Limited Associativity Caching in the Data Plane

Roy Friedman  Or Goaz  Dor Hovav

Computer Science Department
Technion

March 10, 2022

Abstract

In-network caching promises to improve the performance of networked and edge applications as it shortens the paths data need to travel. This is by storing so-called hot items in the network switches on-route between clients who access the data and the storage servers who maintain it. Since the data flows through those switches in any case, it is natural to cache hot items there.

Most software-managed caches treat the cache as a fully associative region. Alas, a fully associative design seems to be at odds with programmable switches’ goal of handling packets in a short bounded amount of time, as well as their restricted programming model. In this work, we present PKache, a generic limited associativity cache implementation in the programmable switches’ domain-specific P4 language, and demonstrate its utility by realizing multiple popular cache management schemes.

1 Introduction

Caching is a fundamental technique for boosting systems performance. In particular, software-managed caches, aka software caches, are employed in multiple data-stores and databases [2, 12, 15, 19, 20, 21, 30, 40, 42, 41], operating systems, middleware, streaming services, and is a major capability of edge computing. The common motivation behind caching is to store data closer to the application than its source and avoid recalculating queries, query plans, and temporal indices. For example, DRAM memory is faster and closer in the memory hierarchy than secondary storage, local storage is closer than accessing data over the Internet, etc. This way, the requested information can be served quickly whenever the requested data is already in the cache, also known as a cache hit. In an edge computing setup, caching also saves network bandwidth and reduces the load on servers. This is because when the data is served from a local edge cache, there is no need to load it from the remote server [19, 20].

Since caches are limited in size, a cache management mechanism is required to decide which items should be kept in the cache. Similarly, when there is not enough space for all items, the management scheme decides which items should get evicted (known as cache victims). A plethora of cache management policies has been devised, including e.g., LRU [22], LFU [29], ARC [33], LIRS [25], FRD [35], Hyperbolic [9], and W-TinyLFU [17], to name a few. Largely speaking, these schemes treat the cache as a fully associative structure. That is, for every item that is inserted into the cache, the management policy can potentially select any other item as its victim.

In contrast, hardware managed caches employ a limited associativity design. That is, the CPU cache is divided into multiple sets, each of which contains $k$ locations, forming a $k$-way associative cache. This is to ensure bounded time look-ups and to reduce the hardware circuitry complexity and cost required to support full associativity. Recently, it was shown that even for software caches it makes sense to employ a limited associativity design [1]. This is because limited associativity enables trivial parallelism, reduces contention on data structures, increases memory density, and enables very simple $O(1)$ inserts and look-ups.

In-network caching promotes storing cached data in network switches and routers. Since the data flows through these devices in any case, it makes sense to cache it there. In fact, in-network caching is considered one of the enabling technologies for 5G and 6G’s promised performance boost as well as edge computing.
**PKache** stores in a P4 register \(d\) sets, where each set contains \(k\) ways (here \(d = 3\) and \(k = 2\)). Each element includes a key, a value and SCN (Sequence Change Number) - the latter encodes any additional information needed for the cache management policy.

Programmable switches are gaining momentum, with offers from large vendors such as Intel [3], Nvidia/Mellanox [37], and Broadcom [24]. Instead of just forwarding packets, the switch can be programmed to manipulate and monitor the data flowing through it. Yet, for speed and energy efficiency, this programming model is restricted compared to general-purpose computing [10, 38].

The pioneering NetCache [26] implements a key-value distributed cache inside programmable switches. NetCache processes queries for hot items and balances the load across the storage nodes, while each caching switch is treated as a fully associative structure, implementing an ad-hoc cache management policy. We claim that limited associativity designs are better fitted to programmable switches’ goal of handling packets in a short bounded amount of time and their restricted programming model. Limited associativity also simplifies the realization of existing well studied cache management approaches inside the programmable switch.

**Contributions:** We study the utility of limited associativity caching in programmable switches. In particular, we present **PKache**, a novel generic P4 caching framework that adheres to \(k\)-way associativity cache design, where \(k\) is a controllable compile time parameter. **PKache** supports both single region caching as well as multi-region caching, and can be instantiated with diverse specific cache management schemes in both cases. In particular, we have implemented LRU [22], LFU [29], FIFO, and Hyperbolic [9] policies for the single region case, as well as the popular W-TinyLFU [17] as a representative of multi-region schemes. Many other cache management policies can be similarly implemented. As we explain in this paper, \(k\)-way associativity helps overcome the limitations presented by the P4 programming model.

We have compared the hit-ratios obtained by our **PKache** system to those obtained by an unrestricted Python implementation of the respective schemes over several synthetic and real-world traces. The results indicate that despite the restrictions of the P4 programming model, **PKache** yields very similar hit ratios to an unrestricted implementation.

**Paper roadmap:** We provide more detailed background and survey related work in Section 2. The P4 Programming Language is briefly presented in Section 3. We present **PKache** in Section 4 and evaluate the obtainable performance in Section 5. We conclude with a discussion in Section 6.
2 Background and Related Work

2.1 Programmable Switches

Programmable switches are capable of performing more sophisticated tasks than the ordinary packet forwarding action, based on a dynamically uploaded program. That is, their data plane functionality can be defined by the given program.

One of the most popular programming languages for such data planes today is P4. P4 is a domain-specific language and it is also a target specific language, so in every target, the spec of the language can be a little bit different, depending on the manufacturer of the switch. P4 supports conditions, match-action tables, state-full objects, and more. It does not support very common control methods like loops, recursion, etc. The match-action table is a list of conditions. When one of them is fulfilled, a predefined action is executed. This action performs some business logic on the incoming packet.

2.2 Cache Management Policies

As mentioned in the Introduction, there are numerous cache management policies. Here we briefly describe the five policies whose k-way variants we implement in our work.

Least Recently Used (LRU) [22] LRU is arguably the most widely employed policy. LRU is based on the underlying principle of time locality, suggesting that the probability of accessing a given item is related to the time that has passed since it was last accessed. Hence, when the cache is full, the algorithm always evicts the least recently used item. The simplest way to implement LRU is through a priority queue where each time an item is being accessed, it is moved to the head of the queue. Yet, such an implementation creates high contention on the queue’s head. It also requires updating the cache meta-data on each access (both hits and misses).

Sampled LRU [40] is an approximate alternative, in which (only) the timestamp of each item’s last access is stored. To find the cache victim, a sample of k random items is selected, and the least recently accessed among them becomes the victim. Alas, this still requires updating the meta-data on each access, invoking the PRNG k times, and accessing k random memory locations.

Another popular approximation of LRU is Clock [13], in which the system treats the cache as a logical ring and lazily resets the access time of one item on each cache access using an analogy of a clock’s moving hand. The victim is the first item whose last access time is zero. Here again, the meta-data needs to be updated on each cache access, and the worst-case eviction time is \(O(C)\) for a \(C\)-sized cache. Also, being approximate, both sampled LRU and Clock exhibit slightly worse hit-ratios than LRU.

Least Frequent Used (LFU) [29] LFU is based on the assumption that the probability of an item being accessed is proportional to its popularity, or in other words, to the frequency by which it was accessed until now. To that end, LFU maintains a frequency counter for each cached item. When the item is accessed, the counter is incremented by 1. When the cache is full and a new item needs to get in, the victim is the cached item with the minimum frequency.

Clearly, for LFU the size of the counters can be significant over time. There are a few mechanisms to decrease the values of the counters once in a while as well as to age the counters to accommodate for recency effects. A naive implementation of LFU uses a heap data structure, whose complexity is \(O(\log C)\). More recently, it was shown how to implement a heap suitable for LFU in \(O(1)\) [7].

FIFO [6] With the FIFO policy, the cache behaves in the same manner as a FIFO (First In, First Out) queue. That is, the cache evicts the elements in the order they were added, regardless of how often or how recently they were accessed before. The benefits of this policy include its simplicity and the fact that an item’s meta-data is never updated after its initial insertion to the cache. Alas, its performance is often worse than other policies and it may suffer from the Belady anomaly in some workloads [6]. Still, a recent work suggests that for modern cloud storage workloads, its performance is comparable to LRU [18], and sometimes even better.
The recent Hyperbolic cache policy combines two metrics: recency and frequency. When a new item enters the cache, the insert time is logged and a new counter is initiated for the newly cached item with a value of 1. The algorithm increments this counter by 1 every time the item is requested. When the cache is full, the eviction mechanism samples a few items and calculates their relative priority. The victim is the item whose priority is the lowest among the sampled items. Formally, denote the insert time of item $i$ as $t_i$ and the request count of $i$ as $n_i$. The priority of item $i$ at time $\text{now}$ is calculated as $p(i) = \frac{n_i}{\text{now} - t_i}$.

