Fast Software Cache Design for Network Appliances

Dong Zhou, *Tsinghua University;* Huacheng Yu, *Princeton University;*
Michael Kaminsky, *BrdgAI;* David Andersen, *BrdgAI and Carnegie Mellon University*

https://www.usenix.org/conference/atc20/presentation/zhou

This paper is included in the Proceedings of the
2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the
2020 USENIX Annual Technical Conference
is sponsored by USENIX.

Fast Software Cache Design for Network Appliances

Dong Zhou[♣] [*], Huacheng Yu[◊], Michael Kaminsky[♡], David G. Andersen[♡ ♠]
Tsinghua University[♣], Princeton University[◊], BrdgAI[♡], Carnegie Mellon University[♠]

Abstract

The high packet rates handled by network appliances and similar software-based packet processing applications place a challenging load on caches such as flow caches. In these environments, both hit rate and cache hit latency are critical to throughput. Much recent work, however, has focused exclusively on one of these two desiderata, missing opportunities to further improve overall system throughput. This paper introduces *Bounded Linear Probing* (*BLP*), a new cache design optimized for network appliances. BLP works well across different workloads and cache sizes by balancing between hit rate and lookup latency. To accompany BLP, we also present a new, lightweight cache eviction policy called Probabilistic Bubble LRU that achieves near-optimal cache hit rate (assuming the algorithm is offline) without using any extra space. We make three main contributions: a theoretical analysis of BLP, a comparison between existing and proposed cache designs using microbenchmarks, and an end-to-end evaluation of BLP in the popular Open vSwitch (OvS) system. Our end-to-end experiments show that BLP is effective in practice: replacing the microflow cache in OvS with BLP improves throughput by up to 15%.

1 Introduction

Network virtualization is a core infrastructure component for cloud computing. In virtualized networks, virtual switches route packets between virtual machines (VMs) and between VMs and the outside world. Like the VMs themselves, the virtual switch resides in the hypervisor. The high speed of modern NICs—40Gb/s, 100Gb/s, and even 200Gb/s [2]—makes virtual switches a critical network performance bottleneck.

Many software-based network systems, such as appliances, middleboxes, packet analytic frameworks, and virtual switches, rely on fast flow caches to achieve good average-case performance [10, 38]. These environments impose challenging—and, indeed, somewhat contradictory—requirements upon the caches they use. First, of course, they benefit from high hit rates. But, either to avoid wasting memory or to fit in faster levels of the CPU cache, they also strive to be compact. In addition, because of the high rates at which packet-centric systems operate, the flow cache lookups must have extremely low latency.

These competing requirements place such systems in an interesting middle ground compared to much of the prior work, which usually fall into one of the two extremes. Higher-level caching systems such as web caches and memcached often adopt comparatively expensive cache designs and replacement algorithms to maximize hit rate [9, 26]. On the other hand, CPU caches have such tight timing requirements that they use very simple set associative designs that sacrifice hit rate for extremely low access time measured in clock cycles.

In this paper, we present the design, theoretical analysis, and empirical evaluation of a new cache design called *Bounded Linear Probing*, or BLP, that provides higher cache hit rates than simple set-associative designs, while remaining fast and hardware-friendly. BLP achieves low latency by ensuring *purely local access* to the cache data structure: Look-ups require a single read that spans at most two consecutive CPU cache lines. At the same time, BLP allows *non-local propagation* of full buckets. A basic set-associative cache provides only one location for a given set of objects. BLP allows those objects to creep into later bins, and over repeated inserts and evictions, this property allows high-occupancy bins to shift some of their load to nearby, less-occupied bins. To better serve skewed workloads, we accompany it with a cache eviction algorithm called *Probabilistic Bubble LRU*, or PBLRU, that fulfills the same design goals as BLP: It requires no extra space, adds little latency overhead and achieves near-optimal cache hit rate.

BLP is a simple and effective design for performance-critical software caches; despite its simplicity, we believe it to be a novel design point in the space of "cache table" designs, and provide a theoretical analysis of why it provides an improved hit rate over basic set-associative designs that access the same number of elements. The result is a design that performs nearly as well as the fastest set-associative designs, with hit rates that are closer to that of more advanced, yet expensive, designs such as cuckoo or hopscotch-based caches. We validate these results empirically using both microbenchmarks and by incorporating BLP into Open vSwitch [30], the most popular virtual switch, which is widely used in production. Replacing the microflow cache in OvS with BLP improves throughput by up to 15%: Its lookup latency is about 10 clock cycles longer than that of the basic set-associative design, but BLP's increased cache hit rate more than compensates for the higher latency. In contrast, many of the more expensive

[*]work started while at Carnegie Mellon University

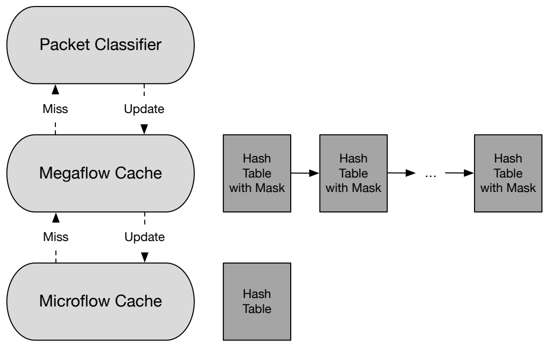


Figure 1: Flow Caching Hierarchy in Open vSwitch

cache designs that can achieve high cache hit rates do not justify their huge latency penalties. Our new cache eviction algorithm PBLRU further improves the throughput by up to 10% even if the workload is only modestly skewed.

2 Flow Caching in Open vSwitch

Open vSwitch achieves high performance through extensive flow caching. Open vSwitch's caching hierarchy consists of three layers: a microflow cache, a megaflow cache, and a caching-aware packet classifier, as illustrated in Figure 1.

The first cache that a packet encounters in OvS is the *microflow cache*, which caches forwarding decisions for each transport connection (or *microflow*). The microflow cache is a hash table that maps microflows to OpenFlow flows if there is an exact match using all the packet header fields. If the packet misses in the microflow cache, then OvS does a lookup in its *megaflow cache*. This cache supports wildcard matching but does not use flow priorities. The megaflow cache is a set of n hash tables, each with a unique wildcard mask. For each hash table, the lookup key is the packet header after applying the mask associated with the table. These hash tables are reactively created and populated by the packet classifier. Because looking up a packet in the megaflow cache searches all n hash tables, it is more expensive than a microflow cache lookup. Therefore, the cache hit rate of the microflow cache (the first cache) is critical to the performance of OvS.

The key observation that inspired our work is that although a microflow cache miss is expensive, it is not *immensely* more expensive than a microflow hit. In typical deployments, where the average number of hash table searches per megaflow lookup is small (as noted in Section 7.2 of Pfaff et al. [30]), the microflow miss penalty is only hundreds to thousands of cycles on modern server CPUs. Hence, making the correct tradeoff between cache hit rate and lookup latency is crucial to the system throughput. In contrast, much of the previous work on software cache designs focuses primarily on improving cache hit rate [9, 14, 18, 13]. In the situations studied by previous work, optimizing for hit rate makes sense: the cache misses in these systems were much more expensive than a hit because they often involved querying very slow backend services such as a database. The cache hit rate, therefore, determines not only the throughput, but also end-to-end request latency [14, 27].

The rest of this paper uses the OvS microflow cache as a case study to analyze and evaluate various design options and demonstrate the effectiveness of our new caching algorithm, Bounded Linear Probing (BLP). We show how BLP can balance cache hit rate (and thus miss penalty) and lookup latency to improve the throughput of Open vSwitch compared to alternate designs.

3 Background and Related Work

3.1 Network Packet and Flow Caching

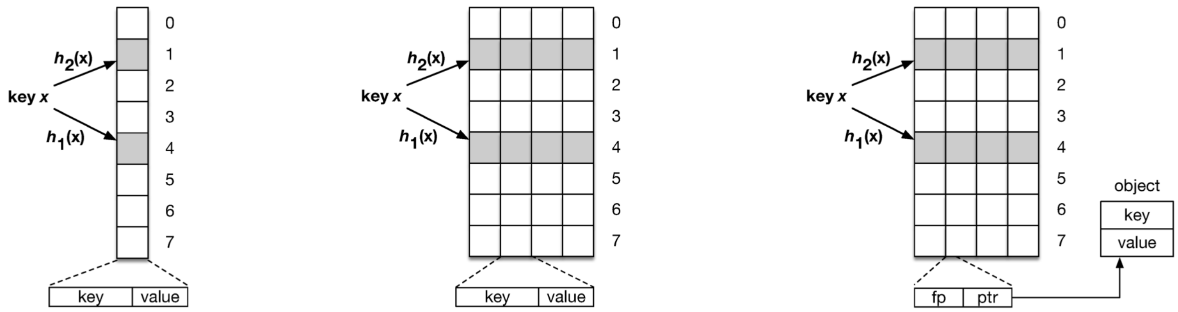
Caching is a common and effective technique for speeding up network packet processing; existing solutions include hardware-based [12, 38, 29] and software-based [10] approaches. Many early hardware routers used flow caching to achieve fast average-case performance. In the modern era, most hardware routers and switches have moved to more costly, but guaranteed-performance designs, such as TCAMs, to be able to provide their maximum forwarding rate under arbitrary (and possibly malicious) traffic. Software switches, however, broadly retain a cache-based design [3, 36].

