Data Structure Engineering for High Performance Software Packet Processing

Dong Zhou
dongz@cs.cmu.edu

January 18, 2019

THESIS PROPOSAL

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
David G. Andersen (Chair)
Michael Kaminsky[⋆],
Justine Sherry,
Sylvia Ratnasamy[†]

Carnegie Mellon University, [⋆]Intel Labs, [†]University of California, Berkeley

Keywords: Data structure, software switch, cuckoo hashing, scalability, packet processing

1 Introduction

Compared with using specialized hardware, software packet processing on general-purpose hardware provides extensibility and programmability. From software routers [17] to virtual switches [5] to Network Function Virtualizations (NFV) [40], we are seeing increasing applications of software-based packet processing. However, software-based solutions often face performance challenges, primarily because general-purpose CPUs are not optimized for processing network packets. To this end, many research efforts have suggested improving performance by distributing the workload across multiple servers [17], adopting accelerators such as GPUs [25, 30, 23], and/or algorithm specialization [41].

This thesis tackles the performance challenge of software packet processing by optimizing the *main data structures* of the application. We observed that for a wide range of packet processing applications, performance is bottlenecked by one or more data structures. To demonstrate the effectiveness of our approach, this thesis examines three applications: Ethernet forwarding, LTE-to-Internet gateways, and virtual switches. For each application, we propose algorithmic refinements and engineering principles to improve its main data structures. Our solutions benefit from a set of novel techniques that are *theory-grounded* and optimized by architectural-aware and workload-inspired observations, including:

- **A concurrent, read-optimized hash table for x86 platform**: Based on memory-efficient, high-performance concurrent cuckoo hashing [19], we propose a set of algorithmic and architectural optimizations to craft an x86-optimized, high-performance hash table that allows multiple readers and a single writer to operate on the same time. Our design focuses on read performance because the workload for Ethernet switches is read-heavy.[1]

- **An extremely compact data structure for set separation**: Extending prior work by Fan et al. [20], we propose *SetSep*, a data structure that divides a set of input elements into a small number of disjoint subsets in a extremely compact manner. This data structure is the backbone of our new design for building scalable, clustered network appliances that must "pin" flow state to a specific handling node without being able to choose which node that should be. We show how our design improves the performance and scalability of software LTE-to-Internet gateway.

- **(Proposed work) Bounded linear probing cache**: We propose a *bounded linear probing cache*, a new cache design that offers low cache miss rate and high throughput at the same time. We demonstrate it can be used in Open vSwitch to improve its datapath performance.

In many, if not all the cases, our improvements toward those data structures focus on the theme of *memory efficiency*. The benefits of memory efficient data structures are twofold. First, for a working set with fixed size, being more memory efficient means smaller memory footprint, which often translates to higher *performance*. Second, for a fixed amount of memory, being more memory efficient means a system can handle larger work sets, which makes it possible to push applications we explored to unprecedented *scales*. In this dissertation, we use several networking examples in which we pursue extreme degrees of scalability, such as support billions of forwarding table entries. We choose these goals for several reasons: First and foremost, because in doing so, we realize many of the gains described above for more practically-sized applications while unearthing novel approaches to do so. And second, we have observed the rapid growth in scale of enterprise networks and datacenter networks in recent years, both because of the adoption of high-bisection topologies and the emergence of low-cost, high-data-rate

[1] In contrast, the improvements proposed by Li et al. [33] in a later work primarily optimize for write performance of concurrent cuckoo hash table.

switches from vendors such as Arista and Mellanox. This growth means that it is not inconceivable to have hundreds of thousands of devices interconnected in a single, extremely large network. The addition of virtualization further increases this scale. Moreover, new network designs such as software-defined networking (SDN) and content-centric networking (CCN) may require or benefit from extremely large, fast forwarding tables. For example, source+destination address flow routing in a software defined network can experience quadratic growth in the number of entries in the FIB; incorporating other attributes increases table sizes even more. Many CCN-based approaches have lookup tables that contain an entry for every cachable chunk of content. To address these challenges, Section 3 describes a FIB design that can handle tens of Gbps of network traffic while maintaining a forwarding table of up to *one billion* entries.

We believe these techniques also apply to many other applications in addition to the ones we examined in this thesis. The road map of this proposal is structured as follows:

Contents	**Section**
Scalable, High Performance Ethernet Forwarding with CuckooSwitch	3
Scaling Up Clustered Network Appliances with ScaleBricks	4
Fast Flow Caching in Open vSwitch with Bounded Linear Probing Cache	5

Contribution from collaborators

- Michael Mitzenmacher contributed the idea of two-level hashing in *SetSep*.

2 Background: Userspace Packet I/O

The basic building block for all the high performance packet processing applications is a fast packet I/O framework. Several such frameworks have been introduced in the past, including PacketShader's I/O engine [25], Netmap [42], RouteBricks [17] and the Intel Data Plane Development Kit (or DPDK) [28]. In order to acheive high throughput, all the frameworks deliver batches of packets to processing threads to be processed together, a feature that our proposed techniques rely upon. Therefore, our improvements are independent from packet I/O frameworks. Because we use unmodified DPDK in our systems for high-throughput packet I/O, this section provides the salient details about it.

DPDK is a set of libraries and optimized NIC drivers designed for high-speed packet processing on x86 platforms. It places device drivers in user-space to allow zero-copy packet processing without needing kernel modifications. For efficiency, it batches received packets and hands them to applications together. These techniques combine to provide developers with a straightforward programming environment for constructing extremely high-performance packet processing applications. Figure 1 shows the high level architecture of Intel DPDK. Our systems use the DPDK for all of its packet I/O. Readers familiar with prior work such as RouteBricks [17] and the PacketShader IO-engine [25] will likely recognize the DPDK as a high-performance, industrially-engineered successor to these approaches.

Three aspects of the DPDK are particularly relevant to the design and performance of our systems:

First, the DPDK's Memory Manager provides NUMA (Non-Uniform Memory Access)-aware pools of objects in memory. Each pool is created using processor "huge page table" support in order to reduce TLB misses. The memory manager also ensures that all objects are aligned properly so that access to them is spread across all memory channels. We use the DPDK's memory manager for all data structures described in this thesis, and as a consequence, they benefit from this NUMA awareness, alignment, and huge page table support.

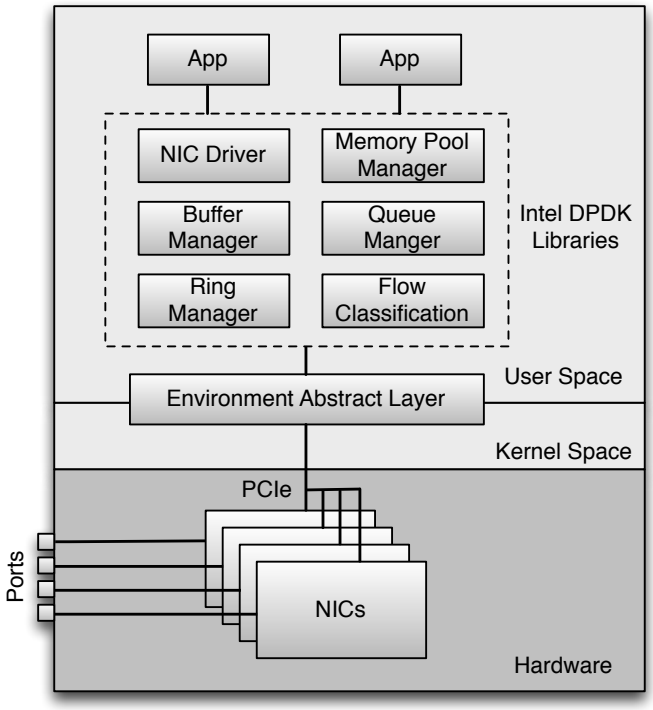


Figure 1: Architecture of Intel DPDK.

Second, the DPDK's user-space drivers operate in polling mode, eliminating interrupt overhead. This speeds up processing, but also consumes CPU. Our results are therefore most relevant to dedicated switching and packet processing scenarios where the continual CPU overhead of polling does not interfere with other tasks on the machine.

Third, the DPDK delivers packets in large batches for efficiency. As a result, many of our techniques also emphasize batching for efficiency in a way that is aligned with the DPDK's batching.

3 Scalable, High Performance Ethernet Forwarding with CuckooSwitch

This section explores a question that has important ramifications for how we architect networks: Is it possible to, and how can we, build a high-performance software-based switch that can handle extremely large forwarding tables, on the order of millions to billions of entries. Our hope is that by doing so, we contribute not just a set of concrete techniques for architecting the forwarding table (or FIB) of such a device, but also show that, indeed, network designs that require huge FIBs *could* be implemented in a practical manner.

We propose the high-performance CuckooSwitch FIB, which provides the basis for building scalable and resource-efficient software-based Ethernet switches. CuckooSwitch combines a new hash table design together with DPDK to create a best-of-breed software switch. The forwarding table design is based upon memory-efficient, high-performance concurrent cuckoo hashing [19], with a new set of architectural optimizations to craft a high-performance switch FIB. The FIB supports many concurrent reader threads *and* allows dynamic, in-place updates in realtime, allowing the switch to both respond instantly to FIB updates and to avoid the need for multiple copies to be stored.

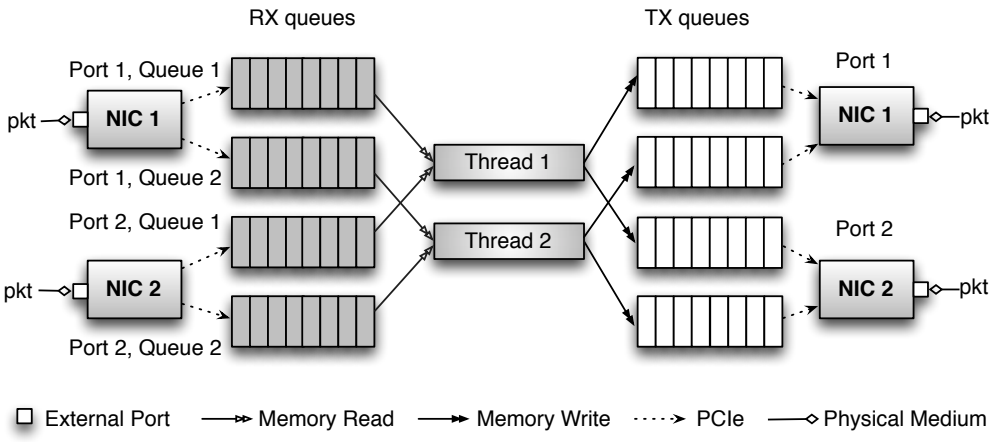


Figure 2: Packet processing pipeline of CuckooSwitch.

One of the biggest challenges in this design is to effectively mask the high latency to memory, which is roughly 100 nanoseconds. The maximum packet forwarding rate of our hardware is roughly 92 million packets per second, as shown in Section 3.7. At this rate, a software switch has on average only 10.8 nanoseconds to process each packet. As a result, it is obviously necessary to both exploit parallelism and to have a deeper pipeline of packets being forwarded. Our techniques therefore aggressively take advantage of multicore, packet batching, and memory prefetching.

3.1 Overview: Packet Processing Pipeline

CuckooSwitch's packet processing pipeline has three stages. In the first stage, NICs receive packets from the network and push them into RX queues using Direct Memory Access (DMA). To spread the load of packet processing evenly across all CPU cores, the NICs use Receive Side Scaling (RSS). RSS is a hardware feature that directs packets to different RX queues based on a hash of selected fields in the packet headers; this ensures that all packets within a flow are handled by the same queue to prevent reordering. After incoming packets are placed into the corresponding RX queues, a set of user-space worker threads (each usually bound to a CPU core) reads the packets from their assigned RX queues (typically in a round-robin manner), and extracts the destination MAC address (DMAC) from each packet. Next, DMACs are looked up in the concurrent multi-reader cuckoo hash table, which returns the output port for each DMAC. Worker threads then distribute packets into the TX queues associated with the corresponding output port. In the final stage, NICs transmit the packets in TX queues. To avoid contention and the overhead of synchronization, as well as to use the inter-NUMA domain bandwidth efficiently, for each CPU core (corresponding to one worker thread), we create one RX queue for this core on each NIC in the same NUMA domain with this core. This NUMA-aware all-to-all traffic distribution pattern allows both high performance packet processing and eliminates skew.