**Hyperbolic Cache** [9]  

**W-TinyLFU** maintains two cache regions, a window cache and a main cache as well as an approximate frequency-based admission filter called TinyLFU. With **W-TinyLFU**, new items are first inserted into the window cache. Victims of the window cache are compared by the TinyLFU filter against the would-be victim of the main cache in terms of their approximate frequency. The winner gets to be in the main cache, while the loser is completely removed from the cache (although its approximate statistics is still tracked by the TinyLFU filter). See illustration in Figure 2.

The TinyLFU filter maintains an approximate representation of the access frequency of a large sample of recently accessed items. **W-TinyLFU** is compact and lightweight as it is based on Count-Min sketch [14]. It includes an aging mechanism and it caps the maximal frequency counts, to expedite the aging of items. **W-TinyLFU** is the management policy employed by the Caffeine [31] and Ristretto [15] caching libraries, and many other projects and products.

### 2.3 Caching in the Data Plane

**NetCache** [26]  

**NetCache** is a rack-scale key-value store design that supports billions of QPS with bounded latency. **NetCache** includes an internal load balancer to help it override spikes in the number of requests. Furthermore, **NetCache** guarantees cache coherence with only a minor overhead. **NetCache** is implemented in $P_4$, and its performance exemplifies the benefits to distributed systems enabled by high-speed programmable switches. We note that the evaluation part of [26] only focused on throughput and latency and did not include any hit ratio measurements.

The cache management mechanism employed by **NetCache** is an ad-hoc policy hard-coded into the solution. In contrast, **PKache** is a generic caching architecture for programmable switches. Also, **PKache** demonstrates the flexibility and simplicity that $k$-way associativity brings to in-network caches implemented inside programmable switches.

**Limited Associativity Software Cache** [1]  

The work in [1] demonstrated that limited associativity is a promising method for software caches. It is easier and simpler to implement a limited associativity cache than a fully associative one. Using limited associativity reduces memory overheads and increases parallelism compared to fully associative memory. The resulting cache also imposes lower CPU overheads than fully associative designs and sampled-based approaches. The major penalty of this method is a slight reduction in hit ratio compared to full associativity in certain workloads.

**KV Direct** [30]  

**KV Direct** offers a smart NIC library for accessing an in-memory key value store using RDMA. Hence, **KV Direct** [30] expedites accesses to nearby machines, connected through a very
fast network, while our work provides a general in-network caching solution for key-value stores, even remote ones.

3 P4 Programming Language

P4 is a domain-specific language describing how a PISA \cite{22} (Protocol Independent Switch Architecture) realization should process its packets. Also, it is a target-specific language, so in every target, the spec of the language can be a little bit different, depending on the manufacturer of the switch. The main motivation behind P4 is to enable expressing how packets should be processed by the data plane of a programmable forwarding element such as a hardware or software switch, network interface card, router, or network appliance \cite{10} \cite{38}.

Unlike most programming languages, P4 does not support common features such as loops, recursive calls, memory allocation, or even the use of division and floating points. This is meant to ensure that the amount of computation per packet can be bounded at compilation time and all data structures’ sizes must be fixed at compile time. On the other hand, P4 supports some special constructs like tables, parsers, deparser and more, which we survey below.

3.1 P4 Constructs

**Parsers:** Parser is a section that describes how to identify the packets’ header, how to extract them, and the permitted sequences we can perform on them. It uses a state machine to extract from the packet its header and has three predefined states: Start, Accept and Reject.

**Deparsers:** Deparser describes how the output packet would look like. It emits the headers into a well-formed packet that will be returned from the switch.

**Actions:** Actions are code sections that can read and write the data being processed and are the main section in which the control plane can influence the data plane. They are like procedures in other languages. The body consists of sequences of statements. Note that some targets, like BMv2 \cite{34}, impose restrictions on the actions’ body, e.g., conditional execution in actions is not supported on BMv2.

**Tables:** Tables are a special structure that describes a match-action unit. Processing a packet using a match-action table involves the following steps: key construction, key lookup in a lookup table (“match” step) and an action execution (“action” step) \cite{38}. A P4 table consists of the following:

**Key:** A key is a property in the form of \((e : m)\) when \(e\) is an expression that describes the data to be matched in the table and \(m\) is a \textit{match\_kind} constant that describes the algorithm used to perform the lookup. There are three kinds of match types. One of them is called \textit{ternary}. \textit{Ternary} has the following meaning: On each entry in the table, a mask is provided. Then, the field value and the mask are bit-wise ANDed before a comparison is made. The field value and the table entry need to agree on the bits set in the entry’s mask. Agreement means that all the “regular” bits are the same between the result and the entry. We can create masks in P4 using the infix operator &&&. It takes two arguments of type \texttt{bit\:〈W〉} and creates a value of type \texttt{set\:bit\:〈〈W〉〉}. The right value is used as a mask, where each bit set to 0 in the mask is a “don’t care” bit. Mathematically, we define &&& in the following way:
\[
a \&\& b = \{ \text{c of type bit < W> where } a \& b = c \& b \}.
\]

**Actions:** All possible actions that may appear within the associated lookup table or in the default action.

**Default action:** An action that is invoked automatically by the match-action unit whenever there is no match for the supplied key in the entries section.

**Entries:** An entry is a list of table properties in the form of \((v : a)\) where \(v\) is a value and \(a\) is an action. When the value is matched by the \textit{match\_kind} algorithm, the desired action is performed. Entries in a table are matched in the program order, stopping at the first matching entry.
**Stateful Objects:** Most P4 constructs are stateless: Given an input packet, they produce a result that depends only on the packet itself. However, some stateful constructs may retain information across packets. One type of stateful object is called *extern*. These are objects that can be read and written by the control plane and data plane. As we mentioned before, all stateful elements must be explicitly allocated at compilation time and have a fixed size.

One of these *extern* objects is called a Register. Registers are stateful and behave like arrays (index and value). We can reference an item in a register by its index using the *register_read* and *register_write* primitives.

3.2 **Challenges with P4**

Unlike other programming languages, P4 does not support loops, recursive calls, floating-point operations and more. Without loops, iterating over registers to determine if an element is inside the cache or not is hard, not to mention the eviction process in which we need to check each element to determine which element should be evicted according to the cache policy. Any code that needs to iterate over a register’s values is impossible, and we have to find workarounds.

The lack of floating point operations complicates certain algorithms and implementing probabilistic structures. In Hyperbolic cache [9], for example, we have to find an approximate way to calculate the priority of each item to determine the preferred element for eviction.

3.3 **P4 and Limited Associativity Caching**

We argue that P4’s design and restrictions as surveyed above encourage us to favor a limited associativity design over a fully associative one. The lack of loops and iterations makes it very inefficient to work with fully associative caches in terms of run time and the number of operations. Further, the lack of a loop construct means that an inlined code needs to be generated instead, which would result in significant code bloat. In principle, the use of TCAM enables us to determine whether an item is in the cache or not in a single comparison. However, as we detail below, there is a limit on the size of a TCAM mask and therefore on the number of items that can be matched in a single TCAM comparison.

In contrast, a limited associativity cache overcomes, or at least ameliorate, these limitations. With such a design, we are able to deploy large caches while minimizing the impact on the runtime or the number of operations performed per packet. For example, when we organize the cache as a k-way, the TCAM comparison only needs to match the k items inside the set to which the item is mapped. This is independent of the number of sets in the cache. Hence, we can increase the cache size by increasing the number of sets while keeping k relatively small. The exact details are explained below.

4 **Design and Implementation**

4.1 **Overview**

PKache is implemented in P4 spec 16 [38, 10] and tested on BMv2 [34]. We implemented PKache to work with both a single cache region and multiple cache regions designs. Currently, PKache supports LFU [29], LRU [22], Hyperbolic [9] and FIFO as cache management policies for each cache region as well as a TinyLFU filter in case of multiple cache regions, thereby also supporting the W-TinyLFU scheme [17]. The P4 code is generated by a Python script [23] using Jinja2 [27] and depends on the following parameters (i.e., these values cannot be changed during run-time):

- $k_m$ - Number of ways in the main cache.
- $d_m$ - Number of sets in the main cache.
- $k_w$ - Number of ways in the window cache (in case of multi cache region).
- $d_w$ - Number of sets in the window cache (in case of multi cache region).