3.2 Hash Table Options For Caching

Caching is typically managed using a hash table as its basic data structure, but unlike the "full" problem of a general hash table, caches gain an extra degree of freedom: By definition, they do not need to store all possible keys and may choose to evict an existing item.

One of the contributions of this paper is to explore the tradeoff between the cache's hit rate and the lookup/insertion cost imposed by its hash table structure. To illustrate this tradeoff, we begin in Section 4 by describing points that operate at two extremes of the spectrum: First, a basic set-associative cache, in which an item can be stored only in one of m different slots shared by all other items that hash to the same bucket (row) of the hash table. This design is fast but achieves a relatively low hit rate. Next, we introduce two more advanced cache designs that incorporate ideas from cuckoo and hopscotch hashing, which can achieve much higher table occupancy (and thus hit rates), but at the cost of more expensive inserts and lookups. In the rest of this section, we present prior work on fast caches, including a brief introduction to cuckoo and hopscotch hashing.

Cuckoo and hopscotch hashing Cuckoo [28] and hopscotch [21] hashing both aim to achieve high table occupancy (upwards of 90%) in an "open-addressed" hash table design, i.e., one that does not need to use linked lists to store data items. The pointer chasing of a linked-list design adds substantial lookup latency, and the pointers themselves can add substantial memory overhead, especially when the entries in the table are small, which is the case for flow caches.



(a) Basic cuckoo hash table (b) 4-way set-assoc. cuckoo hash table (c) Partial-key cuckoo hashing

Figure 2: Cuckoo Hashing

Both designs share a theoretical basis that allowing items to occupy a small number of slots that are "not too close" to each other allows buckets that would otherwise be too full to spill their contents recursively into other, hopefully less-full buckets. (Recall that when throwing n balls into n bins, the maximally loaded bin will have in the range of $(2 + o(1)) \times \frac{\log n}{\log \log n}$ elements. With 1 million entries in a table, this would result in more than 10 elements falling into the most-loaded bin. Traditional linear probing approaches handle this by keeping the table occupancy under 50%, which conflicts with the desire for high table occupancy.)

Cuckoo hashing Cuckoo hashing [28] maps each key x to two candidate buckets using two hash functions (two is a common choice, but using a larger number of hash functions is also possible). Each key is stored in one of these candidate buckets. Figure 2a shows a basic cuckoo hash table with eight buckets, where key x maps to bucket 4 by $h_1(x)$ and bucket 2 by $h_2(x)$. To lookup a key in a cuckoo hash table, we need to check both candidate buckets to see if the key resides in either bucket. Inserting a new key might relocate an existing key to its alternate bucket. In its basic form, cuckoo hashing provides an expected table load factor of 50%. By using 4-way or higher set-associative buckets, as shown in Figure 2b, the table space utilization increases to over 90% [17].

Partial-key cuckoo hashing [24, 18], illustrated in Figure 2c, extends the basic cuckoo hashing by storing in the table itself only a short hash of the key along with a pointer to the value. This short hash, or *fingerprint*, serves two purposes. First, during lookups, it eliminates the necessity of retrieving full keys unless the fingerprints match. Although false positives could happen, the probability is considerably low. For example, with 1-byte fingerprints, the chance of fingerprint-collision is less than 0.4% ($1/2^8$). Second, fingerprints are enough to derive the alternative bucket for a key, thereby allowing insertion to complete without retrieving the full keys. Replacing full keys with fingerprints substantially improves the memory efficiency and the performance of cuckoo hash tables.

Recent work proposes using a partial-key cuckoo hashing–based key/value cache to improve the performance of OvS (Cuckoo Distributor [37]). For each microflow, it stores a 16-bit fingerprint of the full packet header (key) along with a 16-bit megaflow index (value) in the table. This approach achieves high cache hit rate, but, as we will show in Section 7, the need to access two unrelated cache lines is detrimental to its lookup latency, resulting in lower overall performance.

Full key versus fingerprint The decision to store full keys versus smaller fingerprints goes beyond cuckoo hashing. Open vSwitch itself has two implementations of the microflow cache. The first implementation, called the exact-match cache or EMC [1], matches all the packet header fields to avoid false positives. Because each entry in EMC takes more than 550 bytes, it can only cache a small number of microflows. As of OvS v2.10.1, the default EMC configuration has a capacity of 8192 entries. This limitation causes severe performance drops when the number of active flows is large, as most packets will miss the microflow cache and trigger megaflow lookups [37].

To overcome this shortcoming, OvS introduced a second implementation, called the signature-match cache or SMC [4]. The signature-match cache is a 4-way set-associative cache that maps 16-bit fingerprints, instead of full packet headers, to megaflows. The megaflow entry can then be used for verification of the full flow header. This simple cache design has fast lookup latency; however, as our analysis demonstrates in Section 4, it suffers from low cache hit rate. This is the flow cache upon which the rest of our paper focuses.

Hopscotch Hashing Hopscotch hashing [21] is an open addressed hashing scheme that uses multi-phased probing and displacement to resolve hash collisions. In hopscotch hashing, a key hashed to a bucket will always be stored in that bucket, or in one of the next $H - 1$ buckets, where H is a constant. In other words, a bucket and its next $H - 1$ neighbors form a *virtual bucket* with H slots. To accelerate key lookup and insertion, each bucket maintains an H-bit bitmap, indicating which of the H slots in its virtual bucket contain keys that are hashed to the bucket. These bitmaps are updated during key displacement.

Because all the candidate buckets of a key are contiguous in memory, hopscotch hashing has good cache locality. Our cache design, BLP, resembles hopscotch hashing in the sense that it allows a key to be placed in one of the H buckets starting from the one it is hashed to.

3.3 Hardware Cache Designs

Although the focus of this work is on software caches, there are many parallels to related work on hardware caches.

Cache hit rate versus lookup latency Balancing the cache hit rate and lookup latency has been studied in the context of DRAM hardware caches. Alloy Cache [31], for example, improved performance over prior work by reducing the hit latency, even though doing so slightly reduced the hit rate.

Set-associative caches Hardware caches are often organized into *rows* (i.e., buckets) and *ways* (i.e., slots). An m-way set-associative cache uses a subset of the address bits to index into a row; the cache block (cache line) can be stored in one of the row's m ways. To balance the load across rows, researchers have proposed using a hash of the block address as the index [23] as is commonly done in software-based hash tables and caches.

Skewed-associative caches and cuckoo-like cache designs Skewed-associative caches [35] extend this idea and allow each way to be indexed with a different hash function. In an m-way skewed-associative cache, a cache block B could be stored in row $h_i(B)$ for way i, for $0 \leq i < m$.

Inspired by cuckoo hashing, zcache [34] is an extension of skewed-associative caching. Instead of replacing one of the m existing blocks on a cache miss, it performs a breadth-first search to find additional eviction candidates. After picking a victim entry, it relocates blocks on the cuckoo path to accommodate the new block. These designs are not well-suited for high speed, low latency software caches for packet processing, as they require several cache line reads per lookup.

3.4 Cache Design and Eviction Policy

A large amount of prior work on caching [7, 20, 32, 9, 8, 14, 13] focuses on cache eviction policies. Improved policies, ranging from LRU and LFU to modern alternatives such as LHD [9], increase cache hit rate under skewed workload distributions by biasing eviction towards likely less-useful candidates.

The majority of prior cache eviction algorithms require additional tracking metadata to implement their eviction policies. In contrast, our new algorithm, PBLRU, adds no space overhead. The most related work to our algorithm is an earlier paper by Zhang and Xue [39] that explores the same *bubbling* idea. We discuss the differences between PBLRU and their algorithm, DC-Bubble, in Section 5.3.

4 Design and Analysis

We begin by presenting two baseline cache designs, a set-associative option and a "cuckoo-like" option, and analyze their expected hit rates. We then introduce bounded linear probing and its analysis using the same framework.

To understand the expected hit rate, we assume that the working set is fixed, and that each lookup key is drawn *uniformly* at random from that working set. We only analyze uniform distributions in this section, for the following two reasons: a) prior works studied caching performance on uniformly-distributed workloads [25] and b) the expected hit rate under the uniform distribution is easier to analyze, yet it provides a *lower bound* on the hit rate under any other distribution (see Appendix C for a formal argument). We use α to denote the ratio of the working set size to the number of entries in the cache table, which we call the *oversubscription factor*. When $\alpha < 1$, the cache has more capacity than there are items in the working set. We determine hit rate in terms of α, and then provide numerical interpretations for some values observed in the OvS workloads, such as $\alpha = 0.95$. In Section 7, we show empirical hit rate curves for real implementations across a range of α values.

All of the designs we evaluate use some amount of set-associativity. The caches are partitioned into n buckets, each containing m entries. To determine if an item is in a bucket, the implementation examines whether it is stored in *any* of the entries in the bucket. The table contains a total of $n \times m$ entries, and the working set has size $\alpha \times n \times m$. To store a key-value pair (k, v), one hashes the key and determines in a table-specific way a set of (one or more) buckets that could hold the key, and stores both *fingerprint*(k) and v in an appropriate entry. In OvS, v is a pointer to a megaflow cache entry. Each design uses a different algorithm to decide which entry of the table will store a given pair.