Figure 2 illustrates a simplified configuration with two single-port NICs and two worker threads. In this setting, each port splits its incoming packets into two RX queues, one for each thread. Two worker threads grab packets from the two RX queues associated with it and perform a DMAC lookup. Then, packets are pushed by worker threads into TX queues based on the output port returned from the DMAC lookup.

For efficiency, the DPDK manages the packet queues to ensure that packets need not be copied (after the initial DMA) on the receive path. Using this setup, the only packet copy that must occur happens

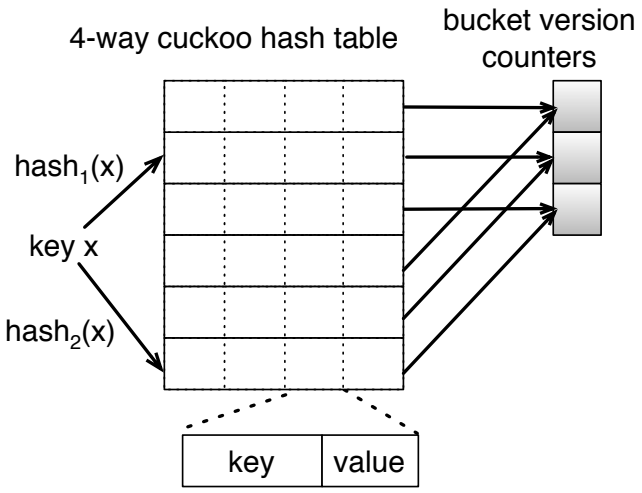


Figure 3: A 2,4 cuckoo hash table.

when copying the packet from an RX queue to an outbound TX queue.

3.2 Background: Optimistic Concurrent Cuckoo Hashing

Our FIB design is based on the recent multiple-reader, single writer "optimistic concurrent cuckoo hashing" [19]. We improve this basic mechanism by optimizing it for high-throughput packet forwarding by leveraging the strong x86 memory ordering properties and by performing lookups in batches with prefetching to substantially improve memory throughput.

At its heart, optimistic cuckoo hashing is itself an extension of cuckoo hashing [39], an open addressing hashing scheme. It achieves high memory efficiency, and ensures expected O(1) retrieval time and insertion time. As shown in Figure 3, cuckoo hashing maps each item to multiple candidate locations by hashing[2] and stores this item in one of its locations; inserting a new item may relocate existing items to their alternate candidate locations. Basic cuckoo hashing ensures 50% table space utilization, and a 4-way set-associative hash table improves the utilization to over 95% [18].

Optimistic concurrent cuckoo hashing [19] is a scheme to coordinate a single writer with multiple readers when multiple threads access a cuckoo hash table concurrently. This scheme is optimized for read-heavy workloads, as looking up an entry in the table (the common case operation) does not acquire any mutexes. To ensure that readers see consistent data with respect to the concurrent writer, each bucket is associated with a version counter by which readers can detect any change made while they were using a bucket. The writer increments the version counter whenever it modifies a bucket, either for inserting a new item to an empty entry, or for displacing an existing item; each reader snapshots and compares the version numbers before and after reading the corresponding buckets.[3] In this way, readers detect read-write conflicts from the version change. To save space, each counter is shared by multiple buckets by striping. Empirical results show that using a few thousand counters in total allows good parallelism while remaining small enough to fit comfortably in cache.

[2]We use terms "location" and "bucket" interchangeably in the thesis

[3]This description of the optimistic scheme differs from that in the cited paper. In the process of optimizing the scheme for x86 memory ordering, we discovered a potential stale-read bug in the original version. We corrected this bug by moving to the bucket-locking scheme we describe here.

Before the changes we make in this section, "2,4 optimistic concurrent cuckoo hashing" (each item is mapped to two candidate locations, each location is 4-way associative) achieved high memory efficiency (wasting only about 5% table space) and high lookup performance (each lookup requires only two *parallel* cacheline-sized reads). Moreover, it allowed multiple readers and a single writer to concurrently access the hash table, which substantially improves the performance of read-intensive workloads without sacrificing performance for write-intensive workloads. For these reasons, we believed it is a particularly appropriate starting point for a read-heavy network switch forwarding table.

3.3 x86 Optimized Hash Table

Our first contribution is an algorithmic optimization in the implementation of optimistic concurrent cuckoo hashing that eliminates the need for memory barriers on the DMAC lookup path. As we show in the evaluation, this optimization increases hash table lookup throughput by over a factor of two compared to the prior work upon which we build.

The x86 architecture has a surprisingly strong coherence model for multi-core and multi-processor access to memory. Most relevant to our work is that *a sequence of memory writes at one core are guaranteed to appear in the same order at all remote CPUs*. For example, if three writes are executed in order W1, W2, and W3, if a remote CPU node issues two reads R1 and R2, and the first read R1 observes the effect of W3, then R2 *must* observe the effects of W1, W2, and W3. This behavior is obtained without using memory barriers or locks, but does require the compiler to issue the writes in the same order that the programmer wants using *compiler reordering barriers*. Further details about the x86 memory model can be found in Section 8.2.2 of the Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A [29]. Here, we describe how we take advantage of this stronger memory model in order to substantially speed up optimistic concurrent cuckoo hashing.

The original optimistic concurrent cuckoo hashing work proposed an optimistic locking scheme to implement lightweight atomic displacement and support a *single writer*. Displacing keys from one to the other candidate bucket is required when we insert new keys using cuckoo hashing. In that optimistic scheme, each bucket is associated with a version counter (with lock striping [26] for higher space efficiency, so each counter is shared among multiple buckets by hashing). Before displacing a key between two different buckets, the single writer uses two *atomic increase* instructions to increase two associated counters by 1 respectively, indicating to other readers an on-going update of these two buckets. After the key is moved to the new location, these two counters are again increased by 1 using *atomic increase* instructions to indicate the completion. On the reader side, before reading the two buckets for a given key, a reader snapshots the version counters of these two buckets using *atomic read* instructions. If either of them is odd, there must be a concurrent writer touching the same bucket (or other buckets sharing the same counter), so it should wait and retry. Otherwise, it continues reading the two buckets. After finishing reading, it again snapshots the two counters using *atomic read* instructions. If either counter has a new version, the writer may have modified the corresponding bucket, and the reader should retry. Algorithm 1 shows the pseudo-code for the process.

The original implementation of concurrent multi-reader cuckoo hashing [3] implemented atomic read/increase using the `__sync_add_and_fetch` GCC builtin. On x86, this builtin compiles to an expensive (approximately 100 cycles) atomic instruction. This instruction acts as both a compiler reordering barrier and a hardware memory barrier, and ensures that any subsequent atomic read at any other core *will* observe the update [29].

These expensive guarantees are, in fact, stronger than needed for implementing an Ethernet switch FIB. On a *general* platform, but not x86, these atomic instructions are necessary for correctness. On a

Algorithm 1: Original lookup and key displacement.

```
OriginalLookup(key)
// lookup key in the hash table, return value
begin
    b1, b2 ← key's candidate buckets
    while true do
        v1, v2 ← b1, b2's version (by atomic read)
        full CPU memory barrier (runtime)
        if v1 or v2 is odd then continue

        if key found in b1 or b2 then
            read value of key from b1 or b2
        else
            set value to NotFound
        full CPU memory barrier (runtime)
        v1', v2' ← b1, b2's version (by atomic read)
        if v1 != v1' or v2 != v2' then continue

        return value

OriginalDisplace(key, b1, b2)
// move key from bucket b1 to bucket b2
// displace operations are serialized (in the single writer)
begin
    incr b1 and b2's version (by atomic incr)
    full CPU memory barrier (runtime)
    remove key from b1
    write key to b2
    full CPU memory barrier (runtime)
    incr b1 and b2's version (by atomic incr)
```

processor allowing stores to be reordered with stores, when a reader first snapshots a version counter, it could see the new version of the counter, but then read the old version of the data (because the stores were reordered). It would then read the version counter again, again seeing the new version, and incorrectly concluding that it must have seen a consistent version of the data.

The stronger, causally consistent x86 memory model does not allow reordering of stores relative to each other, nor does it allow reads to be reordered relative to other reads. Therefore, when a reader thread reads the version counters, the data, and the version counters again, if it observes the same version number in both reads, then there could not have been a write to the data. At the level we have discussed it, a remote reader may not observe the most recent update to the FIB, but will be guaranteed to observe some correct value from the past. In order to achieve this, it is simply necessary to ensure that the *compiler* does not reorder the store and read instructions using a compiler reordering barrier[4] between

[4]In GCC, this can be accomplished using `__asm__ __volatile__("" ::: "memory")`

Algorithm 2: Optimized lookup and key displacement. Requires total store order memory model.

```
OptimizedLookup(key)
// lookup key in the hash table, return values
begin
    b1, b2 ← key's candidate buckets
    while true do
        v1, v2 ← b1, b2's version (by normal read)
        compiler reordering barrier
        if v1 or v2 is odd then continue

        if key found in b1 or b2 then
            read value of key from b1 or b2
        else
            set value to NotFound
        compiler reordering barrier
        v1', v2' ← b1, b2's version (by normal read)
        if v1 != v1' or v2 != v2' then continue

        return value

OptimizedDisplace(key, b1, b2)
// move key from bucket b1 to bucket b2
// displace operations are serialized (in the single writer)
begin
    incr b1 and b2's version (by normal add)
    compiler reordering barrier
    remove key from b1
    write key to b2
    compiler reordering barrier
    incr b1 and b2's version (by normal add)
```

the version counter reads, data read, and subsequent version counter reads.[5] Algorithm 2 shows the optimized pseudo-code for lookup and key displacement.

Finally, while the causal ordering does not guarantee freshness at remote nodes, a full hardware memory barrier after inserting an updated value into the FIB *will* do so. This is accomplished in our system by having the writer thread obtain a pthread mutex surrounding the entire insertion process (not shown), which automatically inserts a full CPU memory barrier that force any writes to become visible at other cores and nodes in the system via the cache coherence protocol.

[5]It is also necessary to mark access to these fields `volatile` so that the compiler will not optimize them into local registers. Doing so does not harm performance much, because the second reads will still be satisfied out of L1 cache.

3.4 Batched Hash Table Lookups

In the original concurrent multi-reader cuckoo hashing, each reader thread issued only one lookup at a time. This single-lookup design suffers from low memory bandwidth utilization, because each lookup only requires two memory fetches while a modern CPU can have multiple memory loads in flight at a time (and, indeed, requires such in order to use its full memory bandwidth). In consequence, the lookup performance of the cuckoo hash table is severely restricted by the memory access latency. As we will show in Section 3.7, when the size of the hash table cannot fit in the fast CPU cache (SRAM), the performance drops dramatically.

However, this design is unnecessarily general in the context of a high-performance packet switch: The packet I/O engine we used already must take a batch-centric approach to amortize the cost of function calls to send/receive packets, to avoid unnecessary operations such as RX/TX queue index management, and to limit the number of bus transactions and memory copies. As a result, by the time a worker thread begins running, it already has a buffer containing between 1 and 16 packets to operate upon. Based on this fact, we therefore combine *all* the packets in the buffer as a *single* batch, and perform the hash table lookup for all of them at the same time. We will discuss the reason why we pick this aggressive batching strategy in Section 3.7.

Because of the synergy between batching and prefetching, we describe the resulting algorithm with both optimizations after explaining hash table prefetching next.

3.5 Hash Table Prefetching

Modern CPUs have special hardware to prefetch memory locations. Programmers and compilers can insert prefetch instructions into their programs to tell the CPU to prefetch a memory location into a given level of cache. In traditional chaining-based hash tables, prefetching is difficult (or impossible) because traversing a chain requires sequential dependent memory dereferences. The original concurrent multi-reader cuckoo hash table also saw little benefit from prefetching because each lookup query reads only two memory locations back to back; prefetching the second location when reading the first allows limited "look ahead."