For any incoming packet with a key request, our algorithm checks if the key exists in the cache. In case the key exists, the value is fetched and returned to the client inside an outgoing packet, without
reaching the storage servers at all. Otherwise, the switch routes the packet to the storage servers and waits for a response. When the response is transmitted to the client through the switch, our algorithm updates the cache depending on the policy and routes the outgoing packet to the client. In a multi-region cache, the process is slightly different, as detailed in Section 4.3 below. We now cover the main components of our solution:

**In-Switch Storage:** Our memory-store consists of one P4 register for each cache region. A P4 register is a stateful predefined fixed-sized object with key-value store capabilities.

PKache stores sets of cached items inside a register. The keys for the register’s entries are numbered from 0 to the number of sets (d_m or d_w) minus 1. Items are mapped to sets by hashing their keys to the range of sets; we refer to the set index as the Key Hash. An illustration of a PKache register is shown in Figure 1.

The value of each entry in the register is the cache set itself, which is a bit array containing all items key-value pairs of the set. Those pairs are stored in the memory space. Further, we allocate some extra auxiliary space for each key-value pair as a dedicated area for the cache policy metrics. We refer to the area reserved for the policy metrics as Sequence Change Number (SCN).

The algorithm parses the data with a bit-slice operator, so the exact key-value-metric can be easily represented as attributes. Also, we use additional stateful registers for our implementation. These include one register for each cache region that is built like the previous one but contains just the keys (and not the whole element). This register helps us determine whether the key is inside the cache or not. Another register represents the potential candidate for eviction and some specific policy implementation registers.

**Pipeline Model:** Programmable switches offer the pipelined PISA execution model and P4 exposes the pipeline architecture to the programmer. Each P4 code can be broken into stages, and different stages can be run in parallel to improve performance. As a rule of thumb, the number of stages should be limited to a few dozens so the switch latency is not detrimentally affected.

**Ternary Operations:** For any new item, we find the relative set that corresponds to the desired key. Then, our algorithm uses a table with a ternary match_kind identifier (TCAM) that indicates whether the requested key is inside the cache or not. This is performed by taking the value in the keys register for the corresponding set and using a bit-wise XOR operation with the requested key concatenated k times. Then we are using a ternary mask to check whether one of the values in the TCAM entries is equal to zero. If it is indeed the case, i.e., the key is found in the cache set, then we know that the item is in the cache; otherwise, it is not.

Thus, with just one table comparison and without the need to iterate over the cache elements, we can discover if the requested key is in the cache. In addition, we know exactly the key’s position inside the cache (the TCAM entry whose value was equal to zero), so we are able to get the item directly from the cache when it exists there. This is made possible by the limited associativity approach. Note that here we assume that the requested key is not equal to zero.

**Reduce Processing:** For inserting an element to the cache, we are using the reduction (or folding) pattern. In our case, we reduce the cache set items to a single cumulative value (the element or the victim). Specifically, we apply our comparison function to the first element’s key (among the k) in the respective set and the requested key to generate a partial result. We then use our partial result together with the second item in the set to generate another partial result. This is repeated until the set is exhausted and then we get a single cumulative value (for example, the candidate for eviction). Obviously, here the fact that k, the number of items per set, is relatively small is important to ensure fast packet processing.

**Eviction Process:** In case the key exists, the algorithm fetches the value from the cache and returns it in an outgoing packet to the client. Otherwise, the algorithm fetches the key from the storage server and stores the key-value pair in the first empty way of the respective set in the cache. In case the cache set is full, based on the chosen cache management policy, the candidate for eviction is chosen by comparing the metrics values of all elements in the set using the reduction technique we described above. Note that in the case of a multi-region cache, the eviction process is more complicated, so we describe it later.
4.2 Single Cache Region

4.2.1 FIFO Single PKache

Algorithm 1 lists the code for PKache’s FIFO policy implementation when inserting an element. Initially, the algorithm inserts the element to the first place in the respective set (lines 17-14). By doing so, we take the first element that was there before and mark it as the candidate (line 14). We then apply the reduction process in which we insert the candidate to the cache and mark the next element in the set as the new candidate (lines 5-9, 18-20). At the end of the process, the last element (the first element that was inserted and is still in the cache) leaves the cache.

Note that FIFO serves as a relatively easy exercise for implementing cache management policies. Still, recent evidence suggests that on many modern workloads, FIFO can obtain comparable (sometimes even better) hit ratios to LRU [18].

Algorithm 1 Insert element to a 4-way cache with FIFO policy (automatically generated for $k_m = 4$)

1: Initialization:
2: cache ← Register($d_m$)
3: keys_cache ← Register($d_m$)

4: procedure GET_CANDIDATE(cache_set, keys_set, index, candidate)
5: temp ← cache_set[index]
6: cache_set[index] ← candidate
7: keys_set[index] ← candidate[KEY_LOC]
8: candidate ← temp
9: return candidate

10: procedure INSERT_TO_CACHE(key)
11: h ← key%d_m
12: cache_set ← cache[h]
13: keys_set ← keys_cache[h]
14: candidate ← cache_set[0]
15: cache_set[0][KEY_LOC] ← pkt.key
16: cache_set[0][VAL_LOC] ← fetch_value_from_storage()
17: keys_set[0] ← pkt.key
18: candidate ← GET_CANDIDATE(cache_set, keys_set, 1, candidate)
19: candidate ← GET_CANDIDATE(cache_set, keys_set, 2, candidate)
20: GET_CANDIDATE(cache_set, keys_set, 3, candidate)
21: return cache_set[0]

4.2.2 LRU Single PKache

LRU needs to keep track of items’ most recent access to evacuate the least recently used item. Algorithm 2 lists the code for our PKache’s LRU policy implementation. Recall that this code is automatically generated for any value of $k_m$; in this example, $k_m = 3$. The implementation uses a timestamp to record when each item was recently accessed. Our algorithm maintains this timestamp in the Sequence Change Number (SCN) field. The SCN is a number that is incremented every time there is an update. Thus, every change has a unique number that is distinguishable from other changes. By using SCN as the last access indicator, it is easy to decide which item has not been accessed for the longest period. Given that each set has only a small number of items, this can be done efficiently (a benefit of limited associativity).

First, the algorithm checks if the key exists in the cache (line 23) using the keys_set and the ternary match table. As mentioned, we concatenate the key cache_size times to get a bit string, and then xor the result with the keys_set represented as a bit string. If any index returns zero, it means that the element is inside the cache.

If the key is inside the cache, we are fetching the value immediately from the relevant index in the cache_set. At this point its SCN is updated to the value defined in line 22 and the cached value is returned to the client (lines 24-26).

If the key does not exist in the cache at all, we insert the key as the first element in the cache_set (and update the keys_set accordingly). We also define the previous first element as our victim for eviction (lines 28-32). The algorithm fetches the value from the server cache or storage. We define an abstract method for this step in line 30 but it can be easily changed.
Algorithm 2 A LRU 3-way cache (automatically generated for $k = 3$)