4.1 Analytical Framework for Hit Rate

To analyze the expected hit rate of a cache design, it suffices to estimate the expected number of keys the cache could hold after a sufficiently long warm-up period. This is because, in our formulation, each cache access is uniformly random, so the cache hit rate is equal to the total number of cached keys divided by the size of the working set. Moreover, the number of keys stored in all the cache designs we evaluate never decreases with an increasing number of cache accesses, and it has a maximum value of $n \times m$. Therefore, it will eventually stop increasing. Denote the *final* number of occupied entries in bucket i by c_i. The probability of a cache hit is equal to

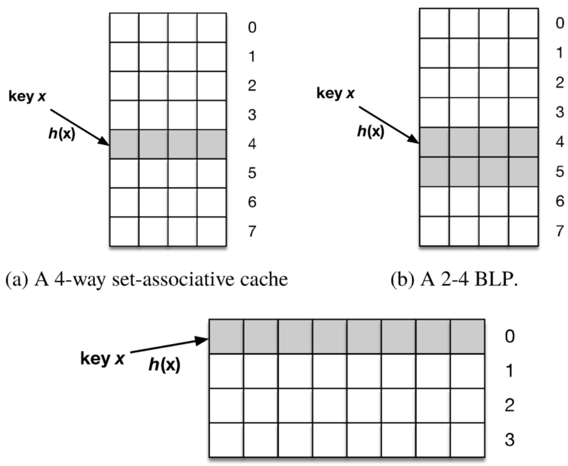
$$\frac{c_0 + \cdots + c_{n-1}}{\alpha n m}. \tag{1}$$

By symmetry, all c_i have the same expected value. Hence, by linearity of expectation, the expected cache hit rate is $\mathbb{E}[c_i]/(\alpha m)$. For each cache design, we describe how its hit rate is estimated from a high level, and leave all the details to appendices.

4.2 Set-associative Cache

We start with a simple design—a set-associative cache. In an m-way set-associative cache, each item is mapped to a bucket by a hash function h, and each bucket has m slots. Figure 3a and Figure 3c show 4-way and 8-way set-associative designs for identical-capacity caches respectively.

The lookup algorithm is straightforward. Given a key k, it checks bucket $h(k)$ for an entry containing *fingerprint*(k). Because it is possible for two different keys to have the same fingerprint, if the fingerprint matches, the full key is fetched to further verify it was indeed the correct one. The chance of fingerprint collision is small, roughly $m/2^{|\text{fingerprint}|}$. Insertion looks for an empty entry in bucket $h(k)$. If one exists, the new key-value pair goes there; if not, we evict a random entry in the bucket to make room for the new item.[1]



(a) A 4-way set-associative cache (b) A 2-4 BLP.

(c) An 8-way set-associative

Figure 3: Set-associative Cache and BLP

We model the expected hit rate when performing a lookup on a uniformly random key from the working set using the standard balls-into-bins problem. We view each key in the working set as a ball and each bucket as a bin. Assuming the hash function is perfectly random, mapping the keys to the buckets can be viewed as placing αnm balls randomly into n bins, where each bin can hold at most m balls. When $\alpha \leq 1$, the data structure as a whole has sufficient *total* space for all balls, i.e., if the balls are placed evenly across the bins, no bin will overflow. However, a small fraction of the bins will still have more than m balls mapped to them. For set-associative caches, that means some keys (balls) will map to such buckets (bins) that are full, triggering a cache miss and an eviction.

Using the aforementioned analytical framework, our analysis shows that $\mathbb{E}[c_i] \approx m - \sum_{t=0}^{m}(m-t)\cdot\frac{(\alpha m)^t}{e^{\alpha m}t!}$. (The full derivation of this result is listed in Appendix A.) Given $\alpha \approx 0.95$ (1,000,000 keys, 2^{20} cache entries), the cache hit rate of a 4-way set-associative cache is $\sim$ 82% and the cache hit rate of an 8-way set-associative cache is $\sim$ 88%. Both 4-way and 8-way set-associative caches have low lookup latency: assuming each item has a size of 32 bits (16-bit fingerprint + 16-bit value), each lookup only requires 1 cache line read. The downside, however, is their relatively low cache hit rates. As we will show in Section 7, low hit rates hurt throughput in many scenarios.

4.3 Cuckoo-lite

The second cache design, which we call *cuckoo-lite*, is very similar to cuckoo hashing. In an H-m cuckoo-lite cache, each bucket has m slots. Each key is mapped to H buckets by hash functions $h_1, h_2, \cdots, h_H$. A 2-4 cuckoo-lite cache shares the same structure as the 2-4 cuckoo hash table (Figure 2b).

Lookup in a cuckoo-lite cache is the same as lookup in a cuckoo hash table. Insertion, however, differs: To insert a key x to a 2-4 cuckoo-lite, we examine whether there is an empty slot in either of x's two candidate buckets. If there is, we insert x there; otherwise, we choose a random slot from the two buckets to replace with x. In other words, instead of searching for a cuckoo path to displace existing keys, cuckoo-lite simply *evicts* one of the existing keys to make room for the new key. This difference highlights the flexibility of a cache over a hash table. In the latter, every inserted key needs to be stored in the table. When there is no empty slot for the new key, the only options are to relocate an existing key via a sequence of key displacements, resize the table, or return an error. A cache, however, can evict arbitrary keys.

Although cuckoo-lite never relocates existing keys, it can achieve at least the same load factor as a cuckoo hash table after a sufficiently long warm-up period.

Following the analytical framework, the expected cache hit rate is at least $\frac{\text{load factor}}{\alpha}$. Prior work [17] estimated the achievable load of cuckoo hashing under various parameters using simulation. In particular, their experimental results showed that for 2-4 cuckoo hashing, 93% load is achievable 99% of the time.[2] Their results, when translated to a 2-4 cuckoo-lite cache, indicate that for $\alpha \leq 0.93$, the expected hit rate must be *at least* 0.99. Therefore, for $\alpha > 0.93$, the expected hit rate is at least $0.99 \times 0.93/\alpha$. Given $\alpha \approx 0.95$, 2-4 cuckoo-lite achieves close to 100% cache hit rate. The drawback, however, is the need to access two unrelated cache lines, which increases lookup latency.

4.4 Bounded Linear Probing (BLP)

We now present our cache design—Bounded Linear Probing, or BLP. The core idea behind BLP is simple: instead of restricting a key to only one bucket, or multiple unrelated buckets, BLP (like hopscotch hashing) can store it at one of the H buckets starting from the one to which it hashes. Unlike hopscotch hashing where each bucket maintains an H-bit bitmap for key displacement, BLP never relocates keys, and searches for all H buckets during the lookup, eliminating the need for per-bucket bitmaps. We call a BLP cache with m-way set-associative buckets, where the key can be in any of the $H-1$ subsequent buckets, an H-m BLP. Figure 3b demonstrates a 2-4 BLP. In this example, key x is mapped to

[1] We use random eviction in all cache designs, and discuss other eviction policies in Section 5.

[2] Their experiments are performed with 64K entries and a fixed amount of work per insertion.

Algorithm 1: `Lookup(k)` for 2-4 BLP

```
f = fingerprint(k);
i = h(k);
if bucket[i] has (f,ptr) then
    if ptr.key == k then
        return ptr.value;
j = (i+1) mod n;
if bucket[j] has (f,ptr) then
    if ptr.key == k then
        return ptr.value;
return not found;
```

Algorithm 2: `Insert(k,v)` for 2-4 BLP

```
f = fingerprint(k);
i = h(k);
if bucket[i] has an empty entry then
    add (f, ptr) to bucket[i];
    return Done;
j = (i+1) mod n;
if bucket[j] has an empty entry then
    add (f, ptr) to bucket[j];
    return Done;
select a random entry e from bucket[i] ∪ bucket[j];
replace e with (f,ptr);
return Done;
```

bucket 4 by hash function h. Therefore, x can be placed in either bucket 4 or bucket 5.

The lookup algorithm is straightforward. Given a key k, if any one of the H buckets contains *fingerprint*(k), the full key and value fetched via the pointer stored in the table. If the full key matches k, the algorithm returns the value, otherwise it returns "not found." See Algorithm 1 for the pseudocode of lookup for 2-4 BLP.

To insert a key-value pair (k,v) to a H-m BLP, the algorithm first checks if there is an empty slot in any one of the H buckets starting from $h(k)$. If so, the item is inserted to the empty slot; otherwise, the algorithm chooses an entry from the H buckets randomly (unless there is an entry with the same fingerprint, in which case the algorithm always chooses the matching entry), and replaces that entry with the new item. Algorithm 2 contains the simplified pseudocode for insertion for 2-4 BLP. Like cuckoo-lite, BLP does not relocate existing keys and reaches its maximum hit rate only after a warm-up period. Appendix D analyzes the relationship between the cache hit rate and the warm-up time in more details.

4.5 Why BLP might be better?

Intuitively, a 2-4 BLP cache should have higher cache hit rate than a 4-way or 8-way set associative cache, but lower hit rate than 2-4 cuckoo-lite. With 32-bit keys, a lookup in 2-4 BLP will read one or two *consecutive* cache lines; therefore, it has higher latency than 4-way and 8-way set-associative cache, but lower than 2-4 cuckoo-lite.

