18 μs. In contrast, the baseline solution sees its latency swell from 0.16 μs to a heftier 0.26 μs. In summation, at its optimal performance, P$_4$LRU$_3$ boasts efficiency that is roughly 1.35× greater than the baseline approach.

LruIndex system (Figure 10): We utilized the DPDK driver to facilitate the query thread and database server, assessing the database performance through the YCSB [13] benchmark. The query transaction set was generated based on the Zipf distribution [45] with a skewness of $\alpha = 0.9$. We employ the two-pipeline version of LruIndex. As depicted in Figure 10(a), for a database housing



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

Figure 12: Comparative experiment of LruTable.



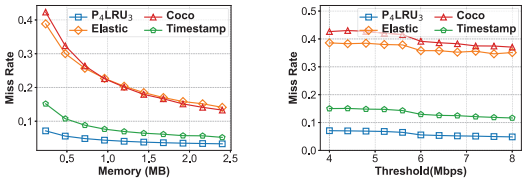
(a) Miss rate *v.s.* memory. (b) Miss rate *v.s.* ΔT.

Figure 13: Comparative experiment of LruIndex.

1×10^6 items and an escalating number of query threads, the query throughput of the P_4LRU_3-integrated system elevates from 98.5 KTPS (Kilo Transactions Per Second) to 644.8 KTPS. Meanwhile, the throughput of the baseline system grows from 100.3 KTPS to 629.2 KTPS. On the whole, the P_4LRU_3 system demonstrates a performance that's up to 1.03× superior to the baseline. Turning to Figure 10(b), for a configuration of 8 query threads, the throughput speedup of the P_4LRU_3 system relative to a naive solution oscillates between 1.26 and 1.36, whereas the baseline system's range is from 1.23 to 1.34. Conclusively, at its peak performance, P_4LRU_3 achieves a 1.08× edge over the baseline.

LruMon system (Figure 11): Employing the DPDK driver, we replayed the CAIDA$_n$ datasets at a 10Gbps speed from the sender client. Packets were directed to the Tofino switch for measurement, with generated data plane entries subsequently being dispatched to an external analyzer. Our focus was on the generation rate of packets, which contained entries sent to the analyzer straight from the data plane. We utilize the CM sketch [15] as the filter for LruMon. As illustrated in Figure 11(a), given a filter threshold of 1500 bytes and a reset span of 10 *ms*, the upload rate of the P_4LRU_3-infused system surges from 35.5 KPPS (Kilo Packets Per Second) to 74.0 KPPS as traffic concurrency intensifies. Conversely, the baseline system sees its rate climb from 48.0 KPPS to 93.7 KPPS. Broadly speaking, the P_4LRU_3 system outperforms the baseline by a factor of up to 1.35×. Figure 11(b) reveals that with an ascending filter threshold, the upload rate of the P_4LRU_3 system diminishes from 92.9 KPPS to 36.0 KPPS. Simultaneously, the baseline system's rate drops from 115.8 KPPS to 47.9 KPPS. Summarizing, the P_4LRU_3 system is, at its zenith, 1.33× more proficient than its baseline counterpart.

Analysis: The relatively muted improvement of P_4LRU in the LruIndex system, compared to the other two systems, can be attributed to the stochastic generation of the query transaction set. This renders the temporal continuity of the same query key less pronounced than that observed in the CAIDA dataset.



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

Figure 14: Comparative experiment of LruMon.

4.2 Simulation Experiments

In this section, we leverage the capabilities of a CPU platform to simulate the performance nuances of the P_4LRU cache across a diverse set of conditions. To quantify the similarity between the LRU replacement policy and other replacement strategies, we employ the metric, *LRU similarity*. Given a cache with a capacity of n, for each evicted entry, if the ranking of its last access time is represented by k, its relative ranking is deduced as $\frac{k}{n}$. In an ideal LRU cache scenario, this relative ranking consistently equals 1. Therefore, we define the LRU similarity as the average relative ranking of all evicted entries. We adopt the CAIDA$_{60}$ dataset as our default, rescaling its temporal aspect to a duration of one second. In the illustrations presented, LRU_{IDEAL} exemplifies the prototypical LRU cache. Meanwhile, P_4LRU_1 is indicative of the cache using the hash table, and P_4LRU_2 and P_4LRU_3 have their definitions anchored in Section 2.3.