1: *Initialization* :
2: \[\text{cache} \leftarrow \text{Register}(d_m)\]
3: \[\text{SCN} \leftarrow 0\]
4: procedure \text{get}\_element(pkt.key, cache\_set, index, SCN, element)
5: \[\text{if} \ cache\_set[index][\text{KEY}\_LOC] == \text{pkt.key} \text{ then}
6: \hspace{1em} cache\_set[index][\text{SCN}\_LOC] \leftarrow \text{SCN}
7: \hspace{1em} \text{element} \leftarrow \text{cache\_set[index]}
8: \text{return element}
9: procedure \text{insert}\_element(cache\_set, keys\_set, index, candidate)
10: \[\text{if} \ cache\_set[index][\text{SCN}\_LOC] < \text{candidate}[\text{SCN}\_LOC] \text{ then}
11: \hspace{1em} \text{temp} \leftarrow \text{cache\_set[index]}
12: \hspace{1em} cache\_set[index][\text{KEY}\_LOC] \leftarrow \text{candidate[KEY}\_LOC]
13: \hspace{1em} cache\_set[index][\text{VAL}\_LOC] \leftarrow \text{candidate[VAL}\_LOC]
14: \hspace{1em} cache\_set[index][\text{SCN}\_LOC] \leftarrow \text{candidate[SCN}\_LOC]
15: \hspace{1em} keys\_set[index] \leftarrow \text{candidate[KEY}\_LOC]
16: \hspace{1em} \text{candidate} \leftarrow \text{temp}
17: \text{return candidate}
18: procedure \text{fetch}(pkt)
19: \hspace{1em} \text{h} \leftarrow \text{pkt.key} \mod d_m
20: \hspace{1em} \text{cache\_set} \leftarrow \text{cache[h]}
21: \hspace{1em} \text{keys\_set} \leftarrow \text{keys\_cache[h]}
22: \hspace{1em} \text{SCN} \leftarrow \text{SCN} + 1
23: \hspace{1em} \text{if} \ \text{index} \in \text{keys\_set}\_\text{contains}(\text{pkt.key}) \text{ then}
24: \hspace{1.5em} \text{element} \leftarrow \text{None}
25: \hspace{1.5em} \text{element} \leftarrow \text{get}\_\text{element}(\text{pkt.key, cache\_set, index, SCN, element})
26: \hspace{1.5em} \text{return element}
27: \hspace{1em} \text{else}
28: \hspace{1.5em} \text{candidate} \leftarrow \text{cache\_set[0]}
29: \hspace{1.5em} \text{cache\_set[0][KEY}\_LOC] \leftarrow \text{pkt.key}
30: \hspace{1.5em} \text{cache\_set[0][VAL}\_LOC] \leftarrow \text{fetch\_value\_from\_storage()}
31: \hspace{1.5em} \text{cache\_set[0][SCN}\_LOC] \leftarrow \text{SCN}
32: \hspace{1.5em} \text{keys\_set[0]} \leftarrow \text{pkt.key}
33: \hspace{1.5em} \text{candidate} \leftarrow \text{insert}\_\text{element}(\text{cache\_set, keys\_set, 1, candidate})
34: \hspace{1.5em} \text{insert}\_\text{element}(\text{cache\_set, keys\_set, 2, candidate})
35: \hspace{1.5em} \text{return cache\_set[0]}

After that, we are iterating over the cache\_set. At each step, we compare the candidate’s SCN to the current element’s SCN in the set. If the SCN is lower (which means the candidate is not the least recently used element), we swap the current element and the candidate in the set (lines 28–35).

4.2.3 LFU Single PKache

LFU maintains the frequency access of each key. Our LFU implementation is very similar to LRU, except for the SCN handling. Here, SCN is incremented each time the element is accessed (frequency count). Since we increment it on each access, we need to add some aging to this mechanism (because we measure the last frequencies and not the total count). Therefore, we decrement the element’s SCN when other elements are accessed.

4.2.4 Hyperbolic Single PKache

Recall that in hyperbolic caching, an item’s priority is set to the number of times the item was accessed since inserted into the cache divided by the duration of time the item remains in the cache. The item whose priority is minimal is deemed the cache victim.

Alas, P4 targets usually support only simple arithmetic operations, such as addition and subtraction, but not multiplication and division. This makes implementing Hyperbolic caching more challenging. Even though such operations exist in the P4 spec [38], popular targets like Intel’s Tofino [3] and Mellanox’ Spectrum [37] do not support them. The main reason is the latency and energy costs of these operations. A multiplication runs many instructions on the CPU compared to addition and subtraction, and in real-
time hardware like a switch, every instruction counts. Hence, to implement Hyperbolic caching in 
PKache, we use two different approaches:

**Semi-Division:** Obtaining an item’s priority in Hyperbolic caching requires division. As mentioned 
above, division is not supported on P4 targets. Following [4], we rewrite the division $\frac{X}{Y}$ as the 
following equation:

$$\frac{X}{Y} = 2^{\log_2(x) - \log_2(y)}.$$ 

In our case, the entire result is not significant. Rather, the goal is to compare the results and free 
the cache slot whose value is minimal. Thus, in PKache we only care about the $\log_2(x) - \log_2(y)$ 
part of the equation. P4 spec does not provide a log operation either, but it provides a cache and 
lookup tables. Our solution involves an extra auxiliary space that is initialized during deployment 
and includes the $\log_2$ values of a specific range of numbers. This means that we add a small error 
since we cannot represent all values.

In summary, we replace the division phase of the counter by the insert time with the subtraction of 
their respective log values. The log values are stored in a register that is initiated at deployment.

**Integer Factor:** A log$_2$ operation returns a floating-point for most numbers. Floating points are not 
supported in P4, therefore most of the results we aim to store are indistinguishable in their integer 
form. Instead, each floating-point is converted into an integer number while a piece of the fractional 
part is represented in an integer form with a pre-defined accuracy error. For instance, storing a 
number with an accuracy of 2 digits after the decimal point transforms the floating-point 123.45678 
into the integer 12345.

Algorithm 3 depicts PKache’s Hyperbolic caching scheme’s code that is generated for a given $km$ ($km = 3$ 
in this example). In Algorithm 3, the first part of fetching a cached value is similar to the previous 
algorithms we have seen (lines 34-36). In case the key exists, we increment its frequency by 1.

Otherwise, we insert the key as the first element like previous algorithms, then set its frequency to 
1 and its insertion time to $SCN$. We also define the previous first element as our victim for eviction 
(lines 39-43).

Next, we iterate over the cache_set. At each step, based on the lifetime period of the elements and their 
frequency, the algorithm fetches the estimated log values for the candidate and the current element (float-
ing points are converted to integers using the integer factor approach). The initialization of the $\log_vals$ 
register is done in line 28. The packet that we send is in the form of $(type=LOG\_INITIALIZATION, 
key=i, value=\lceil \log_2(x) \cdot \text{integerfactor} \rceil)$. The priority is defined by subtracting two log values (lines 11 
and 16). Then, the algorithm compares both the candidate’s and the current element’s priority values and if 
the candidate’s priority is higher, we swap the current element and the candidate in the set (lines 18-23).

4.3 Multiple Cache Region

4.3.1 Differences from a Single Cache Region

PKache supports multiple cache regions, each with its own $d$ and $k$ values. As mentioned above, we 
refer to the first region as window cache and to the second region as main cache. The description below 
focuses on W-TinyLFU, but with very few modifications it can be applied to other multi-region schemes 
like 2Q [25].

In a multi-region cache, we have two ternary tables - one for each region. In case the key exists, the 
algorithm fetches the value from the corresponding cache region directly and returns it in an outgoing 
packet to the client. Otherwise, the algorithm fetches the key from the storage server, and stores the 
key-value pair in the first empty set in the window cache.

In case the window cache set is full, based on the chosen window cache policy, the algorithm chooses 
the candidate for eviction. Then it takes the candidate and inserts it into the main cache if the main 
cache is not full. Otherwise, it will be inserted only if this element is better than one element in the 
main cache according to its policy.
Algorithm 3 A Hyperbolic 3-way cache (automatically generated for \( k_m = 3 \))

1: Initialization:
   2: \( \text{cache} = \text{Register}(d_m) \)
   3: \( \text{packet\_counter} = 0 \)
   4: \( \log\_vals = \text{Register}(\text{MAX\_SCN}) \)

5: procedure \text{get\_element}(\text{pkt.key}, \text{cache\_set}, \text{index}, \text{element})
   6:   if \( \text{cache\_set}[\text{index}][\text{KEY\_LOC}] == \text{pkt.key} \) then
   7:     \( \text{cache\_set}[\text{index}][\text{FREQ\_LOC}] \leftarrow \text{cache\_set}[\text{index}][\text{FREQ\_LOC}] + 1 \)
   8:     \( \text{element} \leftarrow \text{cache\_set}[\text{index}] \)
   9:   return \text{element}