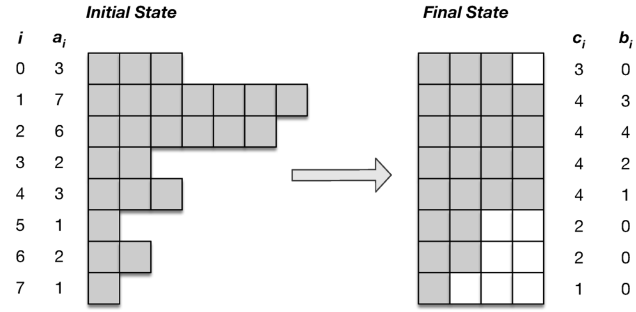


Figure 4: Analysis of BLP

Expected cache hit rate We use the following approach to analyze the expected hit rate of 2-4 BLP. We have two arrays a and b, where a_i is the number of keys from the working set that map to cache bucket i and b_i is the number of keys that spill from bucket i to bucket $i+1$ in the final state (after a sufficiently long warm-up period), for $i = 0,\ldots,n-1$. See Figure 4 for an example.

In the final state, a total of $b_{i-1}+a_i$ keys are served by bucket i. When $b_{i-1}+a_i \le m$, bucket i has enough entries to store all these keys, and no keys will spill to bucket $i+1$, i.e., $b_i = 0$. If $m < b_{i-1}+a_i \le 2m$, bucket i can store m keys, and the extra $b_{i-1}+a_i-m$ will spill to the next bucket.[3] If $b_{i-1}+a_i > 2m$, then bucket i will store m keys, and m of the extra keys will spill to bucket $i+1$. Summarizing the above relations, we have the following equations for b_i and c_i:

$$b_i = \begin{cases} 0 & \text{if } b_{i-1}+a_i \le m, \\ m & \text{if } b_{i-1}+a_i \ge 2m, \\ b_{i-1}+a_i-m & \text{otherwise,} \end{cases} \tag{2}$$

$$\text{and } c_i = \min\{b_{i-1}+a_i, m\}, \tag{3}$$

where we assumed $b_0 = b_n$.

Analyzing b_i By Equation (3), the key to compute $\mathbb{E}[c_i]$ is to estimate the distribution of b_i, which is described by its probability densities $(p_0,\ldots,p_m)$. We observed that b_i has almost identical distribution regardless of i, i.e., the same density vector $(p_0,\ldots,p_m)$ describes all n distributions of $b_0,\ldots,b_{n-1}$. Note that the above observation only concerns the marginal distribution of each b_i, and it does not assert how $b_0,\ldots,b_{n-1}$ are jointly distributed. In fact, b_i and b_j for close i,j can be (very) correlated. We then observe that b_{i-1} is almost independent of a_i, the number of keys mapped to bucket i, and the distribution of a_i has a closed form. Therefore, we are able to derive a system of linear equations on the probability densities based on Equation (2). (The full derivation and solutions are listed in Appendix B.) For $\alpha = 0.95$ and $m = 4$, we calculated $\mathbb{E}[c_i] = 3.59$, and by Equation (1), the expected hit-rate is $\sim 94\%$.

[3] Note that we only allow keys to spill to the next bucket. By definition, b_{i-1} is an integer between 0 and m, which implies that $b_{i-1}+a_i-m \le a_i$. Hence, there are sufficient keys that are mapped to bucket i to spill to the next bucket.

Design	Lookup Speed (cache line reads)	Hit Rate
4-way set-assoc	1	~ 82%
8-way set-assoc	1	~ 88%
2-4 cuckoo-lite	2 *unrelated*	~ 99%
2-4 BLP	1.5 *consecutive*	~ 94%

Table 1: Qualitative comparison of cache designs with α=0.95

Summary Table 1 summarizes 4-way set-associative, 8-way set-associative, 2-4 cuckoo-lite and 2-4 BLP caches in terms of lookup speed and estimated cache hit rate with $\alpha = 0.95$. 2-4 BLP increases the cache hit rate by 12% and 6% compared with 4-way and 8-way set associative caches, respectively, at a cost of 0.5 cache line reads on average.

5 Better Cache Replacement with Probabilistic Bubble LRU

The choice of cache eviction policy is largely orthogonal to the cache design. The cache design proposed in this paper, however, raises interesting and challenging requirements for the cache replacement policy. Like the cache design itself, we need the replacement policy to be fast and memory efficient; we also want a policy that can achieve near-optimal cache hit rate, especially in skewed workloads.

5.1 Traditional LRU approximations are too expensive

Perhaps the most commonly-deployed cache eviction policy is the least recently used (LRU) replacement. Traditional LRU suffers from several drawbacks, including high space overhead [9]. Even its approximations, such as CLOCK [15], requires maintaining 1-bit per key of state [18]. Although many applications can tolerate this amount of space overhead, introducing *any* extra cache management state is likely both to break the cache's careful memory alignment and to increase latency because of state updates.

5.2 Probabilistic Bubble LRU (PBLRU)

To address the aforementioned concerns, we propose a simple cache eviction algorithm that adds *no* extra space and produces near-optimal cache hit rate by combining both recency and frequency information. The core idea behind our algorithm is to assign different priorities to each slot in a bucket: the first slot has the highest priority, the second one has the second highest priority, so forth. After a cache hit, we *promote* the lookup key's priority by exchanging it with the key in the next highest priority slot (unless it is already stored in the first slot of the bucket). When we need to select a victim, we evict the key in the lowest priority slot (i.e., the last slot in a bucket). We call the process of promoting an entry *bubbling* and our algorithm *bubble LRU*. Bubbling effectively achieves the combination of LRU and LFU (least frequently used) without storing any extra information.

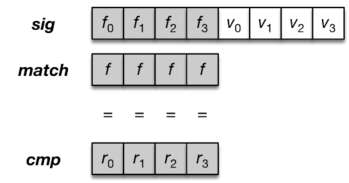


Figure 5: SIMD-optimized 4-way associative cache

Applying Bubble LRU to BLP Applying bubble LRU to set-associative caches is straightforward because each key is mapped to only one bucket. In BLP, however, each key could be placed in two or more consecutive buckets. To avoid introducing more complicated and expensive bucket selection logic, we simply pick a random bucket to apply bubbling.

Probabilistic Bubbling Although the basic bubble LRU does not use any extra space, it still requires a memory write on every lookup, which puts more pressure on the memory subsystem. To alleviate this issue, we adopt an optimization called *probabilistic bubbling*. With probabilistic bubbling, a cache hit does not always promote an entry's position in the bucket. Instead, we promote every n-th hit on average to reduce the number of memory writes. Intuitively, probabilistic bubbling should not hurt the cache hit rate because hot keys will still get promoted more often than cold keys. Our experiments in Section 7 demonstrate that PBLRU substantially improves the cache hit rate with modest overhead of lookup latency.

5.3 PBLRU and DC-Bubble

A non-probabilistic design of the bubble LRU with higher metadata tracking overhead called DC-Bubble was invented in 2009 by Zhang et al. [39]. Instead of always evicting the last entry in a bucket, DC-Bubble implements a more expensive replacement policy that requires updating an extra per-bucket bit on most accesses, which they use to track whether the last access to the same bucket is a hit or a miss. If the last access is a hit, then the last entry is replaced; otherwise, they evict the first entry, promote *all* the rest entries in the bucket, and put the new entry in the last slot.

PBLRU has two advantages over DC-Bubble. First, the authors of DC-Bubble designed their algorithm in the context of hardware cache where operations like setting a flag are cheap, but such operations incur non-negligible overhead in software caches. Second, it is non-trivial to convert their design to a probabilistic version. In our evaluation, we show that PBLRU has substantially lower latency and moderately higher cache hit rate than DC-Bubble.

6 Implementation

We have implemented all of the cache designs in C++. Like stock OvS's SMC (Section 3), we use 16-bit fingerprints and 16-bit values. Within each bucket, we store the fingerprints packed together followed by packed values. We applied several optimizations to improve the cache lookup performance:

SIMD-optimized Lookup To accelerate lookups, we use SIMD instructions to compare multiple fingerprints at the same time (similar to techniques used by Google's Swiss Tables [6]). Figure 5 shows how the lookup works in a 4-way set-associative cache. The stock OvS design does not use SIMD-accelerated reads for its microflow cache, so to ensure a fair basis for comparison, we implemented this optimization and use it as the baseline for comparison.

To search for fingerprint f in a bucket, we first duplicate it four times and store it in a 64-bit integer `match`. Then, we load the first 64 bits of the bucket into another 64-bit integer `sig`. We compare the packed 16-bit integers in `sig` and `match` for equality, storing the results in `cmp`. `cmp` consists of 4 16-bit integers r_0,r_1,r_2 and r_3, where r_i is 0xFFFF if $f_i = f$ and 0 otherwise. We can then count the number of trailing zeros in `cmp` to figure out which slot f matches in the bucket.

Lookup in an 8-way set-associative cache works similarly to the 4-way set-associative cache, but uses 128-bit integers instead of 64-bit integers. For 2-4 cuckoo-lite, because the eight candidate fingerprints are not consecutive, we have to first copy the fingerprints from two buckets into one 128-bit integer, then perform packed integer comparison.

SIMD-accelerated lookup in 2-4 BLP works as follows: The eight candidate fingerprints are not contiguous in memory (unlike cuckoo-lite), but are separated by the 64 packed value bits. Therefore, instead of *copying* fingerprints, we load *both* buckets into a wider 256-bit integer and mask off all the value bits. Eliminating the extra load instruction reduces the lookup latency by $\sim 10\%$ and makes BLP more SIMD-friendly than cuckoo-lite.