With batching, however, prefetching plays an important role. Each batch of lookup requests touches multiple memory locations—two cacheline-sized buckets for each packet in the batch—so intelligently prefetching these memory locations provides substantial benefit. Algorithm 3 illustrates how we apply prefetching along with batched hash table lookups.

For each lookup query, we prefetch *one* of its two candidate buckets as soon as we finish hash computation; the other bucket is prefetched only if the corresponding key is not found in the first bucket. We refer to this strategy as "two-round prefetching." This strategy exploits several features and tradeoffs in the CPU execution pipeline. First, because modern CPUs have different execution units for arithmetic operations and memory loads/stores, carefully interleaving the hash computation with memory prefetching uses all of the execution units in the CPU. Second, the alternative bucket of each key is not prefetched at the very beginning to save CPU's load buffer[6]. Because the chance that reading the second bucket is useful is only 50%, our strategy better use the load buffer and allow the CPU to do more *useful* work compared to prefetching both buckets immediately after the hash computation.

To summarize, our lookup algorithm ensures that there are several memory loads in flight before blocking to wait on the results. It also ensures that all of the lookups will be processed with only two

[6] The interaction with the L1 Dcache load buffers for load operations. When the load buffer is full, the micro-operations flow from the front-end of CPU will be blocked until there is enough space [7].

Algorithm 3: Batched hash table lookups with prefetching.

```
BatchedLookup(keys[1..n])
// lookup a batch of n keys in the hash table, return their values
begin
    for i ← 1 to n do
        b1[i], b2[i] ← keys[i]'s candidate buckets
        prefetch b1[i]
    while true do
        // snapshot versions for all buckets
        for i ← 1 to n do
            v1[i], v2[i] ← b1[i], b2[i]'s version
        compiler reordering barrier
        if ∃i, v1[i] or v2[i] is odd then continue

        for i ← 1 to n do
            if keys[i] found in b1[i] then
                read values[i] of keys[i] from b1[i]
            else
                prefetch b2[i]
        for i ← 1 to n do
            if keys[i] not found b1[i] then
                if keys[i] found in b2[i] then
                    read values[i] of keys[i] from b2[i]
                else
                    set values[i] to NotFound
        compiler reordering barrier
        for i ← 1 to n do
            v'1[i], v'2[i] ← b1[i], b2[i]'s version
        if ∃i, v1[i] != v'1[i] or v2[i] != v'2[i] then continue

        return values[1..n]
```

rounds of memory reads, capping the maximum processing latency that the batch will experience.

The algorithm also explains the synergy between batching and prefetching: Large batch sizes make it beneficial to prefetch many locations at once. Without batching, only the alternate hash location can be prefetched. Thus, one packet is processed per memory-access latency, at a cost of two cache-lines retrieved. With batching, using our two-round prefetching strategy to reduce the overall memory bandwidth use, we can process n packets, when n is appropriately large, in two memory-access latencies with only 1.5 cache-line retrievals per packet on average.

Our evaluation in Section 3.7 shows that the combination of batching and prefetching provides a further 2× increase in local hash table lookup throughput, and a roughly 30% increase in the end-to-end packet forwarding throughput. The performance benefit increases when the table is large.

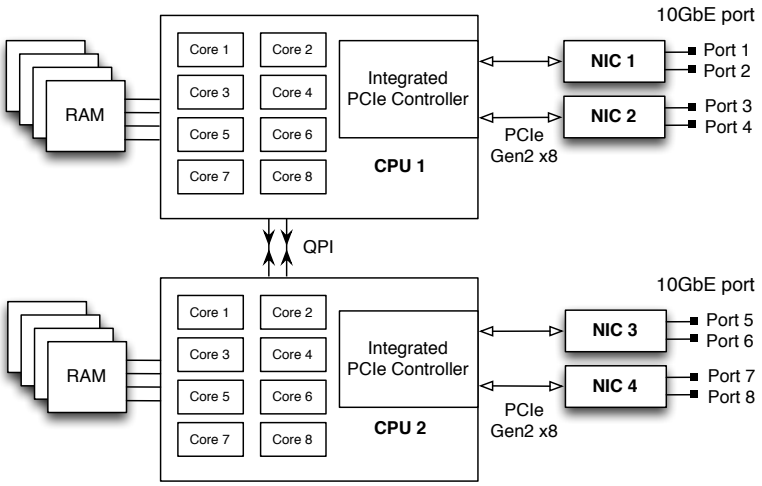


Figure 4: Topology of the evaluation platform.

3.6 Reducing TLB Misses with Huge Pages

The standard unit of page allocation provided by CPUs is 4 KiB. For programs that allocate a large amount of memory, this relatively small size means a large number of page table entries. These page table entries, which translate from virtual to physical addresses, are cached in the CPU's Translation Lookaside Buffer (TLB) to improve performance. The size of the TLB is limited, however, and TLB misses noticeably slow the memory access time. Modern CPUs offer a solution called *Huge Page Table* support, where the programmer can allocate much larger pages (e.g., 2 MiB), so that the same amount of allocated memory takes far fewer pages and thus fewer TLB entries. The CPUs in our experimental platform have 64 DLTB (Data Translation Lookaside Buffer) entries for 4 KiB pages with 32 DLTB entries for 2 MiB pages [7]. The factor of 256$\times$ in the amount of memory that can be translated by TLB greatly reduces the TLB misses and improves the switch performance. As demonstrated in the next section, using huge pages improves packet forwarding throughput by roughly 1 million more packets/second forwarded without batching, and about 3 million more packets/second forwarded when used in conjunction with batching.

3.7 Evaluation

We present how the proposed optimizations contribute to the performance of both the hash table alone (local) and of the full system forwarding packets over the network. For more detailed results and analysis, please refer to [46].

Throughout the evaluation, we use SI prefixes (e.g., K, M, G) and IEC/NIST prefixes (e.g., Ki, Mi, Gi) to differentiate between powers of 10 and powers of 2.

3.7.1 Evaluation Setup

Platform Specification Figure 4 shows the hardware topology of our evaluation platform. CuckooSwitch runs on a server with two Intel Xeon E5-2680 CPUs connected by two Intel Quickpath Interconnect (QPI) links running at 8 GT/s. Each CPU has 8 cores and an integrated PCIe I/O subsystem that provides PCIe communication directly between the CPU and devices. The server has four dual-port

CPU	2 × Intel Xeon E5-2680 @ 2.7 GHz
# cores	2 × 8 (Hyper-Threading disabled)
Cache	2 × 20 MiB L3-cache
DRAM	2 × 32 GiB DDR3 SDRAM
NIC	4 × Intel 82599-based dual-port 10GbE

Table 1: Components of the platform.

10GbE cards, for a total bandwidth of 80 Gbps (10 Gbps × 2 × 4). Table 1 lists the model and the quantity of each component in our platform. The server runs 64-bit Ubuntu 12.04 LTS.

Traffic Generator We configured two servers to generate traffic in our test environment. Each server has two 6-core Xeon L5640 CPUs and two dual-port 10GbE NICs. These two machines connect directly to our evaluation platform (not through a switch). The traffic generators can saturate all eight ports (the full 80 Gbps) with 14.88 million packets per second (Mpps) each port, using minimum-sized 64 byte packets. This is the peak packet rate achievable on 10GbE [42].

3.7.2 Hash Table Micro-benchmark

We first evaluate the stand-alone performance of *the hash table itself.* These experiments do not involve packet forwarding, just synthetically generated, uniform random successful lookups in the hash table. We begin by examining (1) how each optimization aforementioned improves the hash table performance; and (2) how the optimized table compares with other popular hash table implementations.

Factor Analysis of Lookup Performance We first measure the incremental performance improvement from each individual optimization on top of basic concurrent cuckoo hashing.

- **baseline: concurrent cuckoo** is the basic concurrent multi-reader cuckoo hash table, serving as the baseline.
- **+hugepage** enables 2 MiB x86 huge page support in Linux to reduce TLB misses.
- **+memorder** replaces the previous atomic `__sync_add_and_fetch` instruction with x86-specific memory operations.
- **+batching** groups hash table lookups into small batches and issues memory fetches from all the lookups at the same time in order to better use memory bandwidth. Here, we assume a batch size of 14 (a value picked empirically based upon the performance results).
- **+prefetch** is used along with **batching** to prefetch memory locations needed by the lookup queries in the same batch.

Figure 5 shows the result of using 16 threads. The x-axis represents the number of entries in the hash table, while the y-axis represents the total lookup throughput, measured in Million Operations Per Second (Mops). In general, combining all optimizations improves performance by approximately 5.6× over the original concurrent cuckoo hashing. "hugepage" improves performance only slightly, while "x86 memory ordering" boosts the performance by more than 2×. By reducing serialized memory accesses, the optimized cuckoo hash table better uses the available memory bandwidth. Without prefetching,

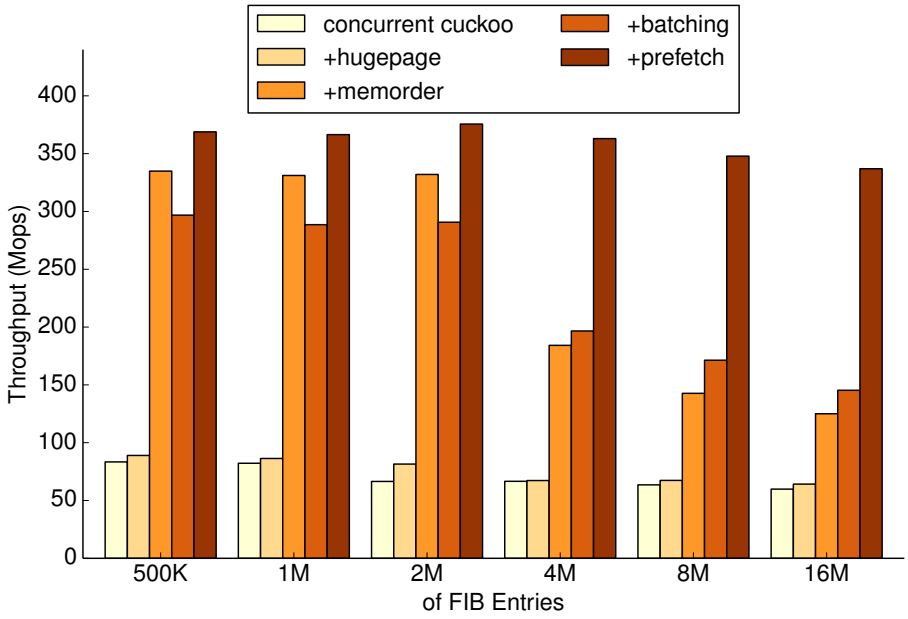


Figure 5: Contribution of optimizations to the hash table performance. Optimizations are cumulative.

"batched lookup" improves the performance by less than 20% when the table is large, and has even worse performance when the table is small enough to fit in the L3 cache of CPU. However, when combined with our "two-round prefetching", performance increases substantially, especially when the table is large. We explain this phenomenon in the next paragraph.

Batch Size As we discussed aboe, the batch size affects the performance of the optimized cuckoo hash table. Because the size of the largest batch we receive from the packet I/O engine is 16 packets, we therefore evaluate the throughput achieved by our optimized cuckoo hash table with no batching (batches of one lookup) up through batches of 16 lookups. Our motivation is to gain insight on how to perform forwarding table lookups for batches with different sizes.

Figure 6 shows five representative batch sizes. As illustrated in the figure, up to batches of 4 lookups, there is a significant performance improvement for both small and large tables. When the batch size grows to 5 or more, the lookup performance of our optimized cuckoo hash table *slightly* drops if the table is small, and the performance increases *greatly* for tables that is too large to fit in CPU's SRAM-based L3 cache. That is to say, larger batches become more important as the table size grows. This makes intuitive sense: Small tables fit entirely in the CPU's SRAM-based L3 cache, and so fewer concurrent requests are needed to mask the latency of hash table lookups. When the number of entries increases from 2 million (8 MiB of hash table) to 4 million (16 MiB), performance of small batch sizes decreases significantly because the hash table starts to exceed the size of cache. Past 4 million entries, doubling the number of hash table entries causes a small, roughly linear decrease in throughput as the cache hit rate continues to decrease and the probability of TLB misses begins to increase, even with huge pages enabled.