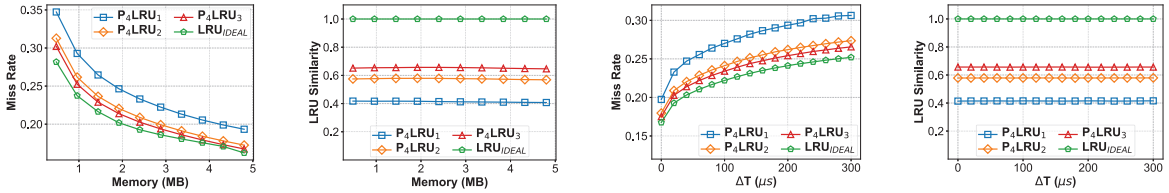
4.2.1 *Comparative Experiments.*

Our study juxtaposes P_4LRU with two LFU-based caching mechanisms, namely Elastic and Coco. These mechanisms utilize Elastic sketch [58] and Cocosketch [59] respectively to dictate item replacements. Additionally, we explore a timeout-based cache approach. This approach logs the last access timestamp for each entry, leveraging a predefined timeout threshold for item replacements. Notably, we've meticulously adjusted the timeout threshold to ensure optimal performance across different settings.

LruTable system (Figure 12): In Figure 12(a), by varying the cache memory allocation, we assess the cache miss rate. The miss rates achieved with the Cocosketch and Elastic replacement policies are comparable. The timeout strategy exhibits a slightly lower miss rate, whereas our P_4LRU_3 policy leads to reductions of up to 26.8%, 20.8%, and 12.7%, respectively. In a subsequent experiment (Figure 12(b)), adjusting the slow path latency ΔT, the cache miss rate demonstrates analogous trends. With P_4LRU_3, we achieve reductions of 18.4%, 17.3%, and 9.3%, respectively.

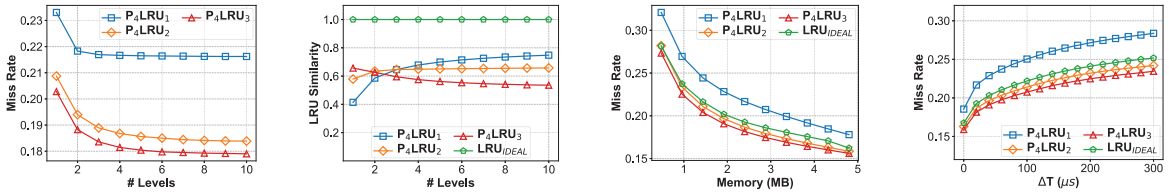
LruIndex system (Figure 13): As depicted in Figure 13(a), we modulate the cache memory and examine the resultant miss rate. The Cocosketch policy incurs a higher miss rate than the Elastic policy. The timeout strategy fares better with a reduced miss rate, and our P_4LRU_3 policy further decreases the rate by up to 33.3%, 23.6%, and 10.4%, respectively. When the query latency ΔT is altered (as seen in Figure 13(b)), the findings mirror the aforementioned results, with P_4LRU_3 delivering reductions of 23.7%, 19.0%, and 9.8%, respectively.

LruMon system (Figure 14): Figure 14(a) showcases the impact of varying cache memory on the miss rate. The miss rates under Cocosketch and Elastic replacement policies appear similar. The



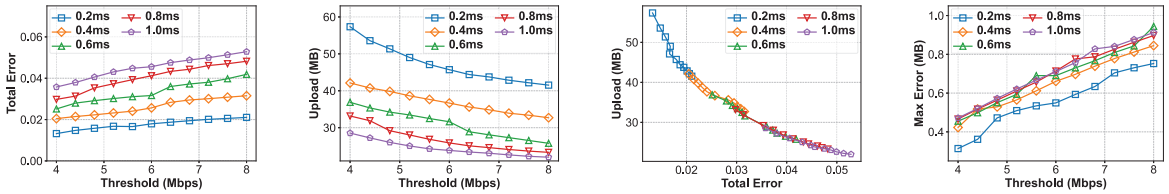
(a) Miss rate *v.s.* memory (b) Similarity *v.s.* memory (c) Miss rate *v.s.* ΔT (d) Similarity *v.s.* ΔT

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.

timeout strategy yields a diminished miss rate, while our P_4LRU_3 policy results in reductions by up to 35.2%, 31.7%, and 8.0%, respectively. Further, when the filter threshold is adjusted (as illustrated in Figure 14(b)), the findings remain consistent with previous observations. Employing P_4LRU_3 allows for reductions by up to 36.0%, 31.2%, and 8.1%, respectively.