Buffer Bucket In 2-4 BLP, if the lookup key hashes to the last bucket of the table, both the first and the last bucket are searched. This corner case requires both a second cache line read and, more importantly, an extra branch. To avoid incurring branch prediction misses, we added a *buffer bucket* following the original cache table. This buffer bucket has minimal impact on the cache miss rate but improves lookup speed: When the lookup key hashes to the last bucket, and that bucket is full, the new key spills to the buffer bucket instead of wrapping around the table. At lookup time, we search both the last bucket and this buffer bucket, which avoids the branch misprediction and allows for processor prefetching[4]. One thing which worth mentioning is that this optimization is specific to BLP and does not work well with other cache designs — it breaks the alignment of the number of buckets (typically a power of 2). Moreover, the extra space given by this buffer bucket is negligible compared to the size of the cache. We therefore only apply the optimization to BLP.

Batched Lookup with Prefetching We use batched lookup with prefetching to overlap bucket computation with memory reads, which minimizes the impact of DRAM access latency. This technique is common in many existing packet processing applications and frameworks [40, 22, 11].

[4]Note that this optimization means that no keys will ever spill *into* the first bucket.

7 Evaluation

We present our evaluation top-down: We begin with a description of the experimental setup followed by a set of end-to-end benchmarks that compare the different cache table designs (described above) in the context of Open vSwitch. These results demonstrate the benefits and generality of BLP in a realistic packet processing application. Next, we use a set of microbenchmarks to understand more deeply the fundamental tradeoffs that each of the cache design brings to the table.

7.1 Experiment Setup

Our experiments are conducted on c220g2 instances from CloudLab [33]. Each of the instances is equipped with the following hardware:

Hardware	Description
CPU	2× Intel Xeon E5-2660v3 CPUs (2.60GHz)
DRAM	160 GiB DDR4 Memory
L3 Cache	2× 24 MiB
NIC	Intel X520 dual-port 10GbE

We also controlled for the following factors, which otherwise had noticeable effects on our results:

Random Number Generator Throughout the experiments, we use PCG-32 [5], a fast and statistically robust algorithm for our random number generation.[5]

Cache Warming As discussed in Section 4, 2-4 cuckoo-lite and 2-4-BLP do not displace keys. Instead, they depend purely on cache warming to reach the maximum hit rate. Therefore, in each experiment, we first warm the testing cache until it reaches a stable state, i.e., the cache hit rate stops increasing.

All experimental results reported below are the average of five runs. The variance was low, so we omit error bars from our graphs. Because the differences between many of the designs are small—in the range of 10% or so—while the absolute performance differences between a high cache hit rate (low alpha) and a low cache hit rate (high alpha) are relatively large, we deliberately choose not to start axes at 0; the graphs are "zoomed-in" to the regions of interest.

7.2 End-to-end Benchmarks

As a concrete end-to-end benchmark using an important application, we modified the microflow cache in Open vSwitch (v2.10.1) to use the various cache designs described above.

Open vSwitch was running on the a c220g2 instance with two 10Gb Ethernet ports, port 0 and port 1. To accurately

[5]The quality of the random number generator directly affects the cache hit rates. Unintended workload locality (i.e., back-to-back keys that hash to nearby buckets) produces higher than expected hit rates; poor random number generators exacerbate this effect. Earlier in the research process for this work, a bad, hand-crafted random number generator caused this issue.

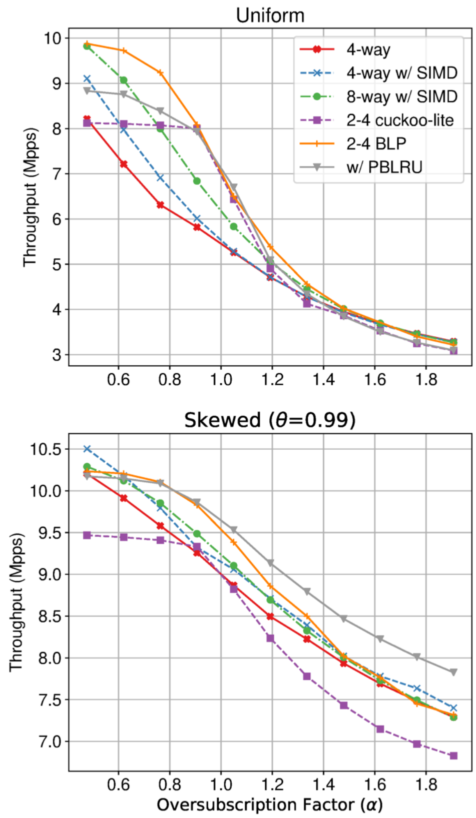


Figure 6: End-to-end Benchmark: Throughput

measure the throughput of Open vSwitch, we instruct the traffic generator to send synthetic network packets to port 0 of OvS and configure flows in OvS to have only one action: send packets out via port 1. This setup allows us to measure how fast OvS can process incoming packets by simply measuring the RX throughput on the traffic generator. We configured Open vSwitch to use one thread per port to handle network packets. On the traffic generator, we used MoonGen [16] to generate minimum-sized (64B) UDP packets. We configured it to use 4 TX threads, because a single thread could not saturate the 10Gb link.

Because public datasets are not readily available[6], we created a synthetic rule set and traffic patterns that creates roughly 20 hash tables in the megaflow cache, similar to the one used by Wang et al. [37]. By default, OvS's SMC has a total of 2^{20} entries (4 MiB), and we use the same cache size in our end-to-end benchmarks. We conducted our experiments with oversubscription factors between 0.5 and 2 because we believe they are the most plausible values in actual deployments – too low of oversubscription factor leads to underutilized caches while too high of oversubscription factor diminishes the effect of caching.

In our experimental result figures, "4-way" means a 4-way set associative cache (OvS's default SMC implementation without our SIMD optimizations), "4-way w/ SIMD" is the same cache design but with SIMD-optimized lookup. "8-way w/ SIMD," "2-4 cuckoo-lite," and "2-4 BLP" are 8-way set associative, 2-4 cuckoo-lite and 2-4 BLP caches respectively, each with their own SIMD-optimized lookups. "w/ PBLRU" is 2-4 BLP with PBLRU instead of random eviction. We use these notations throughout microbenchmarks and end-to-end benchmarks.

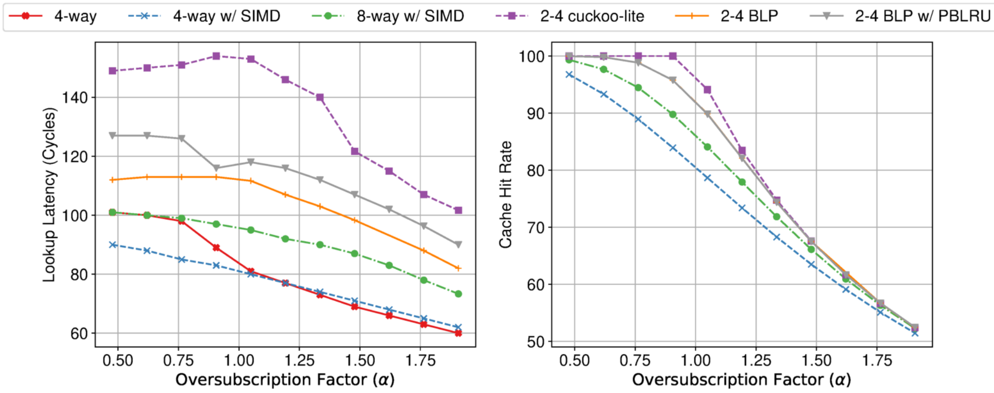
[6]We contacted the lead author of Open vSwitch, who acknowledged this issue.

The overall throughput results are shown in Figure 6. Under a uniform workload, the higher hit rate achieved by BLP causes it to outperform alternative designs through a wide range of α values. Under a Zipf distribution of skewness 0.99 (denoted by θ) [19], BLP similarly outperforms in the middle range, losing to the lower-latency 4-way and 8-way SIMD lookups only at very low and very high oversubscription factors. PBLRU does not help in the uniform distribution, but improves the overall throughput in the skewed distribution. We explore the reasons for these results below.

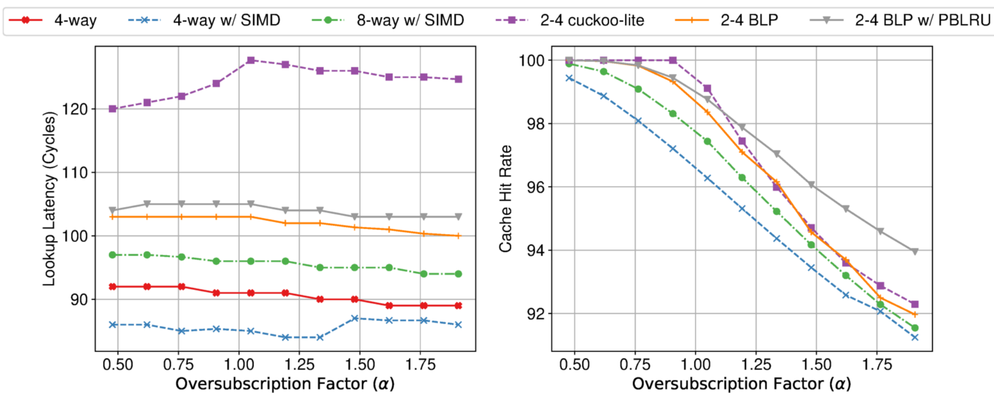
Uniform Workload Figure 7a shows the microflow cache hit rate and microflow cache lookup latency that corresponding to the throughput numbers in Figure 6 (top). Despite 2-4 cuckoo-lite having a higher cache hit rate, and both 4-way and 8-way SIMD having lower lookup latency, 2-4 BLP is the clear winner. It achieves the highest throughput within the entire range of oversubscription factors, outperforming alternative designs by as much as 15%. 2-4 cuckoo-lite's marginal cache hit rate improvement is not enough to compensate for its higher lookup latency. Because PBLRU is unable to improve the cache hit rate in the uniform distribution, it slightly hurts the throughput.