10: procedure \text{insert\_element}(\text{cache\_set}, \text{keys\_set}, \text{index}, \text{candidate}, \text{scn})
   11:   \( \text{element\_freq} \leftarrow \text{cache\_set}[\text{index}][\text{FREQ\_LOC}] \)
   12:   \( \text{element\_lifetime} \leftarrow \text{scn} - \text{cache\_set}[\text{index}][\text{INSERTIME\_LOC}] \)
   13:   \( \text{candidate\_freq} \leftarrow \text{candidate}[\text{FREQ\_LOC}] \)
   14:   \( \text{candidate\_lifetime} \leftarrow \text{scn} - \text{candidate}[\text{INSERTIME\_LOC}] \)
   15:   \( p_{\text{element}} \leftarrow \log\_vals[\text{element\_freq} - \log\_vals[\text{element\_lifetime}]] \)
   16:   \( p_{\text{candidate}} \leftarrow \log\_vals[\text{candidate\_freq} - \log\_vals[\text{candidate\_lifetime}]] \)
   17:   if \( p_{\text{element}} < p_{\text{candidate}} \) then
   18:     \( \text{temp} \leftarrow \text{cache\_set}[\text{index}] \)
   19:     \( \text{cache\_set}[\text{index}][\text{KEY\_LOC}] \leftarrow \text{candidate}[\text{KEY\_LOC}] \)
   20:     \( \text{cache\_set}[\text{index}][\text{VAL\_LOC}] \leftarrow \text{candidate}[\text{VAL\_LOC}] \)
   21:     \( \text{cache\_set}[\text{index}][\text{FREQ\_LOC}] \leftarrow \text{candidate}[\text{FREQ\_LOC}] \)
   22:     \( \text{keys\_set}[\text{index}] \leftarrow \text{candidate}[\text{KEY\_LOC}] \)
   23:   end if
   24:   \( \text{candidate} \leftarrow \text{temp} \)
   25:   return \text{candidate}

26: procedure \text{fetch}(\text{pkt})
   27:   if \( \text{pkt.type} == \text{LOG\_INITIALIZATION} \) then
   28:     \( \log\_vals[\text{pkt.key}] = \text{pkt.value} \)
   29:   return
   30:   end if
   31:   \( h \leftarrow \text{pkt.key} \mod d_m \)
   32:   \( \text{cache\_set} \leftarrow \text{cache}[h] \)
   33:   \( \text{keys\_set} \leftarrow \text{keys\_cache}[h] \)
   34:   if \( \text{index} \leftarrow \text{keys\_set}.\text{contains}(\text{pkt.key}) \) then
   35:     \( \text{element} \leftarrow \text{get\_element}(\text{pkt.key}, \text{cache\_set}, \text{index}, \text{element}) \)
   36:     return \text{element}
   37:   end if
   38:   \( \text{candidate} \leftarrow \text{cache\_set}[0] \)
   39:   \( \text{cache\_set}[0][\text{KEY\_LOC}] \leftarrow \text{pkt.key} \)
   40:   \( \text{cache\_set}[0][\text{VAL\_LOC}] \leftarrow \text{fetch\_value\_from\_storage()} \)
   41:   \( \text{cache\_set}[0][\text{FREQ\_LOC}] \leftarrow 1 \)
   42:   \( \text{cache\_set}[0][\text{INSERTIME\_LOC}] \leftarrow \text{scn} \)
   43:   \( \text{keys\_set}[0] \leftarrow \text{pkt.key} \)
   44:   \( \text{candidate} \leftarrow \text{insert\_element}(\text{cache\_set}, \text{keys\_set}, 1, \text{candidate}, \text{scn}) \)
   45:   \( \text{insert\_element}(\text{cache\_set}, \text{keys\_set}, 2, \text{candidate}, \text{scn}) \)
   46: return \text{cache\_set}[0]
When we have a filter between the caches (for example, the TinyLFU filter), the element is inserted only if the candidate that was evicted from the window cache is worse according to the filter than the candidate for eviction from the main cache. Note that since the main cache and window cache may have different policies, our elements should maintain both SCN’s, so that we can compare elements from the window cache and the main cache correctly (for example, LFU and LRU SCN’s behave quite different).

### 4.3.2 TinyLFU Filter

PKache supports a filter mechanism between the window cache and the main cache. In particular, W-TinyLFU employs the TinyLFU filter [17]. Given an eviction candidate from the window cache, the filter decides based on the recent access history whether it is worth admitting the item into main cache.

There are multiple ways to implement the TinyLFU filter. One efficient way is to use CM-Sketch [14]. Yet, PKache currently implements the filter using an explicit counting structure for all the elements with a de-amortized aging process [17] [31]. The reason we choose to implement the filter with a counting structure instead of CM-Sketch is that our keys are quite small. Further, since we are running on BMv2, we do not have a tight memory limitation. Yet, our implementation can be easily modified to use CM-Sketch, and PKache is independent of the way the filter is implemented.

As described in [17], we need to apply an aging mechanism to the filter to prevent once very popular items from polluting the cache after they stop being popular. In the original W-TinyLFU work, aging is performed by halving all counters once every $W$ accesses, where $W$ are some multiple of the cache size. Alas, this means that once in a while, we have a very long aging operation, which would delay the packet whose treatment invokes this aging process.

Hence, similarly to [31], we de-amortize the aging mechanism to avoid long operations. Specifically, once every $n << W$ accesses, we divide the number of available keys $n$ counters by 2, so that after $W$ turns, all items are divided by 2.

### 4.3.3 Implementation

Algorithm 4 lists the code for the multi-region cache with W-TinyLFU. For clarity, we focus on the multi-cache regions and the filter and encapsulate the single cache regions implementation policies. The filter is shown in lines 24-25. Note that PKache’s insert implementation starts with inserting the element to the first index (line 22), and then calculate the candidate from main cache. Therefore, we compare the candidate that we got from the main cache with the first element (the candidate from window cache) to verify whether our insertion was correct. If not, we will swap the elements. In the latter case, it actually means that the element that is evicted is the element from the window cache whereas the main cache remained intact.

### 4.4 Latency Analysis

When a packet arrives we are performing the following:

- One ternary match table per region to determine whether the key is in the cache or not. There is a read from the register to get the keys.

- In case of a cache hit, we require only one read from the register and one write to the register (read the whole set and then update the whole set after changing the SCN). Since the TCAM match action returns the element’s position, we can access it directly.

- On a cache miss, we iterate over the set as well. The only difference here from a cache hit is that for each way, we read and write auxiliary registers, such as the victim register or the keys register.

Reading and updating the whole set is done at the end of this process. In summary, for most policies, the number of operations on a miss is at most $1 \cdot (TCAM + register\_read + register\_write) + 2k \cdot (register\_read + register\_write)$ and $1 \cdot (TCAM + register\_read + register\_write)$ operations for a cache hit. Hence, hits are handled quickly in $O(1)$ while the latency for handling a miss depends on $k$, due to the need to scan the entire set to find the appropriate victim. Luckily, as we show in Section 5 even when $k = 8$ we obtain comparable results to fully associative caches. In some specific policies there may be a few additional operations. E.g., in Hyperbolic cache we use an extra register for log lookup, and when using the TinyLFU filter we access another register for the filter.
Algorithm 4 Multi Cache Implementation with W-TinyLFU

1: Initialization:
2: main_cache ← Register(\(d_m\))
3: window_cache ← Register(\(d_w\))
4: counting_structure ← Register(number_of_available_keys)

5: procedure fetch(pkt)
6: \(h_{main} \leftarrow \) pkt.key \% \(d_m\)
7: \(h_{window} \leftarrow \) pkt.key \% \(d_w\)
8: main_cache_set ← \text{cache}[\(h_{main}\)]
9: window_cache_set ← \text{cache}[\(h_{window}\)]
10: counting_structure[pkt.key] ← counting_structure[pkt.key] + 1
11: if main_cache_set.contains(pkt.key) then
12: \(element \leftarrow \) get_element_from_main_cache(...)
13: return element
14: else if window_cache_set.contains(pkt.key) then
15: \(element \leftarrow \) get_element_from_window_cache(...)
16: return element
17: else
18: \(window\_candidate \leftarrow \) window_cache[0]
19: \(window\_cache[0] \leftarrow \) create_new_element(pkt.key, ...)
20: \(window\_candidate \leftarrow \) get_candidate_from_window_cache(...)
21: \(main\_candidate \leftarrow main\_cache[0]
22: \(main\_cache[0] \leftarrow window\_candidate
23: \(main\_candidate \leftarrow get\_candidate\_from\_main\_cache(...
24: if counting_structure[candidate] > counting_structure[main_cache[0]][\text{KEY_LOC}] then
25: main_cache[0] ← candidate

4.5 Limitations and Tradeoffs

As described before, there are some restrictions in P4 and its corresponding BMv2 target simulator. Even though we can overcome some of the limitations (like the absence of loops) to implement PKache, there are some restrictions that we cannot. One of the restrictions is that TCAM has a limit on the length of a mask, which is 2048. Thus, if our key is for example 32 bits, it means that \(k\) is limited to 64 entries.