4.2.2 *Parameter Experiments.*

LruTable system (Figure 15): In Figures 15(a) and 15(b), we adjust the memory allocated to the cache and study both the cache miss rate and LRU similarity. When considering the miss rate, the P_4LRU_3 cache consistently mirrors the ideal LRU cache's performance. With ample memory, P_4LRU_2, P_4LRU_3, and the ideal LRU cache exhibit analogous results. In terms of similarity, the P_4LRU_3 cache consistently scores the highest, remaining largely unaffected by memory variations. Figures 15(c) and 15(d) present findings when the slow path latency ΔT is varied. For both the miss rate and similarity, the P_4LRU_3 cache's performance remains commendably close to the ideal LRU cache, and it remains largely constant with latency changes.

LruIndex system (Figure 16): As presented in Figures 16(a) and 16(b), we vary the number of connection levels used in the serial connection technique, evaluating cache miss rate and LRU similarity in the process. In terms of the miss rate, the P_4LRU_3 cache consistently has the lowest rate, with both P_4LRU_2 and P_4LRU_3 outperforming P_4LRU_1 by a significant margin. Notably, as the number of levels increases, the similarities for P_4LRU_2 and P_4LRU_1 rise, while for P_4LRU_3, it decreases. Given that LRU similarity often signifies universality, this implies that more levels boost performance for the CAIDA dataset, but not necessarily for others. As a compromise, we default to four levels, ensuring a desirable balance between a low miss rate and adequate LRU similarity. Figures 16(c) and 16(d) highlight findings from adjustments in memory and query latency ΔT of the database server. Across these experiments, the P_4LRU_3 cache remains the most aligned with the ideal LRU cache.

LruMon system (Figure 17): Figures 17(a) and 17(b) demonstrate the impact of tweaking the bandwidth threshold and reset period of the Tower filter on P_4LRU_3, where we assess the total error rate and upload volume. The bandwidth threshold is derived as the filter threshold's ratio to the reset period, while the total error rate equates to the total underestimation error's ratio over the total byte count. Findings suggest that a shorter reset period decreases errors but increases upload volume. This is attributed to longer reset periods filtering out burst traffic. Intriguingly, Figure 17(c) illustrates that regardless of the reset period, the upload volume remains fairly consistent when total error is kept constant. Lastly, Figure 17(d) showcases the maximum flow-level error, which never surpasses the filter threshold.

5 RELATED WORK

In this section, we first explore standard cache replacement policies presented in §5.1, followed by a discussion on existing data plane cache solutions in §5.2.

5.1 Cache Replacement Policies

The quintessential cache replacement algorithm would always discard the item least likely to be needed in the foreseeable future. Implementing such an ideal is unattainable, given our inability to foresee upcoming requirements. However, several algorithms have been proposed over the years aiming to approximate this ideal. Broadly, these can be categorized into three: recency-based policies, frequency-based policies, and hybrid policies.

Recency-based policies prioritize discarding items based on their most recent reference in their lifespan. For instance, the widely recognized LRU policy targets the eviction of least recently used items. Prominent variants of LRU encompass Early Eviction LRU (EELRU) [53], Segmented LRU (Seg-LRU) [23], RRIP [27], among others [17, 28, 46, 55]. On the other end of the spectrum, the MRU (Most Recently Used) policy [10] seeks to remove the most recently accessed items. Interestingly, studies show that for patterns like random or cyclic access, MRU policies surpass LRU in hits, given their inclination to maintain older data [16]. Furthermore, a niche subset of recency-based policies [26, 48] prolongs the lifespan of selected items by housing them in a supplementary buffer.

Frequency-based policies employ access frequency as a yardstick for item replacement, giving preference to items with higher access frequencies over those accessed less often. The LFU (Least Frequently Used) [11] stands out as the most direct approach in frequency-based policy, attributing a frequency counter to individual items. A notable shortcoming of frequency-based policies is their occasional inability to adapt to shifting application phases; an item revered in a previous phase due to high frequency might continue to be cached in a subsequent phase despite being obsolete. Numerous strategies have been proposed to tackle this challenge, primarily by integrating frequency data with recency metrics to age out older items. Representative solutions feature FBR [50], LRFU [33], and others [19, 42, 51].