Skewed Workload Figure 7b shows the latency and hit rate results for the Zipf distribution throughput in Figure 6 (bottom). As with the uniform distribution, the simpler 4-way and 8-way lookup designs have lower latency, and 2-4 cuckoo-lite has a mildly higher hit rate. But here too, 2-4 BLP achieves the highest throughput except when the oversubscription factor is very small (close to 0.5) or very large (close to 2). Even in such scenarios, its throughput is very close to the highest. PBLRU further improves cache hit rate, especially when the oversubscription factor is high and increases throughput by up to 7.5%. This is rather expected because when the oversubscription factor is low, using random eviction is enough to get hit rates that are close to 100%.

Summary When used to implement the microflow cache in OvS, 2-4 BLP has the best overall performance. It always achieves the highest, or close to the highest throughput across workloads. Compared to other designs, it improves the throughput by up to 15% with uniform workloads and up to 4% with skewed workloads. PBLRU further improves the throughput with skewed workloads by up to 7.5%.



(a) Uniform Distribution



(b) Skewed Distribution

Figure 7: End-to-end Benchmark: Analysis

7.3 Microbenchmarks

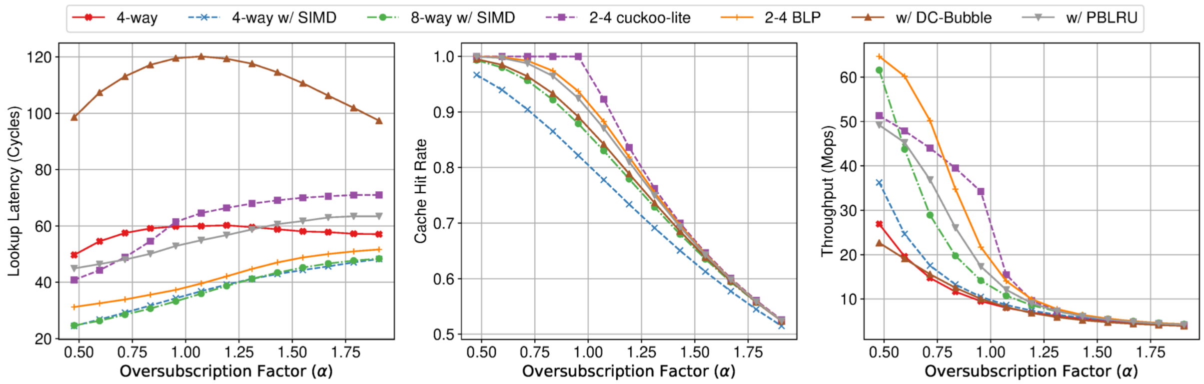
To understand how each cache design performs under different settings, we conducted a series of microbenchmarks outside of Open vSwitch. Each microbenchmark measures lookup latency, cache hit rate, and throughput with a *emulated* cache miss penalty of 1000 cycles. In these experiments, we emulate the cache miss penalty as a fixed cost instead of directing packets into the OvS megaflow lookup process, whose megaflow hash tables are searched can add substantial measurement noise. Because of this decision and because the microbenchmarks lack the instruction and data cache pressure of the full OvS system, the microbenchmark results differ slightly from the end-to-end numbers, but remain useful for exploring why and when the different cache designs excel. For microbenchmarks, we also implemented DC-Bubble [39] and compared it against PBLRU.

We evaluate two cache sizes to understand the performance when the table does and does not fit in the CPU's L3 cache. The workloads are pre-generated to minimize the impact of random number generation; each cache is fed exactly the same set of keys in the same order (for a given trial).

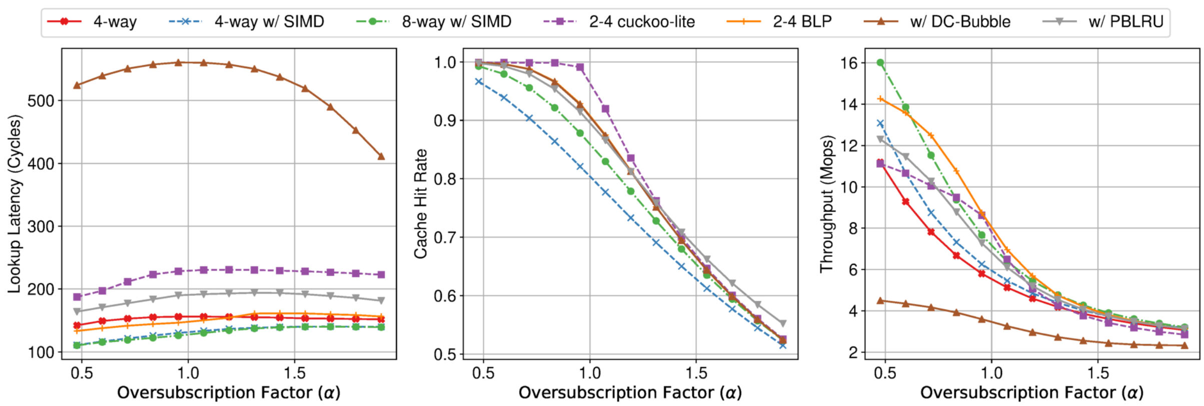
Uniform Workload, Small Cache Size We start by testing the cache designs for a small cache size (4 MiB) under uniform workloads. Figure 8a shows the results.

A few things stand out. As in the end-to-end benchmarks and analytical results, 2-4 cuckoo-lite has the highest cache hit rate for all oversubscription factors and 2-4 BLP has higher cache hit rate than 8-way. Second, 4-way w/ SIMD and 8-way have the lowest lookup latency, between 20 to 50 cycles per lookup. 2-4 BLP's latency is slightly higher and 2-4 cuckoo-lite is the slowest. Third, for oversubscription factors between 0.75 and 1.25, 2-4 cuckoo-lite achieves the highest throughput whereas 2-4 BLP achieves the highest for all other oversubscription factors. This result is because when the cache size is small, it is more likely to reside in the CPU's L3 cache (regardless of caching scheme). Therefore, although 2-4 cuckoo-lite issues one more memory read comparing with other approaches, its increased cache hit rate is enough to overcome the small lookup latency overhead. PBLRU is slower than 2-4 BLP, but much faster than DC-Bubble.

Takeaway When lookup latency overhead is small, higher cache hit rate produces higher throughput.



(a) 4 MiB Cache Size



(b) 64 MiB Cache Size

Figure 8: Micro-benchmark: Uniform Distribution

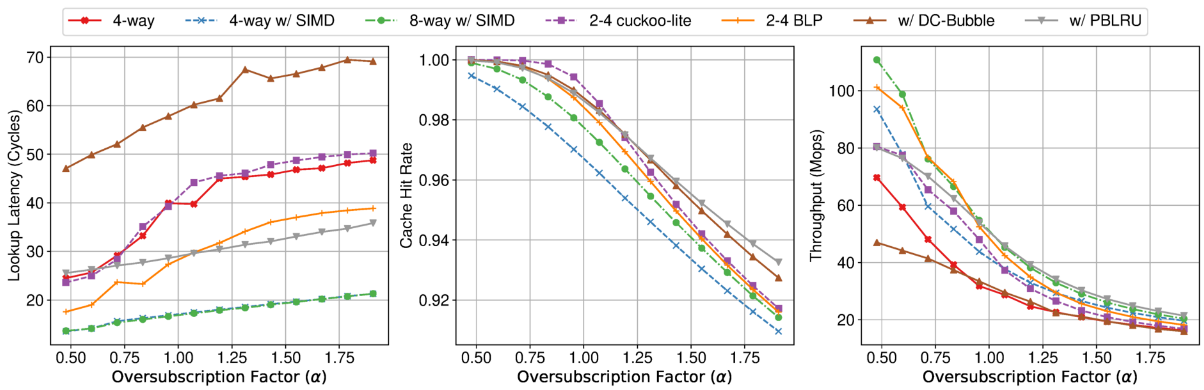
Uniform Workload, Large Cache Size In the second set of microbenchmarks, we increase the cache size to 64 MiB, which is larger than the CPU's L3 cache size. Comparing the results shown in Figure 8b with Figure 8a, the most notable difference is that lookup latencies are higher for all cache designs, especially 2-4 cuckoo-lite. Despite 2-4 cuckoo-lite's advantage in terms of cache hit rate, its overall throughput is lower than 2-4 BLP. Also, when oversubscription factor is small, 8-way outperforms 2-4 BLP because 8-way has a lower lookup latency than 2-4 BLP.

Takeaway When lookup latency overhead is large, lookup latency matters more than cache hit rate.