We can, however, alter \(d\) to enlarge our cache size even with this limitation on \(k\). There is a trade-off between large \(k\) and \(d\). On the one hand, a large \(k\) value means that we can do a single comparison on a wide number of elements to check whether the key is inside the cache or not. Also, a large \(k\) value brings the hit ratio closer to that of full associativity, although with very rapidly diminishing returns. However, bigger \(d\) means that elements are more distributed throughout the cache. This reduces the chance that multiple hot items will reside in the same set, thereby improving the cache hit ratio. In the case of a multi region cache, we can use different \(d\) and different \(k\) values for each of the window cache and the main cache.

5 Experimental Results

As mentioned in the introduction, hit ratio is defined as the ratio between the number of accesses that are found in the cache vs. the total number of accesses. Obviously, the main goal of caches is to obtain high hit ratios, as this is how they improve the overall system’s performance. Hence, we apply this metric to evaluate the effectiveness of our solution. In particular, we study the impact on the hit ratio due to the choice of cache policy (among the policies we implemented), different configurations (for both main cache and window cache), varying \(k\) values and total cache size. For our tests, we used both synthetic and real traces with varying levels of frequency distribution skewness. These traces include:

**Multi3** [25]: A trace obtained by executing four workloads (cpp, gnuplot, glimpse, postgres) concurrently.

**Sprite** [25]: A file system trace of the Sprite network which contains requests to a file server from client workstations over a two-day period.

13
Wikipedia(1192951682) [39]: A trace that contains a part of 10% of all user requests issued to Wikipedia (in all languages) during the period of three months at the end of 2007. This specific dataset contains 4.7M items.

OTLP [33]: A file system trace of an OLTP server. Note that in a typical OLTP server, most operations are performed on objects already in memory and thus have no direct reflection on disk accesses.

Gradle [31]: A trace from the Gradle distributed build cache that holds the compiled output so that subsequent builds on different machines can fetch the results instead of building new ones. It is very recency biased. Since it operates as a build cache, edited files need to be compiled, therefore they are accessed. But after some times, they stop being accessed. However, they may remain in the cache in case of frequency based policies, thereby polluting the cache.

Zipf s [43]: An artificially generated datasets of Zipf distributions with parameter $s \in \{0.6, 0.99, 1.5\}$. Each dataset contains 1M items. In a Zipf distribution, the frequency of any element is inversely proportional to its rank in the frequency table. Formally, we denote $N$ the number of elements, $l$ their rank and $s$ the value of the exponent characterizing the distribution (skewness). The frequency of a flow with rank $l$ is calculated as follow: $f(N, l, s) = \frac{1/l^s}{\sum_{n=1}^{N} (1/n^s)}$.

As the skewness factor increases, the frequency difference between one rank to its next gets larger. In highly skewed traces, there are a few very dominant heavy items, so it is easy to track them. When the trace is mildly skewed (heavy-tailed trace), like Zipf0.6, the differences between closely ranked items are minor. Hence, it is much harder to identify the real heavy items that are worth keeping in the cache.

5.1 Single Region Cache

In this section, we examine PKache’s operational envelop over four different aspects:

- The impact of Integer factor when used in Hyperbolic PKache on the hit rate.
- Comparison between PKache and a pure Python implementation (i.e, that is not bounded by P4’s limitations).
- The impact of $k$ given fixed cache size ($k \cdot d = \text{const}$).
- The impact of enlarging the cache size.

5.1.1 PKache Hyperbolic Cache Integer Factor

The first set of experiments studies the impact of the Integer factor on the hit ratio. The results of varying the Integer factor from 0.1 to 1000 with the different traces are listed in Table 1. As shown, increasing the Integer factor improves the hit ratio. Yet, as long as the Integer factor is 10 or above, the difference is at most ±0.01% in these measurements. When the Integer factor is 1 or 0.1, the difference is larger. This is because now many items become indistinguishable, which may result in incorrect evictions. Still, the largest difference we found was below 1.5%. This is expected since in any case, the exact number is only a rough indicator for the probability that an item will be accessed again shortly. In summary, the Integer factor has only a marginal impact on the hit ratio.

There is a tradeoff between the Integer factor and memory consumption. When the former is small, the $\log_2$ lookup table requires fewer entries since its values are small. For example, suppose the maximal value is 2048. For an Integer factor of 0.1 we need just 3 bits to represent the value (2) whereas for an Integer factor of 100, we need 11 bits to represent the value (2000). Hence, for Integer factor 0.1, we need 14 bits in total (3 bits for the log value and 11 bits for key) multiplied by 2048 (number of possible values) - 3.5KB, while for Integer factor of 100, we need 22 bits multiplied by 2048 - 5.6KB (around 160%). From now on, we fix the Integer factor at 100.
Table 1: PKache Hyperbolic with \(k_m = 8, d_m = 16\)

|               | OLTP   | Multi3 | Wikipedia | Zipf0.6 | Zipf0.99 | Zipf1.5 | Sprite | Gradle |
|---------------|--------|--------|-----------|---------|----------|---------|--------|--------|
| PKache (Integer factor=0.1) | 10.0945% | 8.1525% | 23.1059% | 27.1986% | 60.5630% | 91.8984% | 27.0456% | 44.1559% |
| PKache (Integer factor=1)  | 10.1417% | 8.1984% | 23.2129% | 27.3134% | 60.9874% | 92.0045% | 27.1676% | 44.3599% |
| PKache (Integer factor=10) | 10.1596% | 8.2135% | 23.3563% | 27.8849% | 61.2741% | 92.1665% | 27.3398% | 44.8592% |
| PKache (Integer factor=100) | 10.1744% | 8.2390% | 23.9045% | 27.3349% | 61.9211% | 92.5856% | 27.3801% | 44.8999% |
| PKache (Integer factor=1000) | 10.1820% | 8.2799% | 23.9215% | 27.4439% | 61.4347% | 93.0027% | 27.3848% | 44.8603% |

Table 2: PKache and Python LRU with \(k_m = 8, d_m = 16\)

|               | OLTP   | Multi3 | Wikipedia | Zipf0.6 | Zipf0.99 | Zipf1.5 | Sprite | Gradle |
|---------------|--------|--------|-----------|---------|----------|---------|--------|--------|
| Python (Fully associative) | 10.34%  | 8.17%  | 22.21%    | 26.38%  | 61.43%   | 93.33%  | 27.36% | 43.66% |
| Python        | 10.29%  | 8.61%  | 21.56%    | 26.3%   | 61.38%   | 93.43%  | 27.46% | 44.2%  |
| PKache        | 10.29%  | 8.61%  | 21.56%    | 26.3%   | 61.38%   | 93.43%  | 27.46% | 44.2%  |

Table 3: PKache and Python LFU with \(k_m = 8, d_m = 16\) – the reason why for recency biased traces (OLTP, Multi3, Sprite, Gradle) fully associative LFU is worse than limited associativity is explained in the text

5.1.2 Python Implementation vs PKache

Next, we compare PKache to a pure Python implementation, which is not bound by P4’s restricted programming model and is therefore potentially more accurate. We have implemented in Python both a fully associative cache and a \(k\)-way cache. Table 2 exhibits the results for PKache with the LRU cache replacement policy while Table 3 lists the results for LFU. As shown, the LRU results are nearly identical in all implementations.

With LFU the situation is more involved. First, the results of PKache are often worse than the limited associativity Python implementation. This is because in LFU there may be multiple items whose frequency is the same and hence are considered equal eviction candidates. In our Python implementation, such symmetry is broken by preferring to remove the least recently used items. In contrast, in PKache, elements may change their position within their set during the reduction process. When we are looking for a victim, we take as a victim the first element with the lowest SCN. Hence, the symmetry between the same frequency items is broken in a somewhat arbitrary manner. This is not the case with LRU; since every SCN is unique, there is no symmetry to break.

Even though our implementation for LFU in PKache is restricted, the traces exhibit an expected behavior: Zipf traces, Multi3 and Wikipedia, which are more frequency biased in their nature, perform better with LFU than LRU, while OLTP, Sprite and Gradle results are better with LRU. Further, for recency biased traces, fully associative LFU is worse than limited associativity. This is because for recency biased traces, frequency can in fact be an anti-signal. Specifically, past frequent items that are no longer being accessed pollute the cache. In this particular case, limited associativity limits this “damage”. This is further discussed in Section 5.1.3 below.