On the other hand, larger batches of lookups ensure relatively stable performance as the number of FIB entries increases. In other words, batching and prefetching successfully masks the effect of high DRAM access latency. This is exactly what CuckooSwitch desires, because when the number of FIB entries is small, the lookup performance of the hash table is already more than enough to serve

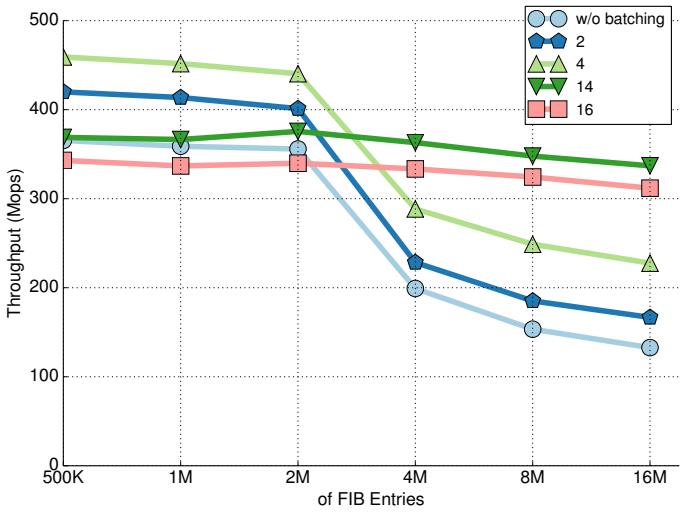


Figure 6: Lookup throughput of hash table vs. batch size.

the network traffic; however, when the number of FIB entries is large, improved lookup performance increases forwarding throughput.

The figure also shows that the overall best batch size is 14, which is why we pick this batch size in the previous experiment. Moreover, even though having a batch size of 14 gives us the highest performance, other batch sizes are not far below. For example, batching 16 lookups is only 6% slower than batching 14. Based on this fact, we adopt a batching strategy, which we called *dynamic batching*, to perform hash table lookups.

To be more specific, whenever a worker thread receives packets from the I/O engine, it will get between 1 and 16 packets. Instead of having a fixed batch size and grouping the received packets with this size, we instead combine all of the packets as a *single batch* and perform the hash table lookups at once. Two factors let us discard fixed batch sizes. First, as explained above, larger batches usually perform better when the number of entries in the table is large. Second, having a fixed batch size will have to deal with extra packets if the number of received packets is not a multiple of the batch size, and these packets cannot benefit from batching.

Summary: Our optimized concurrent cuckoo hashing provides nearly 350 million small key/value lookups per second without sacrificing thread-safety. When used as the forwarding table in a software switch, dynamic batching should be used in hash table lookup.

3.7.3 Full System Forwarding Evaluation

We evaluate how each optimization improves the full system packet forwarding throughput. In these experiments, we measure the throughput using worst-case 64 byte packets (minimum sized Ethernet packets) unless otherwise noted.

Factor Analysis of Full System Throughput These experiments mirror those in the hash table micro-benchmark: starting from basic concurrent multi-reader cuckoo hashing and adding the same set of optimizations cumulatively. As discussed in the previous subsection, we use dynamic batching. Figure 7 shows the results across different FIB sizes. In total, the hash table optimizations boost throughput by

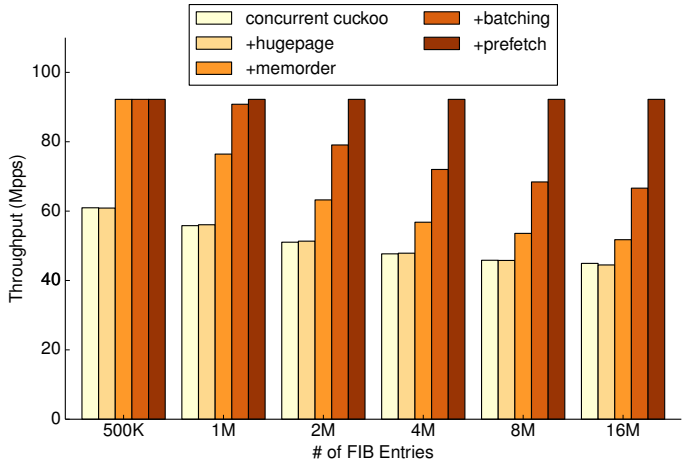


Figure 7: Full system packet forwarding throughput factor analysis. Packet size = 64B.

roughly 2×. CuckooSwitch achieves maximum 64 byte packet throughput with even *one billion* FIB entries.

Summary By using our optimized concurrent cuckoo hashing and dynamic batching policy, CuckooSwitch can saturate the maximum number of packets achievable by the underlying hardware, even with one billion FIB entries in the forwarding table.

4 Scaling Up Clustered Network Appliances with ScaleBricks

RouteBricks [17] proposed an architecture, which parallelizes the functionality across multiple servers, to build scalable software routers. In RouteBricks, adding another node to a cluster increases only the total throughput and number of ports, but does not increase the total number of keys that the cluster can support.

Meanwhile, many clustered applications require *predetermined* partitioning of a flat key space among a cluster of servers. When a packet enters the cluster, the *ingress* node will direct the packet to its *handling* node. The handling node maintains state that is used to process the packet, such as the packet's destination address or the flow to which it belongs. Examples include carrier-grade NATs, per-flow switching in software-defined networks (SDNs), and, as we will discuss in Section 4.1, the cellular network-to-Internet gateway [8] in the core network of Long-Term Evolution (LTE).

In this section, we explore a less-examined aspect of scalability for such clustered applications: can we create a design in which the FIB that maps flat keys to their corresponding handling nodes "scales out" alongside throughput and port count as one adds more nodes to the cluster? And, critically, can we do so *without* increasing the amount of traffic that crosses the internal switching fabric? Our proposed architecture – ScaleBricks, allows the FIB to continue to scale through 8, 16, or even 32 nodes, increasing the FIB capacity by up to 5.7x.

To be more specific, we focus on three properties of cluster scaling:

Throughput Scaling. The aggregate throughput of the cluster scales with the number of cluster servers;

FIB Scaling. The total size of the forwarding table (the number of supported keys) scales with the number of servers; and

Update Scaling. The maximum update rate of the FIB scales with the number of servers.

In all cases, we do not want to scale at the expense of incurring high latency or higher switching fabric cost. As we discuss further in Section 4.2, existing designs do not satisfy these goals. For example, the typical approach of duplicating the FIB on all nodes fails to achieve FIB scaling; a distributed hash design such as used in SEATTLE [31] requires multiple hops across the fabric.

4.1 Driving Application: Cellular Network-To-Internet Gateway

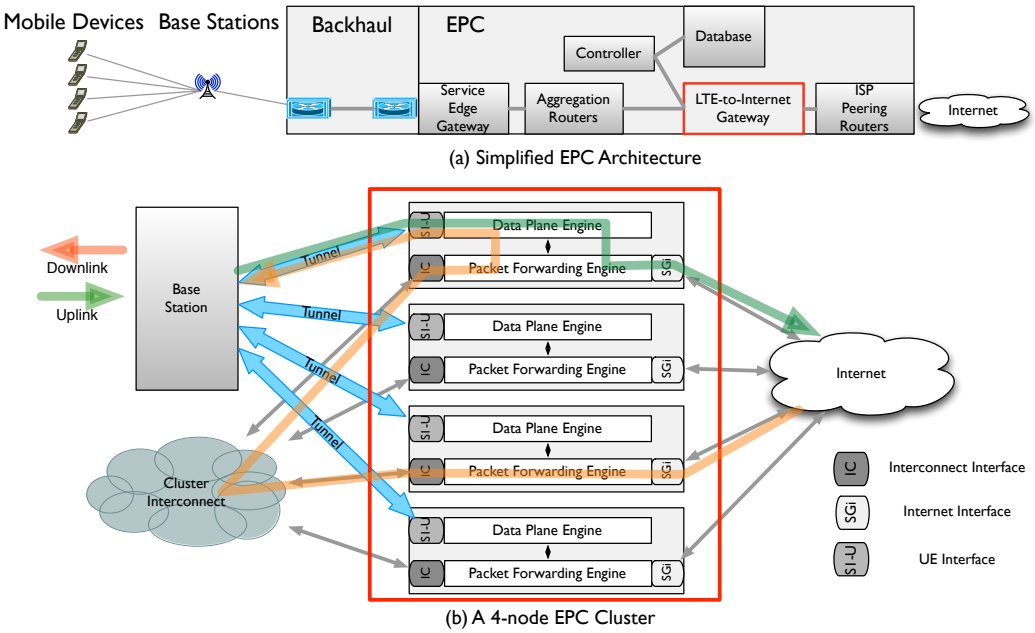


Figure 8: (a) Simplified Evolved Packet Core (EPC) architecture and (b) a 4-node EPC cluster

To motivate ScaleBricks, we begin by introducing a concrete application that can benefit from ScaleBricks: the Internet gateway used in LTE cellular networks. The central processing component in LTE is termed the "Evolved Packet Core," or EPC [8]; Figure 8a shows a simplified view of the EPC architecture. The following is a high-level description of how it services mobile devices ("mobiles" from here on); more details are described in the Internet draft on Service Function Chaining Use Cases in Mobile Networks [24].

- When an application running on the mobile initiates a connection, the *controller* assigns the new connection a tunnel, called the GTP-U tunnel, and a unique Tunnel End Point Identifier (TEID).[7]
- Upstream traffic (from the mobile to the Internet), sends packets through several middleboxes to the LTE-to-Internet gateway (the red box in the figures). After performing administrative functions such as charging and access control, the gateway decapsulates packets from the GTP-U tunnel, updates the state associated with the flow, and sends them to ISP peering routers, which connect to the Internet.

[7]For clarity, we have used common terms for the components of the network. Readers familiar with LTE terminology will recognize that our "mobile device" is a "UE"; the base station is an "eNodeB"; and the tunnel from the UE to the eNodeB is a "GTP-U" tunnel.

- Downstream traffic follows a reverse path across the elements. The LTE-to-Internet gateway processes and re-encapsulates packets into the tunnels based on the flow's TEID. The packets reach the correct base station, which transmits them to the mobile.

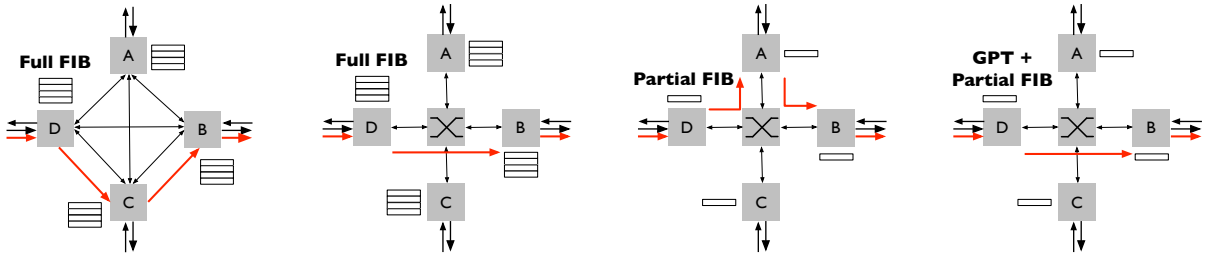
In this section, we focus our improvements on a commercially-available software EPC stack from Connectem [13]. This system runs on commodity hardware and aims to provide a cost-advantaged replacement for proprietary hardware implementations of the EPC. It provides throughput scalability by clustering multiple nodes: Figure 8b shows a 4-node EPC cluster. When a new connection is established, the controller assigns a TEID to the flow and assigns that flow to one node in the cluster (its *handling node*). This assignment is based on several LTE-specific constraints, such as geometric proximity (mobile devices from the same region are assigned to the same node), which *prevents* us from modifying it (e.g., forcing hash-based assignment), thereby requiring *deterministic* partitioning. It then inserts a mapping from the 5-tuple flow identifier to the (handling node, TEID) pair into the cluster forwarding table. Upstream packets from this flow are directed to the handling node by the aggregation router. Downstream packets, however, could be received by *any* node in the cluster because of limitations in the hardware routers that are outside of our control. For example, the deployment of an equal-cost multi-path routing (ECMP) strategy may cause the scenario described above because all nodes in the cluster will have the same distance to the destination. Because the cluster maintains the state associated with each flow at its *handling node*, when the ingress node receives a downstream packet, it must look up the handling node and TEID in its forwarding table and forward the packet appropriately. The handling node then processes the packet and sends it back over the tunnel to the mobile.