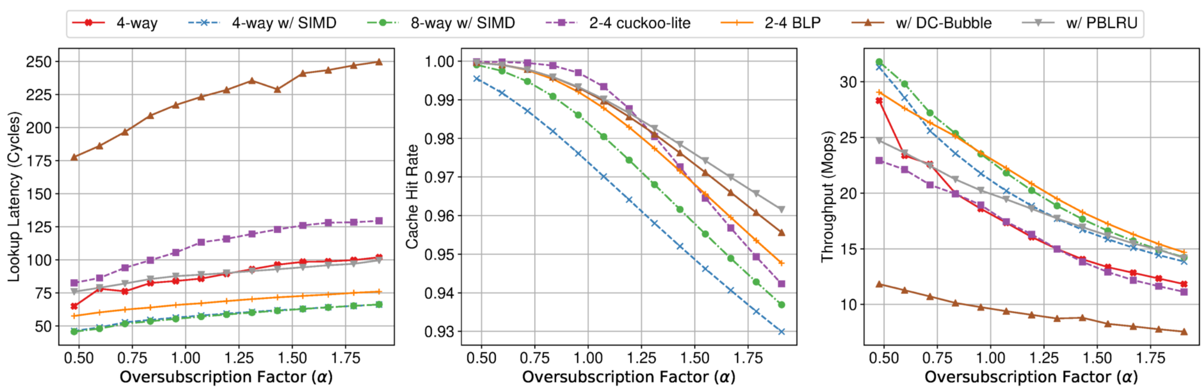
Skewed Workload The third and fourth set of experiments evaluate the cache designs with the same skewed workload (Zipf-distribution with θ=0.99) as in end-to-end benchmarks. Figures 9a and 9b depict the results for small and large cache sizes, respectively. Compared with uniform workloads, the relative order of 4-way w/ SIMD, 8-way, 2-4 cuckoo-lite and 2-4 BLP remains the same for both cache hit rate and lookup latency. However, across all the cache designs, cache hit rates are significantly increased and lookup latencies are significantly lower, which means that skewed workloads *favor* 4-way and 8-way set associative caches. As shown in these two figures, 8-way achieves the highest throughput. 2-4 BLP has much higher throughput than 2-4 cuckoo-lite (only 1-2% behind 8-way). With skewed workloads, PBLRU achieves higher cache hit rate, lower latency, and therefore higher overall throughput than DC-Bubble. Unlike in the end-to-end benchmarks, PBLRU has a lower throughput than 2-4 BLP in these microbenchmarks. This inversion is because the lookup latencies of both designs are substantially lower in the microbenchmarks, which amplifies the significance of the overhead introduced by PBLRU.

Takeaway With skewed workloads, 8-way and 2-4 BLP provide the highest throughput. PBLRU achieves higher cache hit rate than 2-4 BLP with modest overhead.

Summary As demonstrated by the micro-benchmarks above, 2-4 BLP balances cache hit rate and lookup latency across different cache sizes and workload skews. Compared with 4-way and 8-way set-associative caches, it has slightly higher lookup latency (1.5 v.s. 1 cache line reads / lookup), but much higher cache hit rate, especially with uniform workloads. Compared with 2-4 cuckoo-lite cache, 2-4 BLP replaces 2 *distant* cache line reads with one or two *consecutive* cache line reads, and therefore achieves much lower lookup latency at the cost of slightly lower cache hit rate.



(a) 4 MiB Cache Size



(b) 64 MiB Cache Size

Figure 9: Micro-benchmark: Skewed Distribution (θ=0.99)

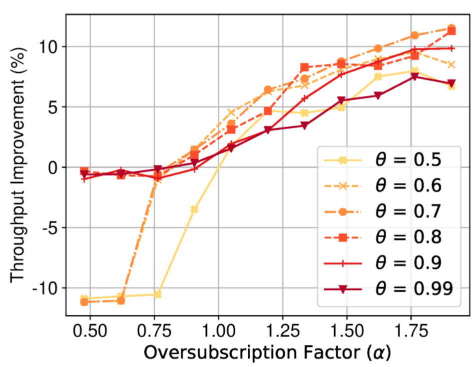


Figure 10: Throughput Improvement of PBLRU over 2-4 BLP

7.4 When should we use PBLRU?

Experiments above have shown that PBLRU outperforms random eviction in certain scenarios. To better understand when should we use PBLRU instead of random eviction, we compared the end-to-end throughput of 2-4 BLP with PBLRU against 2-4 BLP with random eviction using workloads with various skewnesses (θ). Higher θ means the workload is more skewed. Figure 10 illustrates the result.

Two things stand out. First, the performance improvement of PBLRU increases with the oversubscription factor. This is expected because when the oversubscription factor is low, the cache hit rate with random eviction is already high enough so that PBLRU's marginal cache hit rate improvement is not enough to overcome the additional lookup latency. Second, save for a few operating regimes (both skewness and oversubscription factor are low), PBLRU always provides equivalent or better throughput than the baseline.

8 Conclusion

Bounded Linear Probing (BLP) is a new cache design that introduces a new point in the "cache table" design space that balances between cache hit rate and lookup latency. BLP combines the speed of traditional set-associative caches with the high load factor (cache hit rate) and compact size of modern open-addressed hash tables. PBLRU is a new lightweight cache eviction algorithm that is fast and adds no space overhead. Our analysis and microbenchmarks show when each design offers advantages—for which workloads, data sizes, and miss penalties. Finally, we show that replacing the microflow cache in Open vSwitch can improve system throughput by up to 15% and PBLRU further improves the throughput by up to 10% when the workload is skewed.

References

[1] The open vswitch* exact-match cache. https://software.intel.com/en-us/articles/the-open-vswitch-exact-match-cache, .

[2] Mellanox ConnectX-6 product brief. https://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-6_EN_Card.pdf, .

[3] Open vSwitch, . http://www.openvswitch.org.

[4] [ovs-dev] [patch v5 1/2] dpif-netdev: Add smc cache after emc cache. https://mail.openvswitch.org/pipermail/ovs-dev/2018-July/349395.html, .

[5] Pcg, a family of better rarndom number generators. http://www.pcg-random.org, .

[6] Swiss tables and absl::hash. https://abseil.io/blog/20180927-swisstables, .

[7] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez. Scavenger: A new last level cache architecture with global block priority. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 421–432, Dec 2007. doi: 10.1109/MICRO.2007.42.

[8] N. Beckmann and D. Sanchez. Modeling cache performance beyond lru. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 225–236, March 2016. doi: 10.1109/HPCA.2016.7446067.

[9] N. Beckmann, H. Chen, and A. Cidon. Lhd: Improving cache hit rate by maximizing hit density. In *Proc. 15th USENIX NSDI*, Apr. 2018.

[10] F. Chang, W. chang Feng, and K. Li. Approximate caches for packet classification. IEEE INFOCOM, 2004.

[11] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3), Aug. 2007. ISSN 0362-5915. doi: 10.1145/1272743.1272747. URL http://doi.acm.org/10.1145/1272743.1272747.

[12] T.-C. Chiueh and P. Pradhan. Cache memory design for network processors. In *6th International Symposium on High-Performance Computer Architecture (HPCA 2000)*, 2000.

[13] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. USENIX Association, 2015.

[14] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334. USENIX Association, 2017. ISBN 978-1-931971-38-6.

[15] F. Corbato and M. I. O. T. C. P. MAC. *A Paging Experiment with the Multics System*. Defense Technical Information Center, 1968. URL http://books.google.com/books?id=5wDQNwAACAAJ.

[16] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 15nd ACM SIGCOMM conference on Internet measurement*, IMC '15, 2015.

[17] U. Erlingsson, M. Manasse, and F. Mcsherry. A cool and practical alternative to traditional hash tables. In *Proc. Seventh Workshop on Distributed Data and Structures (WDAS'06)*, Jan. 2006.

[18] B. Fan, D. G. Andersen, and M. 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. USENIX, 2013. ISBN 978-1-931971-00-3.

[19] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, May 1994.

[20] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, pages 107–116, June 2000. doi: 10.1145/339647.339660.

[21] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *Proceedings of the 22Nd International Symposium on Distributed Computing*, 2008.

[22] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen. Raising the bar for using GPUs in software packet processing. In *Proc. 12th USENIX NSDI*, May 2015.

[23] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using prime numbers for cache indexing to eliminate conflict misses. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 288–299, Feb 2004. doi: 10.1109/HPCA.2004.10015.

[24] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[25] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proc. 11th USENIX NSDI*, Apr. 2014.

[26] Memcached. Memcached: A distributed memory object caching system. http://memcached.org/, 2011.

[27] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398. USENIX, 2013. ISBN 978-1-931971-00-3.

[28] R. Pagh and F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, May 2004.

[29] C. Partridge and Others. A 50-Gb/s IP router. *IEEE/ACM Transactions on Networking*, 6(3):237–248, June 1998.

[30] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2015)*, 2015.

[31] M. K. Qureshi and G. H. Loh. Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design. In *Proc. ACM MICRO*, 2012.

[32] M. K. Qureshi, D. Thompson, and Y. N. Patt. The v-way cache: demand-based associativity via global replacement. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 544–555, June 2005. doi: 10.1109/ISCA.2005.52.

[33] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 2014.

[34] D. Sanchez and C. Kozyrakis. The zcache: Decoupling ways and associativity. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 187–198, Dec 2010. doi: 10.1109/MICRO.2010.20.

[35] A. Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 169–178. ACM, 1993. ISBN 0-8186-3810-9. doi: 10.1145/165123.165152. URL http://doi.acm.org/10.1145/165123.165152.

[36] N. Shelly, E. J. Jackson, T. Koponen, N. McKeown, and J. Rajahalme. Flow caching for high entropy packet fields. In *Proceedings of the third workshop on Hot topcis in software defined networking*, HotSDN '14, 2014.