5.1.3 Impact of Different \(k\) values on PKache

Next, we explore the impact that the value of \(k\) has on measured performance. We run the OLTP, Zipf0.99, Wikipedia, Multi3, Sprite and Gradle traces with all PKache single policies modes and different \(k\) values for the same cache size of 512 items (\(k \cdot d = 512\)) and measured the hit ratio. As can be seen, in the OLTP trace (Figure 3), the impact for LFU, Hyperbolic, and FIFO is marginal and inconclusive. For LRU, the hit ratio increases with the value of \(k\), but it is also merely 0.5%.

In Zipf0.99 (Figure 4), Wikipedia (Figure 5) and Multi3 (Figure 6), the impact for LFU, LRU and FIFO is at most 0.5%. Yet, for Hyperbolic, there is an increase of hit ratio with the value of \(k\) by at most 1.5% (between \(k = 8\) and the rest of the \(k\)'s). This is because the Hyperbolic implementation, as mentioned before, is not accurate. Hence, increasing \(k\) may limit the “damage” caused by these
inaccuracy errors when we potentially replace the “wrong” item (there is a much higher chance of an error when the \( k \) is small) compared to what an implementation that is not bounded by \( P4 \) restricted programming model would do. This is especially true for both Zipf and Wikipedia as they are frequency biased, hence the error in the frequency might play a more significant role.

Gradle’s behavior is non-intuitive (Figure 8). In this trace, increasing \( k \) damages the hit rate, and is especially noticeable for LFU (32.19% vs. 11.92%). When \( k \) is small, we have more sets (\( d \)), so keys are more evenly distributed between the sets. This reduces the likelihood of multiple hot items residing in the same set. In contrast, when \( k \) is bigger, the cache becomes closer to a fully associative one. Frequency is almost an anti-signal for Gradle. Hence, a fully associative cache with LFU performs the worse in this trace.

Another observation is that in the recency biased traces, namely OLTP (Figure 3), Sprite (Figure 7) and Gradle (Figure 8), the best policies are those involving recency (either LRU or Hyperbolic). In Wikipedia (Figure 5), Multi3 (Figure 6) and Zipf (Figure 5) the LFU policy is slightly better due to the traces’ frequency biased nature.

### 5.1.4 The impact of enlarging the cache size

As reported, e.g., in \([1, 16]\), OLTP can obtain a hit rate of over 40% when the cache size is around 1700 items. We would like to test whether we can achieve these high rates in \( PKache \) with a larger cache (the cache size in Figure 3 is 512, and the cache size for Table 2 and Table 3 is 128). To that end, we took \( k_{m} = 16 \) and measured hit rates while varying cache sizes. The policy we chose is LRU, since OLTP is a recency biased trace, so we wanted to take a policy that has to consider recency in its eviction process. Figure 9 shows the hit rate of \( PKache \). As we can see, indeed when we have a larger cache size, the hit rate of \( PKache \) reaches 40% as expected from previous findings.

As reported in \([25]\), Multi3 can obtain a 40% hit rate when the cache size is around 1700 items. Again, we would like to verify that \( PKache \) can equal these numbers. For this, we chose Hyperbolic cache even though this is a frequency biased trace. The reason we chose this policy and not LFU is that we wish to see if we can get to these levels of hit rate even when we are not using the best policy for this trace, and even if the implementation is not purely accurate due to the restricted model in \( P4 \). Figure 10 shows the hit rate of \( PKache \). Indeed, with a larger cache size, the hit rate of \( PKache \) reaches 40% as in previous findings.
In summary, our findings echo previously published results for these traces on fully associative caches with similar cache sizes and cache management policies, e.g., \cite{17, 25, 33}.

5.2 Multiple Region Cache

To test the results of multiple region cache deployments, we took the following configuration $k_w = 4, k_m = 16, d_w = d_m = 16$. We measured the obtained hit ratios with OLTP, Zipf0.99, Wikipedia, Multi3, Sprite and Gradle. We also measured a fully associative cache of the same size that was implemented in Python ($k_w = 64, k_m = 256, d_w = d_m = 1$).

For each trace, we measured the following policy combinations: FIFO in window cache and LRU in the main cache (we denote it FIFO×LRU×TinyLFU), FIFO in the window cache and LFU in the main (denoted FIFO×LFU×TinyLFU), FIFO in the window and LRU in the main without the TinyLFU filter (denoted FIFO×LRU) and LRU in both the window cache and the main (denoted by LRU×LRU×TinyLFU). Note that the latter configuration is the one used in \cite{17, 31}. The results for OLTP are reported in Table 4, for Zipf0.99 in Table 5, for Wikipedia in Table 6, for Multi3 in Table 7, for Sprite in Table 8 and for Gradle in Table 9.

As shown, FIFO×LRU behaves the same between the Python implementation and PKache in all traces, for the reasons discussed above. When we add a TinyLFU filter, the results are no longer the same. This is due to the de-amortized aging process. While the Python implementation for aging frequencies is by dividing all counters by 2 at the same time, in PKache we divide by 2 only some of the counters, but more frequently. Hence, the TinyLFU filter may take somewhat different decisions on whether the element from the window cache should enter the main cache or not.

In general, when the hit rate is higher, the difference between PKache and Python is higher. This makes sense because when the hit ratio is high, the cost of a wrong eviction can be higher, and there is a higher chance that the de-amortization process will have an impact (since not all counters are divided at the same time, the possibility of wrong partial comparison is higher). Still, in Zipf the difference is just over 2%, in OLTP it is less than 1%, and for Wiki, Multi3, Sprite and Gradle is approximately 1.5%.

When comparing the multi-cache region to the single region (Figure 3) in OLTP, even when the cache size is smaller (for the single trace we took $k_m = 16$ and $d_m = 32$, cache size was 512, and now 320), our results for the (FIFO×LRU×TinyLFU are better than the results for just LRU (23.51%, 23.74% vs. 23.44%). The same is true for Multi3 (Figure 10) (34.07%, 34.89% vs. 31.71%). With Wikipedia (Figure 5), our FIFO×LFU×TinyLFU is very close to the LFU configuration in the single region (39.5% vs. 39.76%). Further, for all recency biased traces, (OLTP, Sprite and Gradle), LRU×LRU×TinyLFU is better than FIFO×LRU×TinyLFU, as expected.

Interestingly, in almost all traces, FIFO×LRU obtains lower hit ratios than FIFO×LRU×TinyLFU. The only exceptions are Gradle and Multi3. In Gradle, as we explained earlier, and as can be seen in Figure 8 and Table 9, any consideration of frequency harms the results, so it is not surprising that adding the TinyLFU filter is worsening the hit ratio. In Sprite (Table 8), which is another recency biased trace,
Table 4: OLTP on multi-region with $k_w = 4, k_m = 16, d_w = d_m = 16$

|                  | FIFO×LRU×TinyLFU | FIFO×LFU×TinyLFU | FIFO×LRU | LRU×LRU×TinyLFU |
|------------------|------------------|------------------|----------|-----------------|
| **Python (Fully associative)** | 24.20%            | 13.64%            | 18.81%   | 24.86%          |
| **Python**       | 24.44%            | 12.76%            | 22.71%   | 24.63%          |
| **PKache**       | 23.51%            | 12.01%            | 22.71%   | 23.14%          |

Table 5: Zipf0.99 on multi-region with $k_w = 4, k_m = 16, d_w = d_m = 16$

|                  | FIFO×LRU×TinyLFU | FIFO×LFU×TinyLFU | FIFO×LRU | LRU×LRU×TinyLFU |
|------------------|------------------|------------------|----------|-----------------|
| **Python (Fully associative)** | 78.46%            | 78.61%            | 76.76%   | 76.95%          |
| **Python**       | 78.37%            | 78.9%             | 76.76%   | 76.65%          |
| **PKache**       | 76.32%            | 76.8%             | 76.76%   | 76.39%          |

Table 6: Wikipedia on multi-region with $k_w = 4, k_m = 16, d_w = d_m = 16$

|                  | FIFO×LRU×TinyLFU | FIFO×LFU×TinyLFU | FIFO×LRU | LRU×LRU×TinyLFU |
|------------------|------------------|------------------|----------|-----------------|
| **Python (Fully associative)** | 35.67%            | 43.91%            | 33.02%   | 35.68%          |
| **Python**       | 35.37%            | 41.26%            | 32.66%   | 35.51%          |
| **PKache**       | 33.89%            | 39.5%             | 32.66%   | 34.67%          |

Table 7: Multi3 on multi-region with $k_w = 4, k_m = 16, d_w = d_m = 16$

|                  | FIFO×LRU×TinyLFU | FIFO×LFU×TinyLFU | FIFO×LRU | LRU×LRU×TinyLFU |
|------------------|------------------|------------------|----------|-----------------|
| **Python (Fully associative)** | 58.40%            | 34.31%            | 60.97%   | 59.78%          |
| **Python**       | 59.85%            | 39.01%            | 59.44%   | 59.92%          |
| **PKache**       | 59.15%            | 37.57%            | 59.44%   | 59.12%          |

Table 8: Sprite on multi-region with $k_w = 4, k_m = 16, d_w = d_m = 16$

|                  | FIFO×LRU×TinyLFU | FIFO×LFU×TinyLFU | FIFO×LRU | LRU×LRU×TinyLFU |
|------------------|------------------|------------------|----------|-----------------|
| **Python (Fully associative)** | 35.00%            | 21.00%            | 64.25%   | 35.98%          |
| **Python**       | 36.25%            | 27.63%            | 64.25%   | 36.58%          |
| **PKache**       | 35.72%            | 25.89%            | 64.25%   | 35.98%          |

Table 9: Gradle on multi-region with $k_w = 4, k_m = 16, d_w = d_m = 16$

the difference is very marginal. This is in line with the results published in [17] motivating the use of a two-region configuration with an admission filter between them.