Our goal is to demonstrate the effectiveness of ScaleBricks by using it to improve the performance and scalability of this software-based EPC stack. We chose this application both because it is commercially important (hardware EPC implementations can cost hundreds of thousands to millions of dollars), is widely used, and represents an excellent target for scaling using ScaleBricks because of its need to pin flows to a specific handling node combined with the requirement of maintaining as little states at each node as possible (which makes keeping a full per-flow forwarding table at each node a less viable option). ScaleBricks achieves these goals without increasing the inter-cluster latency. Compared with alternative designs, this latency reduction could be important in several scenarios, including communication between mobile devices and content delivery networks, as well as other services deployed at edge servers. In this work, we change only the "Packet Forwarding Engine" of the EPC; this is the component that is responsible for directing packets to their appropriate handling node. We leave unchanged the "Data Plane Engine" that performs the core EPC functions.

4.2 Design Overview

There are several design choices we made for ScaleBricks. Here, we explain those choices and compare them to representative alternative designs to illustrate our decisions. Throughout the section, we use the following terms to describe the cluster architecture:

- *Ingress Node*: the node where a packet enters the cluster.
- *Handling Node*: the node where a packet is processed within the cluster.
- *Indirect Node*: an intermediate node touched by a packet, not including its ingress and handling node.



(a) RouteBricks (b) Full Duplication (c) Hash Partitioning (d) ScaleBricks

Figure 9: Packet forwarding in different FIB architectures

- *Lookup Node*: If node X stores the forwarding entry associated with a packet P, X is P's lookup node. A packet may have no lookup nodes if the packet has an unknown key, or more than one lookup node if the FIB entry has been replicated to more than one node.

4.2.1 Cluster Architecture

Two topologies are classically used for building cluster-based network functions. The first connects cluster servers directly to each other, as exemplified by RouteBricks [17]. In such systems, the servers are connected in a full mesh or a butterfly-like topology, as shown in Figure 9a. On top of this topology, load-balancing routing algorithms—e.g., Valiant Load Balancing (VLB) [43]—guarantee 100% throughput and fairness without centralized scheduling.

This solution has the advantage that the total bandwidth of internal links used to construct the full mesh or the butterfly needs to be only $2\times$ the total external bandwidth; furthermore, these links are fully utilized. The disadvantage of VLB, however, is that the ingress node must forward each incoming packet to an intermediate indirect node before it reaches the handling node. This extra step ensures efficient use of the aggregate internal bandwidth. Unfortunately, in most cases, each packet must be processed by three nodes (two hops). This increases packet processing latency, server load, and required internal link capacity.

The second class of topologies uses a hardware switch to connect the cluster nodes (Figures 9b–9d). This topology offers two attractive properties. First, it allows full utilization of internal links without increasing the total internal traffic. To support R Gbps of external bandwidth, a node needs only R Gbps of aggregate internal bandwidth, instead of the $2R$ required by VLB. Second, without an indirect node, packet latency depends on the hardware switch's latency instead of the indirect node. Compared to VLB, a switch-based topology could reduce latency by 33%.

Interestingly, RouteBricks intentionally rejected this design option. The authors argued that the cost of four 10 Gbps switch ports was equal to the cost of one server, and hence a switched cluster was more expensive than a server-based cluster. Today, however, the economics of this argument have changed. New vendors such as Mellanox offer much cheaper hardware switches. For example, a Mellanox 36 port 40 GbE switch costs roughly $13,000, or ~$9 / Gbps. This is 80% lower than the number reported in the RouteBricks section. More importantly, hardware switches are particularly suitable for building interconnections for the cluster nodes. Their strengths—high bandwidth, low latency, simple and clear topology—are well-suited to our requirements; and their weakness—limited FIB size—is essentially irrelevant with our approach.

ScaleBricks thus connects servers using a switch. This topology reduces the internal bandwidth requirement and provides the opportunity to reduce the packet processing latency. However, this choice

also makes the design of a scalable forwarding architecture challenging, as explained next.

4.2.2 FIB Architecture

Given a switch-based cluster topology, the next question is what forwarding architecture to use. In the simplest design, each node in the cluster stores a full copy of the entire forwarding table (Figure 9b). When a packet arrives at its ingress node, the ingress node performs a FIB lookup to identify the handling node, and then forwards the packet directly to that node. (The ingress node thus also serves as the lookup node.)

This simple architecture requires only one hop, unlike VLB. Unfortunately, the memory required by the globally replicated FIB increases linearly with the number of nodes in the cluster. Furthermore, every update must be applied to all nodes in the cluster, limiting the aggregate FIB update rate to that of a single server.

An alternative is a hash-partitioned design (Figure 9c). For an N-node cluster, each node stores only $1/N$ FIB entries based on the hash of the keys. The ingress node must forward arriving packets to the indirect node that has the relevant portion of the FIB; the indirect node then forwards the packet to the handling node by looking up in its slice of the FIB. This approach is nearly perfectly scalable, but reintroduces the two-hop latency and bandwidth costs of VLB.

In this section, we present a design that forwards directly from ingress to handling nodes, but uses substantially less memory than a typical fully-replicated FIB (Figure 9d). At a high level, ScaleBricks distributes the entire routing information (mapping from flat keys to their corresponding nodes and other associated values), or "RIB" for short, across the cluster using a hash-partitioned design. From the RIB, it generates two structures. First, an extremely compact global lookup table called the "GPT" or Global Partition Table, that is used to direct packets to the handling node. The GPT is much smaller than a conventional, fully-replicated FIB. Second, the RIB is used to generate FIB entries that are stored only at the relevant handling nodes, *not* globally. In the LTE-to-Internet gateway example, GPT stores the mapping from flow ID to handling node, while FIB stores the mapping from flow ID to TEID.

The GPT relies upon two important attributes of switch-based "middlebox" clusters: First, the total number of nodes is typically modest—likely under 16 or 32. Second, they can handle one-sided errors in packet forwarding. Packets that match a FIB entry must be forwarded to the correct handling node, but it is acceptable to forward packets with no corresponding entry to a "wrong" (or random) handling node, and have the packet be discarded there. This property is true in the switch-based design: The internal bandwidth must be sufficient to handle traffic in which all packets are valid, and so invalid packets can be safely forwarded across the interconnect.

The Full FIB entries that map keys to handling nodes (along with, potentially, some additional information) are partitioned so that each handling node stores the FIB entries that point to it. If the handling node receives, via its internal links, a packet with a key that does not exist in its FIB, the input processing code will report that the key is missing (which can be handled in an application-specific way). The handling node FIB is based upon space-efficient, high-performance hash tables for read-intensive workloads as described in Section 3. We extend it to handle configurable-sized values with minimal performance impact.

The Global Partition Table is replicated to every ingress node. This table maps keys to a lookup/handling node. Because the GPT is fully replicated, it must be compact to ensure scalability; otherwise, it would be no better than replicating the FIB to all nodes. For efficiency, the GPT is based upon a new data

structure with one-sided error. Observing that the range of possible values (i.e., the number of nodes) in the GPT is small, using a general-purpose lookup table mapping arbitrary keys to arbitrary values is unnecessary. Instead, the GPT's mapping can be more efficiently viewed as *set separation*: dividing a set of keys into a small number of disjoint subsets. In this section, we extend prior work by Fan et al. [20] to create a fully-functional set separation data structure called SetSep, and use it at the core of the GPT. SetSep maps each key to a small set of output values—the lookup/handling node identifiers—without explicitly storing the keys at all. The tradeoff is that unknown destinations map to incorrect values; in other words, the SetSep cannot return a "not found" answer. This behavior does not harm the correctness of ScaleBricks, because the lookup node will eventually reject the unknown key. The advantage, though, is that lookup is very fast and each entry in the SetSep requires only 2–4 *bits* per entry for 4-16 servers. Section 4.3 describes the SetSep data structure in detail.

RIB Updates are sent to the appropriate RIB partition node based upon the hash of the key involved. This node generates new or updated FIB and GPT entries. It then sends the updated FIB entry to the appropriate handling node, and sends a delta-update for the GPT to all nodes in the cluster. Because the SetSep data structure used for the GPT groups keys into independently-updatable sub-blocks, the RIB partitioning function depends on how those sub-blocks are partitioned. Section 4.4.3 provides further details about RIB partitioning.

4.3 Set Separation

The GPT is stored on every node and is consulted once for each packet that enters the cluster. It must therefore be both space efficient and extremely fast. To achieve these goals, and allow ScaleBricks to scale well as a result, we extended and optimized our previous work on set separation data structures [20] to provide memory-speed lookups on billions of entries, while requiring only a few bits per entry. As discussed in the previous subsection, the design of SetSep leverages three properties of ScaleBricks:

- The GPT returns integer values between 0 and $N-1$, where N is the number of servers in the cluster.
- The GPT may return an arbitrary answer when queried for the handling node of a packet with an unknown destination key (e.g., an invalid packet). Such packets will be subsequently dropped or dealt with when the handling node performs a full FIB lookup.
- GPT lookups are frequent, but updates much less so. Therefore, a structure with fast lookup but relatively expensive updates is a reasonable tradeoff.

At a high level, the basic idea in SetSep is to use brute force computation to find a function that maps each input key to the correct output (the "set", here the cluster node index). Rather than explicitly storing all keys and their associated values, SetSep stores only indices into families of hash functions that map keys to values, and thereby consumes much less space than conventional lookup tables. Finding a hash function that maps each of a large number of input keys to the correct output value is effectively impossible, so we break the problem down into smaller pieces. First, we build a high-level index structure to divide the entire input set into many small groups. Each group consists of approximately sixteen keys in our implementation. Then, for each small group, we perform a brute force search to find a hash function that produces the correct outputs for each key in the group. The rest of this section carefully presents these two pieces, in reverse order.

4.3.1 Binary Separation of Small Sets

We start by focusing on a simple set separation problem: *divide a set of n keys into two disjoint subsets when n is small*. We show how to extend this binary separation scheme to handle more subsets in Section 4.4.1.

Searching for SetSep To separate a set of n key-value pairs (x_j, y_j), where x_j is the key and y_j is either "0" or "1", we find a hash function f that satisfies $f(x_j) = y_j$ for $j \in [0, n)$. Such a hash function is discovered by iterating over a hash function family $\{H_i(x)\}$ parameterized by i, so $H_i(x)$ is the i-th hash function in this family. Starting from $i = 1$, for each key-value pair (x_j, y_j), we verify if $H_i(x_j) = y_j$ is achieved. If any key x_j fails, the current hash function H_i is rejected, and the next hash function H_{i+1} is tested on all n keys again (including these keys that passed H_i). In other words, we use brute force to find a suitable hash function. As shown later, this search can complete very rapidly for small n and an appropriate hash function family.

Once a hash function H_i that works for all n keys is found, its index parameter i is stored. We choose some maximum stopping value I, so that if no hash function succeeds for $i \leq I$, a fallback mechanism is triggered to handle this set (e.g., store the keys explicitly in a separate, small hash table).

Storing SetSep For each group, the index i of the successful hash function is stored using a suitable variable-length encoding. As shown in the next paragraph, ideally, the expected space required from this approach is near optimal (1 bit per key). In practice, however, storing a variable length integer adds some overhead, as do various algorithmic optimizations we use to speed construction. Our implementation therefore consumes about 1.5 bits per key.