[37] Y. Wang, T.-Y. C. Tai, R. Wang, S. Gobriel, J. Tseng, and J. Tsai. Optimizing open vswitch to support millions of flows. In *Proceedings of the 2017 IEEE Global Communications Conference (GLOBECOM 2017)*, 2017.

[38] J. Xu, M. Singhal, and J. Degroat. A novel cache architecture to support layer-four packet classification at memory access speeds. IEEE INFOCOM, 2000.

[39] C. Zhang and B. Xue. Divide-and-conquer: A bubble replacement for low level caches. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 80–89. ACM, 2009. ISBN 978-1-60558-498-0. doi: 10.1145/1542275.1542291. URL http://doi.acm.org/10.1145/1542275.1542291.

[40] D. Zhou, B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proc. 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Dec. 2013.

A Expected Cache Hit Rate of Set-associative Caches

In a m-way set-associative cache with n buckets, for each bucket, the probability that there are exactly t keys mapped to it is

$$\binom{\alpha nm}{t} \cdot n^{-t} \cdot (1 - 1/n)^{\alpha nm - t}$$
$$= \frac{\alpha nm(\alpha nm - 1) \cdots (\alpha nm - t + 1)}{t! \cdot n^t} \cdot (1 - 1/n)^{\alpha nm - t}$$

which when $t \ll \alpha nm$, is approximately

$$\frac{(\alpha nm)^t}{t! \cdot n^t} \cdot (1 - 1/n)^{\alpha nm} = \frac{(\alpha m)^t}{t!} \cdot (1 - 1/n)^{\alpha nm}$$

which by the fact that $1 - \epsilon \approx e^{-\epsilon}$ for small ϵ, is approximately

$$\frac{(\alpha m)^t}{t!} \cdot e^{-\alpha nm/n} = \frac{(\alpha m)^t}{e^{\alpha m} t!}$$

If $t \le m$, then all t keys will be cached; otherwise, only m will be cached. Therefore, the expected number of keys that are cached in a bucket is approximately

$$\sum_{t=0}^{m} t \cdot \frac{(\alpha m)^t}{e^{\alpha m} t!} + \sum_{t=m+1}^{\infty} m \cdot \frac{(\alpha m)^t}{e^{\alpha m} t!},$$

which by the fact that $\sum_{t \ge 0} \frac{(\alpha m)^t}{e^{\alpha m} t!} = 1$, is equal to

$$= m - \sum_{t=0}^{m} (m - t) \cdot \frac{(\alpha m)^t}{e^{\alpha m} t!},$$

Hence, the expected cache hit rate is $(m - \sum_{t=0}^{m} (m - t) \cdot \frac{(\alpha m)^t}{e^{\alpha m} t!})/(\alpha m)$.

B Expected Cache Hit Rate of 2-4 BLP

Recall that a_i is the number of keys from the working set that map to cache bucket i and b_i is the number of keys spill from bucket i to $i + 1$ after a sufficiently long warm-up period. For $j \ge 0$, we have:

$$\Pr[a_i = j] = \binom{\alpha nm}{j} \cdot n^{-j} \cdot (1 - 1/n)^{\alpha nm - j}.$$

By the law of total expectation and Equation (2) in Section 4.5, for $0 < l < m$, we have

$$p_l \approx \begin{cases} \sum_{j'=0}^{m} \left(\sum_{j=0}^{m-j'} \Pr[a_i = j] \right) \cdot p_{j'} & \text{if } l = 0, \\ \sum_{j=l}^{l+m} \Pr[a_i = j] \cdot p_{l+m-j} & \text{if } 0 < l < m, \end{cases}$$

and by the definition of probability distribution,

$$p_0 + \cdots + p_m = 1.$$

Solving the above system of linear equations for $(p_0, \ldots, p_m)$ with $m = 4$, $\alpha = 0.95$ gives us $p_0 = 0.37889778$, $p_1 = 0.15160669$, $p_2 = 0.14369602$, $p_3 = 0.12041777$ and $p_4 = 0.20538175$. By Equation (3), we get $E[c_i] = 3.59$.

C Expected Hit Rate of BLP under Non-Uniform Distributions

In the following, we prove that the expected hit rate obtained in Section 4.5 for the uniform distribution is always a lower bound for any other distribution. Fix any distribution over the working set S, let p_x be the probability of key x. Without loss of generality, we may assume $p_x > 0$ for all $x \in S$, since otherwise, we could simply remove all x with zero probability from the working set. Recall that c_i denotes the *final* number of occupied entries in bucket i. Observe that $c_0, \ldots, c_{n-1}$ are determined only by the hash function, i.e., how many keys are mapped to each bucket. They do not depend on the probability distribution of the keys (as long as all keys have non-zero probability), the distribution only affects how fast the final numbers are achieved.

After a sufficiently long warm-up period, all buckets achieved their final numbers of occupied entries. Now, consider all possible memory configurations *after* the warm-up. Further key lookups define a *Markov chain* over them, where the transition probability from memory configuration A to configuration B is the probability that A becomes B after one lookup. Observe that this Markov chain is *aperiodic* (i.e., there does not exist a $t > 1$ and a state A such that A can only go back to itself after steps of multiples of t). It is well-known that for any aperiodic Markov chain and any initial state, as the number of steps (key lookups) increases, the distribution of the state will approach *some* final stationary distribution (note that the stationary distribution may not be unique, hence the final stationary distribution may depend on the initial state). The *final* hit rate is computed from this stationary distribution and the distribution of the keys. More specifically, let q_x be the expectation, over a random hash function and random lookups in the warm-up period (which determine the Markov chain and the distribution of initial state), of the probability that key x is cached according to the final stationary distribution. Thus, the expected hit rate is equal to $\sum_{x\in S} p_x q_x$.

Next, by linearity of expectation, $\sum_{x\in S} q_x$ is equal to the expected total number of occupied entries in the data structure, $\mathbb{E}[c_i]\cdot n$. The key observation is that if $p_x \geq p_y$ then $q_x \geq q_y$, i.e., if a key is more likely to occur, then it has a higher probability to appear in the final stationary distribution, *over a random hash function and warm-up period*.[7] Let $S_{\text{high}} := \{x : p_x \geq 1/|S|\}$ be the set of keys that occur with at least the average probability and $S_{\text{low}} := \{x : p_x < 1/|S|\}$ be the set of keys that occur with probability lower than the average, and let $q := \min_{x\in S_{\text{high}}} q_x$. Hence, $q \geq q_x$ for all $x \in S_{\text{low}}$. We have

$$\begin{aligned}\sum_{x\in S} p_x q_x &= \sum_{x\in S}\frac{q_x}{|S|} + \sum_{x\in S}(p_x - 1/|S|)q_x \\ &= \frac{1}{|S|}\sum_{x\in S} q_x + \sum_{x\in S_{\text{high}}}(p_x - 1/|S|)q_x + \sum_{x\in S_{\text{low}}}(p_x - 1/|S|)q_x\end{aligned}$$

which by the fact that $p_x \geq 1/|S|$ and $q_x \geq q$ for $x \in S_{\text{high}}$, and the fact that $p_x < 1/|S|$ and $q_x \leq q$ for $x \in S_{\text{low}}$, is at least

$$\begin{aligned}&\geq \frac{1}{|S|}\sum_{x\in S} q_x + \sum_{x\in S_{\text{high}}}(p_x - 1/|S|)q + \sum_{x\in S_{\text{low}}}(p_x - 1/|S|)q \\ &= \frac{\mathbb{E}[c_i]\cdot n}{|S|} + \sum_{x\in S} p_x\cdot q - \sum_{x\in S}\frac{1}{|S|}\cdot q \\ &= \frac{\mathbb{E}[c_i]\cdot n}{|S|} + q - q \\ &= \frac{\mathbb{E}[c_i]\cdot n}{|S|}.\end{aligned}$$

The last quantity $\frac{\mathbb{E}[c_i]\cdot n}{|S|}$ is precisely the expected hit rate under the uniform distribution, as we argued in Section 4. Therefore, the expected hit rate under any non-uniform distribution is always lower bounded by the hit rate under the uniform distribution.

D Analysis on Warm-up Time

In the following, we present an informal estimation on the relationship between the hit rate of BLP and its warm-up time. As we argued in Section 4.5, the hit rate is equal to the number of occupied entries in the BLP divided by the size of the working set. In each lookup in the warm-up period, the key may be either a) in the BLP already, or b) not in the BLP and the buckets are full, or c) not in the BLP and the bucket is not full. Only in case c), do we increase the number of occupied entries by one. Denote the *final* hit rate by $r_{\max}$. When the current hit rate is r, we are going to *approximate* the probability of case c) by $r_{\max} - r$. That is, we assume there is a fixed set of $(r_{\max} - r)\cdot(\alpha mn)$ keys in the working set such that they are the missing keys from the BLP in order to achieve the maximum hit rate of $r_{\max}$.

Therefore, let L be the length of the warm-up, we have

$$\frac{\mathrm{d}r}{\mathrm{d}L} = \frac{r_{\max} - r}{\alpha mn},$$

and when $L = 0, r = 0$. By solving this ordinary differential equation, we obtain

$$r = r_{\max}(1 - e^{-\frac{L}{\alpha mn}}).$$

That is, when the length of the warm-up is a large constant times the working set size, the estimated hit rate becomes very close to $r_{\max}$. For $\alpha = 0.95$, we have verified by experiments that a warm-up period of length 20 times the working set size is sufficient to obtain a hit rate that is less than 1% lower than $r_{\max}$.

[7]Note that this is not true if the hash function is fixed.