Hybrid policies nimbly adjust their replacement strategies in alignment with the prevailing working set. The challenge here is to discern the most fitting policy while minimizing hardware overhead, as they juggle multiple strategies simultaneously. Noteworthy implementations of hybrid policies comprise ARC (Adaptive Replacement Cache) [25, 40, 56], Set Dueling [46, 47], and others [29].

5.2 Data Plane Cache Solutions

Existing data plane cache solutions can broadly be categorized into three primary domains: hash-based solutions, frequency-based solutions, and non-real-time solutions. For additional insights into other solutions, readers are directed to references such as [31, 36, 38, 44, 54].

Hash-based solutions focus on the recording of recent entries by sustaining a hash table on the data plane. Pioneering works in this domain include NetSeer [61], Pegasus [34], SpiderMon [37], and Jaqen [39]. To elucidate, NetSeer [61] orchestrates a basic hash table in the data plane, ensuring that older entries are replaced upon a collision of two items in the same slot. Similarly, SpiderMon [37] employs comparable techniques to retain telemetry data within the data plane. However, a dominant challenge faced by these methods arises from hash collisions. The frequency of accessed items colliding into the same slot can precipitate a substantial decline in the hit rate.

Frequency-based solutions prioritize the conservation of frequently accessed items on the data plane. Such techniques craft sophisticated data structures, aiming to proficiently log voluminous flows while filtering out minuscule ones. Even though they do not explicitly promote themselves as data plane caches, their underlying functionality essentially serves as a cache—retaining frequently accessed items and discarding the seldom-used ones. Renowned works in this area include HashPipe [52], PRECISION [7], CocoSketch [59], and Elastic sketch [58]. Nevertheless, one critical limitation of these frequency-centric solutions is their prolonged retention of frequent items, even if they have become obsolete.

Non-real-time solutions adopt a more deferred approach to cache updating. For instance, NetCache [30] operates a table of recurrent items on the data plane, with periodic interchanges of these items through the control plane. BeauCoup [9], on the other hand, logs the last access time of every entry, periodically expunging stale entries to make space for newer items. However, a prevalent drawback of these non-real-time solutions is their elongated update latency, making them less adept at adapting to abrupt workload fluctuations. Elaborating further, PKache[22] utilizes the P4 language and BMv2 architecture to implement a plethora of cache replacement strategies, inclusive of LRU. Nonetheless, PKache demands deferred cache updates on cache misses, necessitating a second access of the same requested packet to the data plane. Moreover, when executing LRU policies, PKache must account for the storage of timestamps.

6 CONCLUSION

In this paper, we begin by examining the challenges of implementing the classical LRU cache on the data plane of programmable switches. Subsequently, we introduce a pipelined variant of the LRU implementation, termed as P_4LRU. Within the P_4LRU design, we sidestep the pitfalls of circular data access by incorporating a deterministic finite automaton (DFA), denoted as S_{lru}. We delve into the intricacies of employing the stateful arithmetic logical unit (ALU) of programmable ASICs to facilitate the storage and state transitions of the DFA. Building upon the P_4LRU cache, we conceptualize three innovative 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. These systems undergo rigorous performance evaluations to ascertain their efficacy. The empirical results are compelling. In comparison to the baseline replacement policy, the P_4LRU cache registers end-to-end performance enhancements of up to 35%, 8.2%, and 35% across our triad of systems, respectively. For those interested in further exploration or adaptation, all associated source codes for the three systems are accessible on 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] Roy Friedman, Or Goaz, and Dor Hovav. Pkache: A generic framework for data plane caching. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, pages 1268–1276, 2023.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] Nimrod Megiddo and Dharmendra S Modha. Outperforming lru with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, 2004.

[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] 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.

[43] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[44] 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.

[45] David MW Powers. Applications and explanations of zipf's law. In *New methods in language processing and computational natural language learning*, 1998.

[46] 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.

[47] 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.

[48] 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.

[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] 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.

[51] Dennis Shasha and T Johnson. 2q: A low overhead high performance buffer replacement algorithm. In *Very large database systems conference 1994, September 1994*, 1994.

[52] 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.

[53] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. Eelru: simple and effective adaptive page replacement. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):122–133, 1999.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, et al. {LightGuardian}: A {full-visibility}, lightweight, in-band telemetry system using sketchlets. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 991–1010, 2021.

[61] 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.