There is an anomaly in Zipf0.99 in which in **Python**, FIFO×LRU is worse than FIFO×LRU×TinyLFU by about 2%, whereas in **PKache** there is much smaller margin (0.4%). This is explained by the fact that **PKache**’s TinyLFU filter implementation is not accurate. Hence, when the hit rate is so large, our mistakes may have a more noticeable impact. Further, with Multi3 in **Python**, FIFO×LFU×TinyLFU is the best (as we can expect), but in **PKache** it is only the second best. This can be explained by the fact that our implementation for LFU is not accurate, so when the trace is frequency biased, our mistakes may have more noticeable impact as well.

6 Discussion

We have presented **PKache**, a generic limited associativity design for implementing caching in the data plane. **PKache** supports multiple popular caching schemes, both when the entire cache is treated as a single region or when we have multiple regions. The limited associativity design helps overcome certain
and PISA limitations, such as lack of iterators, and facilitates bounded fast handling of packets. We enable instantiating PKache with varying parameters by generating its P4 code automatically from our set of Python scripts [22]. Let us emphasize that our goal in this work is not to promote any specific cache management policy. Rather, PKache is intended to be a generic framework that facilitates realizing existing as well as novel policies inside the data plane.

We demonstrated our multi-region capabilities using W-TinyLFU, because it is a recent policy that is also very widely adopted [31, 15], and because its use of an admission filter between the two regions adds an extra challenge. Yet, our multi-region support can be applied to other policies involving more than one region such as the seminal 2Q [28] policy and the recent FRD [35] scheme. Here, the filter would simply always evaluate to true. An interesting challenge left for future work is how to support multi-region caches in which the relative size of each region changes at runtime, such as ARC [33] and Adaptive W-TinyLFU [16].

In this work, we ignored potential differences in object sizes. This is because in many storage systems, cached objects have the same sizes or nearly the same size, often a page or block size. Further, when there are significant size variations, the overall cache area is often partitioned into slabs, where each slab holds objects of similar size and is treated as an independent size oblivious cache, e.g., [10]. A slabling based solution can easily fit limited associativity designs by adding the object’s size range to the hashing function that maps an object to its respective set. We leave combining a limited associativity design with a non-slabbled size-aware cache management policy like [5, 8, 11] for future work.

References

[1] Dolev Adas, Gil Einziger, and Roy Friedman. Limited Associativity Makes Concurrent Software Caches a Breeze. (2021).
[2] Aaron Archer, Kevin Aydin, Mohammad Hossein Bateni, Vahab Mirrokni, Aaron Schild, Ray Yang, and Richard Zhuang. 2019. Cache-Aware Load Balancing of Data Center Applications. Proc. of the VLDB Endowment, 12, 6, (February 2019), 709–723.
[3] [n. d.] Barefoot Tofino. https://barefootnetworks.com/products/brief-tofino/.
[4] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minlan Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic In-band Network Telemetry. In ACM SIGCOMM.
[5] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 389–403.
[6] L. A. Belady. 1966. A Atudy of Replacement Algorithms for a Virtual-Storage Computer. IBM Systems Journal, 5, 2, 78–101.
[7] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. 2016. Heavy Hitters in Streams and Sliding Windows. In INFOCOM, 1–9.
[8] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. 2017. AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 483–498.
[9] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. 2017. Hyperbolic Caching: Flexible Caching for Web Applications. In USENIX Annual Technical Conference (ATC). (July 2017), 499–511.
[10] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: programming protocol-independent packet processors. ACM SIGCOMM Computer Communication Review, 44, 3, 87–95.
[11] Ludmila Cherkasova. 1998. Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy. Technical report. In HP Tech. Report.
[12] Gregory V. Chockler, Danny Dolev, Roy Friedman, and Roman Vitenberg. 2000. Implementing a Caching Service for Distributed CORBA Objects. In IFIP/ACM International Conference on Distributed Systems Platforms (Middleware) (Lecture Notes in Computer Science). Joseph S. Sventek and Geoff Coulson, editors. Volume 1795, 1–23.

[13] Fernando J Corbató and Victor A Vyssotsky. 1965. Introduction and Overview of the Multics System. In Proceedings of the Fall Joint Computer Conference, part I, 185–196.

[14] Graham Cormode and Shan Muthukrishnan. 2005. An Improved Data Stream Summary: the Count-Min Sketch and Its Applications. Journal of Algorithms, 55, 1, 58–75.

[15] Dgraph. Ristretto: A High Performance Memory-Bound Go Cache. https://github.com/dgraph-io/ristretto (2020).

[16] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. 2018. Adaptive Software Cache Management. In Proceedings of the 19th International Middleware Conference, 94–106.

[17] Gil Einziger, Roy Friedman, and Ben Manes. 2017. TinyLFU: A Highly Efficient Cache Admission Policy. ACM Transactions on Storage (ToS), 13, 4, 1–31.

[18] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen I. Kat. 2020. It’s Time to Revisit LRU vs. FIFO. In 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage). Anirudh Badam and Vijay Chidambaram, editors.

[19] Charles Garrod, Amit Manjhi, Anastasia Ailamaki, Bruce Maggs, Todd Mowry, Christopher Olston, and Anthony Tomasic. 2018. Scalable Query Result Caching for Web Applications. Proc. of the VLDB Endowment, (August 2018), 550–561.

[20] Felix Gessert, Michael Schaarschmidt, Wolfram Wingerath, Erik Witt, Eiko Yoneki, and Norbert Ritter. 2017. Quaestor: Query Web Caching for Database-as-a-Service Providers. Proc. of the VLDB Endowment, 10, 12, (August 2017), 1670–1681.

[21] Shahram Ghandeharizadeh and Hieu Nguyen. 2019. Design, Implementation, and Evaluation of Write-Back Policy with Cache Augmented Data Stores. Proc. of the VLDB Endowment, 12, 8, (April 2019), 836–849.

[22] John L. Hennessy and David A. Patterson. 2012. Computer Architecture - A Quantitative Approach (5. ed.) Morgan Kaufmann.

[23] Dor Hovav. 2021. PKache Project. https://github.com/managersop/PKache. (2021).

[24] 2019. Jericho and Trident. https://www.broadcom.com/blog/trident4-and-jericho2-offer-programmability-at-scale. (2019).

[25] Song Jiang and Xiaodong Zhang. 2002. LIRS: an Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance. ACM SIGMETRICS Performance Evaluation Review, 30, 1, 31–42.

[26] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In ACM Proceedings of the 26th Symposium on Operating Systems Principles (SOSP).

[27] [n. d.] Jinja Documentation. https://jinja.palletsprojects.com/en/2.11.x/.

[28] Theodore Johnson and Dennis Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In Proc. of the 20th Int. Conf. on Very Large Data Bases (VLDB), 439–450.

[29] G. Karakostas and D. N. Serpanos. 2002. Exploitation of Different Types of Locality for Web Caches. In Proc. of the 7th Int. Symposium on Computers and Communications (ISCC).

[30] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, En-hong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In Proceedings of the 26th Symposium on Operating Systems Principles (SOSP), 137–152.

[31] Ben Manes. 2012. Caffeine: A High performance caching library for Java 8. https://github.com/benmanes/caffeine.