Why SetSep Saves Space Let us optimistically assume our hash functions produce fully random hash values. The probability a hash function H_i maps one key to the correct binary value is $1/2$, and the probability all n keys are properly mapped is $p = (1/2)^n$. Thus, the number of tested functions (i.e., the index i stored) is a random variable with a Geometric distribution, with entropy

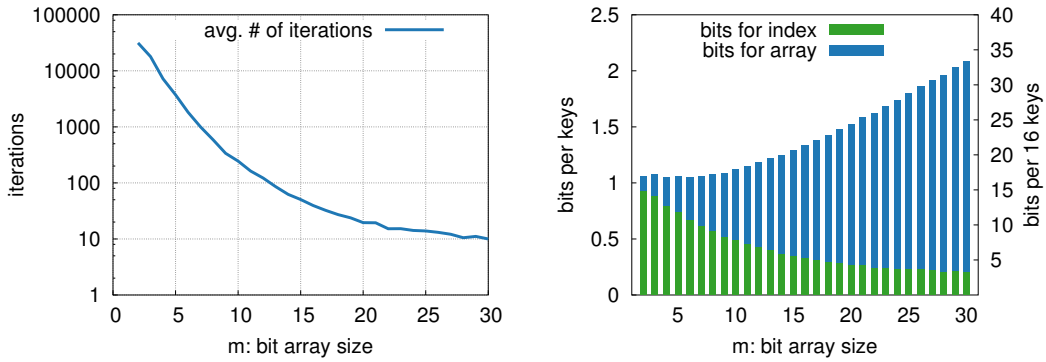
$$\frac{-(1-p)\log_2(1-p) - p\log_2 p}{p} \approx -\log_2 p = n \tag{1}$$

Eq. (1) indicates that storing a function for binary set separation of n keys requires n bits on average (or 1 bit per key), which is independent of the key size.

Insights: The space required to store SetSep approximately equals the total number of bits used by the values; the keys do not consume space. This is the source of both SetSep's strength (extreme memory efficiency) and its weakness (returning arbitrary results for keys that are not in the set).

Practically Generating the Hash Functions A simple but inefficient approach that creates the hash function family $\{H_i(x)\}$ is to concatenate the bits of i and x as the input to a strong hash function. This approach provides independence across H_i, but requires computing an expensive new hash value for each i during the iteration.

Instead of this expensive approach, we draw inspiration from theoretical results that two hash functions can sufficiently simulate additional hash functions [32]. Therefore, we first compute two approximately independent hash functions of the key, G_1 and G_2, using standard hashing methods. We then compute the remaining hash functions as linear combinations of these two. Thus, our parameterized hash function family to produce random bits is constructed by



(a) Avg. # of iters. to get one hash func.

(b) Space cost breakdown

Figure 10: Space vs. time, as the function of bit array size m

$$H_i(x) = G_1(x) + i \cdot G_2(x)$$

where $G_1(x)$ and $G_2(x)$ are both unsigned integers. In practice, only the most significant bit(s) from the summation result are used in the output, because our approach of generating parameterized hash function family will have shorter period if the least significant bits are used instead of the most significant bits.

Both hash computation and searching are fast using this mechanism: H_i can be computed directly using one multiplication and one addition. Furthermore, the hash family can be iterated using only one addition to get the value of $H_{i+1}(x)$ from the previous result of $H_i(x)$.

The hash functions described above are theoretically weak: they lack sufficient independence, and as such are more likely than "random" hash functions to fail to find a suitable mapping. Empirically, however, we observe that this approach fails only once every few billion keys. The fallback mechanism of looking the keys up in a separate, small table handles such failures.

4.4 Trading Space for Faster Construction

One problem with this basic design is the exponential growth of the number of iterations to find a hash function mapping n input items to their correct binary values. We must test and reject 2^n hash functions on average. By trading a small amount of extra space (roughly 5% compared to the achievable lower bound), SetSep optimizes the construction to be an order of magnitude faster.

Instead of generating the possible output value for x using the hash function $H_i(x)$ directly, SetSep adds an array of m bits ($m \geq 2$) and makes $H_i(x)$ map each input x to one of the m bits in the array. In other words, the output value for x is the bit stored in `bitarray[`$H_i(x)$`]` rather than $H_i(x)$. To construct the bit array, at the beginning of the iteration testing hash function $H_i(x)$, all bits in the array are marked "not taken." For each key-value pair (x_j, y_j), if $H_i(x_j)$ points to a bit that is still "not taken," we set the bit to y_j and mark it as "taken." If the bit is marked as "taken," we check if the value of the bit in the array matches y_j. If so, the current hash function is still good and can proceed to the next key. Otherwise, we reject the current hash function, switch to the next hash function H_{i+1}, re-initialize the bit array, and start testing from the first key. Intuitively, with more "buckets" for the keys to fall in, there are fewer collisions, increasing the odds of success. Thus, adding this bit array greatly improves the chance of finding a working hash function.

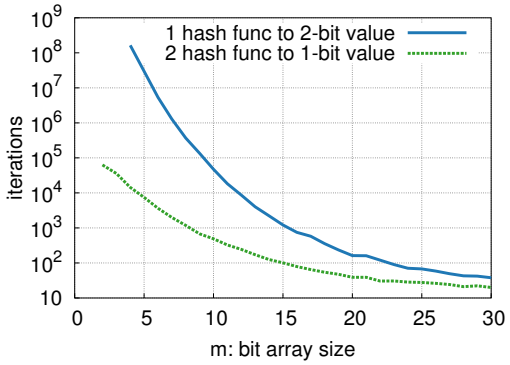


Figure 11: One hash func. vs. multiple hash func.

Space vs. Speed Storing the bit array adds m bits of storage overhead, but it speeds up the search to find a suitable hash function and correspondingly reduces the number of bits needed to store i—since each hash function has a greater probability of success, i will be smaller on average. Figure 10a shows the tradeoff between the space and construction speed for a SetSep of $n = 16$ keys while varying the bit array size (m) from 2 to 30. Increasing the size of the bit array dramatically reduces the number of iterations needed. It requires more than 10,000 hash functions on average when $m = 2$; this improves by $10\times$ when $m = 6$, and when $m \geq 12$, it needs (on average) fewer than 100 trials, i.e., it is $100\times$ faster.

Figure 10b presents analytical results for the total space (i.e., bits required to store the index i plus the m bits of the array) for this SetSep. The total space cost is almost an increasing function of m. The minimum space is 16 bits, but even when $m = 12$, the total space cost is only about 20 bits. (This is less than $16 + 12 = 28$ bits, because of the reduction in the space required to store i.)

Insights: Trading a little space efficiency (e.g., spending 20 bits for every 16 keys rather than 16 bits) improves construction speed by $100\times$.

Representing the SetSep. In light of the above result, we choose to represent the SetSep using a fixed 24-bit representation per group in our implementation, with up to 16 bits to represent the hash index and $m = 8$. This yields 1.5 bits per key on average. (We show the effect of the choice m later in the section.) Although we could use less space, this choice provides fast construction while ensuring that fewer than 1 in 1 million groups must be stored in the external table, and provides for fast, well-aligned access to the data.

4.4.1 Representing Non-Boolean Values

We have described the construction for two disjoint subsets (two possible values). For $V > 2$ different subsets, a trivial extension is to look for *one hash function* that outputs the right value in $\{1, \ldots, V\}$ for each key. However, this approach is not practical because it must try $O(V^n)$ hash functions on average. Even when $n = 16$ and $V = 4$, in the worst case, it could be 65536 times slower than $V = 2$.

We instead search for $\log_2 V$ hash functions, where the j-th hash function is responsible for generating the j-th bit of the final mapping value. As an example, assume we want to construct a mapping to a value in $\{0, 1, 2, 3\}$ from a given set of size 2. If the final mapping is ("foo", 01_2) and ("bar", 10_2), we look for two hash functions so that the first hash function maps "foo" to 0 and "bar" to 1, and the second hash function hashes "foo" to 1 and "bar" to 0. The expected total number of iterations to construct a final mapping is then $\log_2 V \cdot 2^n$, which scales linearly with the number of bits to represent a value. Figure 11

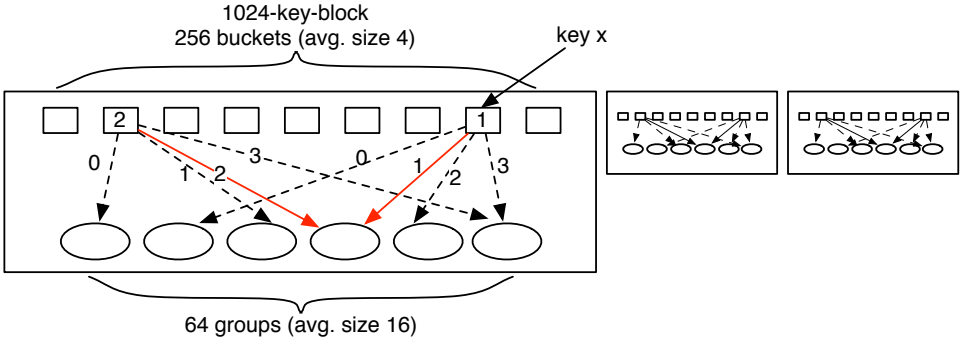


Figure 12: Illustration of two-level hashing

compares the number of iterations needed to build a separation of 4 subsets by searching for one hash function mapping to {0,1,2,3} or two hash functions mapping to {0,1} respectively. Splitting the value bits is orders of magnitude faster.

4.4.2 Scaling to Billions of Items

The basic idea of efficiently scaling SetSep to store mappings for millions or billions of keys, as noted above, is to first partition the entire set into many small groups, here of roughly 16 keys each. Then, for each group of keys, we find and store a hash function that generates the correct values using the techniques described above. Therefore, two properties are critical for the scheme that maps a key to a group:

- The mapping must ensure low variance in group size. Although a small load imbalance is acceptable, even slightly larger groups require much longer to find a suitable hash function using brute-force search because the time grows exponentially in the group size.
- The mapping should add little space. The per-group SetSep itself is only a few bits per key, and the partitioning scheme should preserve the overall space efficiency.

Conventional Solutions That Do Not Work Well To calculate the group ID of a given key, one obvious way is to compute a hash of this key modulo the total number of groups. This approach is simple to implement, and does not require storing any additional information; unfortunately, some groups will be significantly more loaded than the average group even with a strong hash function [37]. Our experiments show that when 16 million keys are partitioned into 1 million groups using even a cryptographic hash function, the most loaded group typically contains more than 40 keys vs. the average group size of 16 keys; this matches the corresponding theory. Finding hash functions via brute force for such large groups is impractical.

An alternative solution is to sort all keys and assign every n consecutive keys to one group. This approach ensures that every group has exactly sixteen keys. Unfortunately, it has several serious limitations: (1) it requires storing the full keys, or at least key fragments on the boundary of each group, as an index; (2) it requires a binary search on lookup to locate a given key's group; and (3) update is expensive.

Our Solution: Two-Level Hashing. SetSep uses a novel two-level hashing scheme that nearly uniformly distributes billions of keys across groups, at a constant storage cost of 0.5 bits per key. The first

level maps keys to buckets with a small average size—smaller than our target group size of sixteen—using simple direct hashing. These buckets will have the aforementioned huge load variance. To address this problem, at the second level, we assign buckets to groups with the aim of minimizing the maximum load on any group. The storage cost of this scheme, therefore, is the bits required to store the group choice for each bucket.

Figure 12 shows this process. Each first-level bucket has an average size of 4 keys but the variance is high: some buckets could be empty, while some may contain ten or more keys. However, across a longer range of small buckets, the average number of stored keys has less variance. For 256 buckets, there are 1024 keys on average. We therefore take consecutive blocks of 256 buckets and call them a *1024-key-block*. We then map these blocks to 64 groups of average size 16.

Within the block of 256 buckets, each bucket is mapped to one of four different "candidate" groups. We pre-assigned candidate groups for each bucket in a way that each group has the same number of associated buckets. These choices are denoted by the arrow from bucket to groups in Figure 12. All keys in the small bucket will map to one of these four candidate groups. Therefore, the only information that SetSep needs to store is the bucket-to-group mapping (a number in {0,1,2,3} indicating which candidate group was chosen).

The effectiveness of this bucket-to-group mapping is important to the performance of SetSep, since as we have explained more balanced groups make it easier to find suitable hash functions for all groups. Ideally, we would like to assign the same number of keys to each group. However, finding such an assignment corresponds to an NP-hard variant of the knapsack problem. Therefore, we use a greedy algorithm to balance keys across groups. We first sort all the buckets in descending order by size. Starting from the largest bucket, we assign each bucket to one of the candidate groups. For each bucket, we pick the candidate group with the fewest keys. If more than one group has the same least number of keys, a random group from this set is picked. We repeat this process until all the buckets have been assigned, yielding a valid assignment. In fact, we run this randomized algorithm several times per block and choose the best assignment among the runs. To lookup a key x, we first calculate the key's bucket ID by hashing. Then, given this bucket ID, we look up the stored choice number to calculate which group this key belongs to. Each bucket has 4 keys on average, and spends 2 bits to encode its choice. So on average, two-level hashing costs 0.5 bits per key, but provides much better load balance than direct hashing. When partitioning 16 million keys into 1 million groups, the most loaded group usually has 21 keys, compared to more than 40 for direct hashing.

4.4.3 Scalable Update

Allowing lookups without storing keys is the primary reason SetSep is so compact. The original construction and updates, however, require the full key/value pairs to recompute SetSep. In ScaleBricks, this information comprises the RIB, where keys are destination addresses and values are the corresponding handling nodes.

To provide scalability for the RIB size (e.g., the number of flows that the EPC can keep track of) and update rate, ScaleBricks uses a partitioned SetSep construction and update scheme. The RIB entries are partitioned using a hash of the key, so that keys in the same 1024-key-block are stored in the same node. For construction, each node computes only its portion of SetSep, and then exchanges the produced result with all the other nodes. When updating a key k, only the node responsible for k recomputes the group that k belongs to, and then broadcasts the result to other nodes. Because applying a delta-update on the other nodes requires only a memory copy (the delta is usually tens of bits), this approach allows ScaleBricks to scale the update rate with the number of nodes. To allow high-performance reads with

Construction setting			Construction throughput	Fallback ratio	Total size	Bits/ key
$x+y$ bits to store a hash function, x-bit hash function index and y-bit array						
16+8	1-bit value	1 thread	0.54 Mkeys/sec	0.00%	16.00 MB	2.00
8+16	1-bit value	1 thread	2.42 Mkeys/sec	1.15%	16.64 MB	2.08
16+16	1-bit value	1 thread	2.47 Mkeys/sec	0.00%	20.00 MB	2.50
increasing the value size						
16+8	**2-bit value**	1 thread	0.24 Mkeys/sec	0.00%	28.00 MB	3.50
16+8	**3-bit value**	1 thread	0.18 Mkeys/sec	0.00%	40.00 MB	5.00
16+8	**4-bit value**	1 thread	0.14 Mkeys/sec	0.00%	52.00 MB	6.50
using multiple threads to generate						
16+8	1-bit value	**2 threads**	0.93 Mkeys/sec	0.00%	16.00 MB	2.00
16+8	1-bit value	**4 threads**	1.56 Mkeys/sec	0.00%	16.00 MB	2.00
16+8	1-bit value	**8 threads**	2.28 Mkeys/sec	0.00%	16.00 MB	2.00
16+8	1-bit value	**16 threads**	2.97 Mkeys/sec	0.00%	16.00 MB	2.00

Table 2: Construction throughput of SetSep for 64 M keys with different settings

safe in-place updates, techniques analogous to those proposed in CuckooSwitch [46] and MemC3 [19] could be applied, although we have not designed such a mechanism yet.

4.5 Evaluation

We present micro-benchmark results for SetSep construction and lookup performance on modern hardware. For more detailed results and analysis, please refer to [47].

These micro-benchmarks are conducted on a moderately fast dual-socket server with two Intel Xeon E5-2680 CPUs (HT disabled), each with a 20 MiB L3 cache. The machine has 64 GiB of DDR3 RAM.

4.5.1 Construction

The construction speed of SetSep depends primarily on three parameters:

- The number of bits to store the hash index and to store the bit-array in each group;
- The number of possible values or sets; and
- The number of threads used to parallelize construction.

The first experiments measure the construction rate of SetSep with different parameter combinations. The per-thread construction rate (or throughput) is nearly constant; construction time increases linearly with the number of keys and decreases linearly with the number of concurrent threads. Table 2 shows results for 64 M keys.

The first group of results shows a modest tradeoff between (single-threaded) construction speed and memory efficiency: Using a "16+8" SetSep (where 16 bits are allocated to the hash function index and 8 bits to the bit array) has the slowest construction speed but almost never needs to use the fallback table, which improves both query performance and memory efficiency. "16+16" SetSep also has low fallback ratio, but consumes more space. *We therefore use 16+8 for the remaining experiments in this section*. Its speed, $\frac{1}{2}$ million keys per second per core, is adequate for the read-intensive workloads we target.

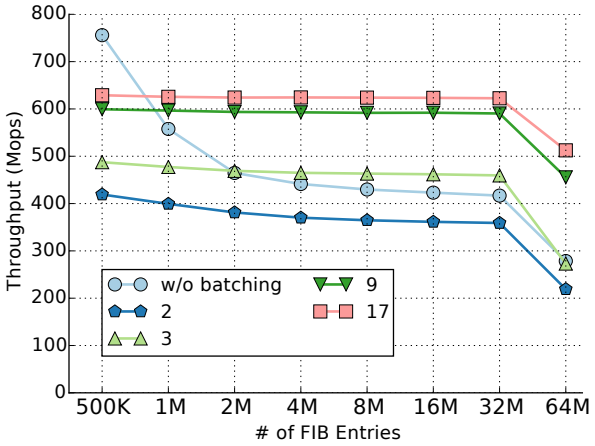


Figure 13: Local lookup throughput of SetSep (GPT)

Increasing the value size imposes little construction overhead. The results in practice are even better than linear scaling because we optimized our implementation as follows: as we iterate the hash function, we test the function for each value bit across the different keys in the group before moving on to the next hash function in the hash function family. As a result, we perform less work than searching hash functions for each value bit one at a time. In addition, storing larger values further amortizes the 0.5 bits of overhead added by the first-level key-to-group mapping.

Summary The core SetSep data structure construction speed is fast enough for a variety of important applications in which the read rate is larger than the (already high) update rate that SetSep can handle. ScaleBricks uses SetSep for its global partition tables, which fall into this category.

4.5.2 Lookup

Figure 13 shows the local lookup throughput of SetSep for different numbers of FIB entries (keys). In addition, given a FIB size, this figure also compares SetSep performance with different batch sizes, varying from 1 (no batching) to 32 (the maximum packet batch size provided by DPDK). All lookup experiments use 2-bit values, a "16+8" configuration, and 16 threads.

These lookup micro-benchmarks provide three insights. First, batching generally increases the lookup performance of SetSep. When the batch size is increased to 17, the lookup throughput is ∼520 Mops (million operations per second) even with 64 million keys; batch sizes larger than 17 do not further improve performance. Second, as the number of FIB entries increases from 32 million to 64 million, the performance drops dramatically. This occurs because the 64 million entry, 2-bit SetSep exceeds the size of L3 cache, occupying 28 MiB of memory. Third, for small FIBs (e.g., 500 K entries), lookup performance is actually higher *without* batching. This too arises from caching: These small structures fit entirely in L3 or even L2 cache, where the access latency is low enough that large batches are not required, but merely increase register pressure.

Summary ScaleBricks batches for all table sizes to ensure fast-enough lookup performance regardless of the number of entries. Because DPDK receives packets in batches, ScaleBricks handles incoming packets using the same *dynamic batching* policy as described in Section 3: instead of having a fixed

batch size, when a CPU core receives a batch of packets from DPDK, ScaleBricks looks up the entire batch in SetSep. Therefore, the batch size of SetSep adjusts semi-automatically to the offered load.

5 (Proposed work) Fast Flow Caching with Bounded Linear Probing

The rise of server virtulization and public cloud services, such as Amazon Web Services [1], Microsoft Azure [4] and Google Cloud Platform [2] drives the developement of virtual switches. Today, the de facto solution for virtual switches is Open vSwitch [5], which supports multiple protocols and provides a switch stack for virtulization environments.

For the control plane, Open vSwitch uses OpenFlow [38] to control and manage network flows. For the data plane, to accelerate the speed of packet classification, Open vSwitch adopts two layers of cache – a microflow cache that remembers per connection forwarding decisions and a megaflow cache that caches forwarding decisions for a group of flows that produce the same result when classified by the input rules.

Recent literature [44] demonstrated the inefficiency of the microflow cache under an adversarial workload. Wang et al. then proposed a new data structure called *the signature-match cache* or SMC to replace the microflow cache. Open vSwitch has incorpated this new cache layer [6] into its datapath.

The SMC is a approximate set membership data structure (ASMDS) that supports probablistic membership testing. Moreover, it allows each key to have an associated value. Therefore, SMC is a data structure that supports the following two operations:

- `Insert`(k,v), which inserts the key-value pair;
- `Lookup`(k), which returns either `NULL` if k has never been added to SMC or some value v', such that with high probability, $v' = v$.

In the current design, the SMC is organized as an array of buckets, each containing four 32-bit entries (i.e., a 4-way associative cache). Among these 32 bits, 16 of them store a *fingerprint* – a bit string derived from the key using a hash function – for each inserted key, and the rest 16 bits are used to store the value. Algorithms 4 and 5 show the pseudocode for SMC insertion and lookup, respectively.

Algorithm 4: `Insert(k,v)`

```
f = fingerprint(k);
i = hash(k);
if bucket[i] has an empty entry then
    add (f, v) to bucket[i];
    return Done;
randomly select an entry e from bucket[i];
replace e with (f,v);
return Done;
```

Although the SMC is proven to be effective against adversarial workloads, it has its own limitations. With Open vSwitch's default configuration (2^{18} buckets, 2^{20} entries) and 1,000,000 random flows, it only achieves a theortical cache hit rate of $\approx 80\%$ and these cache misses are *expensive*. Therefore, the overall performance of Open vSwitch could be improved substantially if the cache miss rate could be lowered.

Algorithm 5: `Lookup(k)`

```
f = fingerprint(k);
i = hash(k);
if bucket[i] has (f,v) then
    return v;
return NULL;
```

To address this limitation, we propose a new data structure, *Bounded Linear Probing Cache* or BLPC, which improves the cache hit rate of the SMC without sacrificing lookup performance. BLPC does so by harnessing a simple while powerful idea: instead of pinning each key to only one bucket, a key hashed to one bucket is allows to be placed either in that bucket, or in one of the next H-1 buckets, where H is a constant. This idea is not new: *hopsctoch hashing* [27] uses the exact same idea to build a high performance hash table. However, *hopsctoch hashing* requries full keys (or pointers to full keys) to be stored within the hash table in order to perform key displacement. Given a fixed memory size, this requirement significantly reduces the number of keys it can support. BLPC leverages the nature of a cache, which requires only eviction instead of displacement, so it can store partial keys instead of full keys. Assume each bucket has a capacity of K, we call such a data structure (H,K)-BLPC.

5.1 Bounded Linear Probing Cache

In this secion, we present the design of BLPC.

Algorithm 6: `Insert(k,v)` for (2,4)-BLPC

```
f = fingerprint(k);
i = hash(k);
if bucket[i] has an empty entry then
    add (f, v) to bucket[i];
    return Done;
j = next(i);
if bucket[j] has an empty entry then
    add (f, v) to bucket[j];
    return Done;
random select an entry e from bucket[i];
replace e with (f,v);
return Done;
```

Algorithms 6 and 7 differ from the original algorithms by allowing a key to be placed not only within its original bucket (*bucket*[hash(k)]), but also the bucket next to its original bucket (*bucket*[next(hash(k))]). The advantage of this new algorithm is clear: with increased set-associativity, the occupancy of the table becomes higher. The disadvantages, however, are two-fold. First, although there are fewer cache misses, however, because we only store *fingerprints* in the table, the number of false positives (i.e., SMC returns a wrong value for the key k) will increase. Second, because a key could be stored in two adjacent buckets, it takes more CPU cycles to perform insertion and lookup.

Algorithm 7: `Lookup(k,v)` for (2,4)-BLPC

```
f = fingerprint(k);
i = hash(k);
if bucket[i] has (f,v) then
    return v;
j = next(i);
if bucket[j] has (f,v) then
    return v;
return nil;
```

Cache setting	Lookup throughput	False positive ratio	Cache hit rate
SMC: 1 bucket, 4 way-associated	12.796 Mops/sec	0.020808%	81%
SMC: 1 bucket, 8 way-associated	11.41 Mops/sec	0.03818%	86%
(2,4)-BLPC (2 buckets, 4 way-associated)	12.194 Mops/sec	0.038924%	88%

Table 3: Lookup throughput, false positive ratio and cache hit rate of (2,4)-BLPC for 1 M keys

5.2 Preliminary Results

We present the preliminary results for BLPC. In this microbenchmark, we pre-generated 1 M key/value pairs and a random stream of lookup keys. A lookup is considered as cache hit if and only if the cache returns the correct value for the lookup key. Table 3 shows the lookup throughput, false positive ratio and cache hit rate of three cache designs. As we can see from the table, (2,4)-BLPC achieves much higher cache hit rate than the SMC with low penalty to lookup throughput. More interestingly, compared with an alternative design where we simply increase the associativity of each bucket from 4 to 8, (2,4)-BLPC has higher throughput and cache hit rate.

5.3 Future Work

The generalized goal of this work is to answer the following question: given a hash table design, what is the best way to turn it into a hash cache? This seemingly easy question is indeed hard to answer – a hash table needs to accommodate all the inserted items but a cache does not. This flexiblity makes many straightforward design decisions less clear. For example, in many hash table designs, an existing key could be displaced, or the table could be expanded, to make room for the new key. In a cache, however, eviction is a viable alternative to displacement. In fact, given a fixed cache size, the tradeoff between eviction and displacement sits at the core of designing a high performance hash cache. Advanced displacement algorithms (such as cuckoo hashing and hopscotch hashing) often increase the cache hit rate at the cost of lookup/insertion throughput. Eviction, on the other hand, lowers the cache hit rate but offers great performance.

We summarize a list of research questions that needs to be answered:

1. Our preliminary results shows that BLPC achieves higher cache hit rate than SMC with little performance penalty in the microbenchmark. How does BLPC perform in the full system evaluation?

2. How should we balance displacement and eviction?

3. What kind of evicition policy should we use?

4. Some applications could tolerate an approximate cache, meaning the cache could return a wrong value for the lookup key. Compared with the scenario where such false positives are not allowed, what differences does it make?

5. Are there any general rules / principles we should follow when designing a hash cache?

We would like to gain more insights into these questions in the final dissertation.

6 Related Work

Hash Tables Cuckoo hashing [39] is an open-addressing hashing scheme with high space efficiency that assigns multiple candidate locations to each item and allows inserts to kick existing items to their candidate locations. FlashStore [14] applied cuckoo hashing by assigning each item 16 locations so that each lookup checks up to 16 locations. SILT [34] used an early version of *partial key cuckoo hashing* and it achieved high occupancy with only two hash functions, but has a limitation in the maximum hash table size. This thesis eliminates the table size limitation while still retaining high memory efficiency. To make cuckoo operations concurrent, the prior approach of Herlihy and et. al. [26] traded space for concurrency. In contrast, our optimistic locking scheme achieves high read concurrency without losing space efficiency.

Conventional Solutions for Global Partition Table There is a wealth of related work on building efficient dictionaries mapping keys to values.

Standard **hash tables** cannot provide the space and performance we require. Typically, they store keys or fingerprints of keys to resolve collisions, where multiple keys land in the same hash table bucket. Storing keys is space-prohibitive for our application. To reduce the effect of collisions, hash tables typically allocate more entries than the number of elements they plan to store. Simple hashing schemes such as linear probing start to develop performance issues once highly loaded (70–90%, depending on the implementation). Multiple-choice based hashing schemes such as cuckoo hashing [39] or d-left hashing [36] can achieve occupancies greater than 90%, but must manage collisions and deal with performance issues from using multiple choices.

Perfect Hashing schemes try to find an injective mapping onto a table of m entries for n ($n \leq m$) distinct items from a larger universe. Seminal early work in the area includes [16, 22, 21], [21], and such work refers to set separating families of hash functions. More recent work on attempting to design perfect hash functions for on-chip memory [35] is most similar to ours. Our approach uses both less space and fewer memory accesses.

Bloom Filters [9] are a compact probabilistic data structure used to represent a set of elements for set-membership tests, with many applications [11]. They achieve high space efficiency by allowing false positives.

Bloom filters and variants have been proposed for set separation. For example, BUFFALO [45] attempts to scale the forwarding table of a network switch. It does so by looking up the destination address in a sequence of Bloom filters, one per outgoing port. However, this approach to set separation is inefficient. A query may see positive results from multiple Bloom filters, and the system must resolve these false positives. SetSep is also more space efficient. Finally, updating the Bloom filter to change the mapping of an address from port x to port y is expensive, because it must rebuild the filter to delete

a single item, or use additional structure (such as counting Bloom filters). Bloomier filters [12] provide an alternative approach; the value associated with a key is the exclusive-or of values set in multiple hash table locations determined by the key. Approximate concurrent state machines [10] similarly provide an alternative structure based on multiple choice hashing for efficiently representing partial functions. Our approach is again more scalable than these approaches.

Scaling Forwarding Tables Bloom filters [9] have been used to improve memory-efficiency in packet forwarding engines. Dharmapurikar et al. [15] deployed Bloom filters to determine which hash table to use for a packet's next hop. Their scheme often requires additional hash table probes per address lookup on false positives returned by the Bloom filters. BUFFALO has a similar goal, as noted above.

7 Plan

We propose the following timeline to finish the remainder of this thesis:

Timeline	Action	Current Status
Jan 2019 – Feb 2019 (2 months)	Complete proposed work for bounded linear probing cache	Design and implementation is done, it needs more performance optimization and evaluation
Mar – May 2019 (3 months)	Evaluate full system	Null
Jun – Aug 2019 (3 months)	Write Thesis and prepare for defense	Null

References

[1] Amazon Web Services. `http://aws.amazon.com/`.

[2] Google Cloud Platform. `https://cloud.google.com/`.

[3] libcuckoo. `https://github.com/efficient/libcuckoo`.

[4] Microsoft Windows Azure. `http://www.microsoft.com/windowsazure/`.

[5] Open vSwitch. `http://www.openvswitch.org`.

[6] [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`.

[7] Intel 64 and IA-32 architectures optimization reference manual. `http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf`, 2012.

[8] 3GPP. The Evolved Packet Core. `http://www.3gpp.org/technologies/keywords-acronyms/100-the-evolved-packet-core`.

[9] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[10] Flavio Bonomi, Michael Mitzenmacher, and Rina Panigrahy. Beyond Bloom filters: From approximate membership checks to approximate state machines. In *Proc. ACM SIGCOMM*, Pisa, Italy, August 2006.

[11] Andrei Broder, Michael Mitzenmacher, and Andrei Broder. Network Applications of Bloom Filters: A Survey. In *Internet Mathematics*, volume 1, pages 636–646, 2002.

[12] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, Ayellet Tal, and Oh Boy. The Bloomier filter: An efficient data structure for static support lookup tables. In *Proceedings of SODA*, pages 30–39, 2004.

[13] Connectem Inc. `http://www.connectem.net/`, 2017.

[14] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: high throughput persistent key-value store. *Proc. VLDB Endow.*, 3:1414–1425, September 2010.

[15] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor. Longest prefix matching using Bloom filters. In *Proc. ACM SIGCOMM*, Karlsruhe, Germany, August 2003.

[16] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer Auf Der Heide, Hans Rohnert, and Robert E Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[17] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[18] 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)*, CA, USA, January 2006.

[19] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proc. 10th USENIX NSDI*, Lombard, IL, April 2013.

[20] Bin Fan, Dong Zhou, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. When cycles are cheap, some tables can be huge. In *Proc. HotOS XIV*, Santa Ana Pueblo, NM, May 2013.

[21] Michael L Fredman and János Komlós. On the size of separating systems and families of perfect hash functions. *SIAM Journal on Algebraic Discrete Methods*, 5(1):61–68, 1984.

[22] Michael L Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with O(1) worst case access time. *Journal of the ACM (JACM)*, 31(3):538–544, 1984.

[23] Younghwan Go, Muhammad Asim Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. APUNet: Revitalizing GPU as packet processing accelerator. In *Proc. 14th USENIX NSDI*, Boston, MA, March 2017.

[24] W. Haeffner, J. Napper, M. Stiemerling, D. Lopez, and J. Uttaro. *Service Function Chaining Use Cases in Mobile Networks*. Internet Engineering Task Force, January 2015.

[25] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-accelerated software router. In *Proc. ACM SIGCOMM*, New Delhi, India, August 2010.

[26] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[27] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *Proceedings of the 22Nd International Symposium on Distributed Computing*, 2008.

[28] Intel. Intel Data Plane Development Kit (Intel DPDK). `http://www.intel.com/go/dpdk`, 2013.

[29] Intel 64 and IA-32 architectures developer's manual: Vol. 3A. `http://www.intel.com/content/www/us/en/architecture-and-technology/`, 2011.

[30] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G. Andersen. Raising the bar for using GPUs in software packet processing. In *Proc. 12th USENIX NSDI*, Oakland, CA, May 2015.

[31] Changoon Kim, Matthew Caesar, and Jennifer Rexford. Floodless in SEATTLE: A scalable Ethernet architecture for large enterprises. In *Proc. ACM SIGCOMM*, Seattle, WA, August 2008.

[32] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better Bloom filter. *Random Structures & Algorithms*, 33(2):187–218, 2008.

[33] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proc. 9th ACM European Conference on Computer Systems (EuroSys)*, April 2014.

[34] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011.

[35] Yi Lu, Balaji Prabhakar, and Flavio Bonomi. Perfect hashing for network applications. In *Information Theory, 2006 IEEE International Symposium on*, pages 2774–2778. IEEE, 2006.

[36] M. Mitzenmacher and B. Vocking. The asymptotics of selecting the shortest of two, improved. In *Proc. the Annual Allerton Conference on Communication Control and Computing*, volume 37, pages 326–327, 1999.

[37] Michael Mitzenmacher, Andréa W. Richa, and Ramesh Sitaraman. The power of two random choices: A survey of techniques and results. In *Handbook of Randomized Computing*, pages 255–312. Kluwer, 2000.

[38] OpenFlow Switch Consortium. `http://www.openflowswitch.org/`, 2008.

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

[40] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *Proc. 12th USENIX OSDI*, Savannah, GA, November 2016.

[41] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. Bandwidth-efficient live video analytics for drones via edge computing. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2015)*, Oakland, CA, 2015.

[42] Luigi Rizzo. netmap: a novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, June 2012.

[43] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81.

[44] Yipeng Wang, Tsung-Yuan Charlie Tai, Ren Wang, Sameh Gobriel, Janet Tseng, and James Tsai. Optimizing open vswitch to support millions of flows. In *Proceedings of the 2017 IEEE Global Communications Conference (GLOBECOM 2017)*, Singapore, Singapore, 2017.

[45] Minlan Yu, Alex Fabrikant, and Jennifer Rexford. BUFFALO: Bloom filter forwarding architecture for large organizations. In *Proc. CoNEXT*, December 2009.

[46] Dong Zhou, Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proc. 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, December 2013.

[47] Dong Zhou, Bin Fan, Hyeontaek Lim, David G. Andersen, Michael Kaminsky, Michael D. Mitzenmacher, Ren Wang, and Ajaypal Singh. Scaling up clustered network appliances with ScaleBricks. In *Proc. ACM SIGCOMM*, London, UK, August 2015.