Scalable, High Performance Ethernet Forwarding with CUCKOOSWITCH

Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky[†], David G. Andersen
Carnegie Mellon University, [†]Intel Labs

{dongz,binfan,hl,dga}@cs.cmu.edu, michael.e.kaminsky@intel.com

ABSTRACT

Several emerging network trends and new architectural ideas are placing increasing demand on forwarding table sizes. From massive-scale datacenter networks running millions of virtual machines to flow-based software-defined networking, many intriguing design options require FIBs that can scale well beyond the thousands or tens of thousands possible using today's commodity switching chips.

This paper presents CUCKOOSWITCH, a software-based Ethernet switch design built around a memory-efficient, high-performance, and highly-concurrent hash table for compact and fast FIB lookup. We show that CUCKOOSWITCH can process 92.22 million minimum-sized packets per second on a commodity server equipped with eight 10 Gbps Ethernet interfaces while maintaining a forwarding table of *one billion* forwarding entries. This rate is the maximum packets per second achievable across the underlying hardware's PCI buses.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network communications; C.2.6 [**Internetworking**]: Routers; E.1 [**Data**]: Data Structures; E.2 [**Data**]: Data Storage Representations

General Terms

Algorithm, Design, Measurement, Performance

Keywords

Software Switch; Cuckoo Hashing; Scalability

1. INTRODUCTION

This paper 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, this paper contributes not just a set of concrete techniques for architecting the forwarding table (or FIB) of such a device, but also shows that, indeed, network designs that require huge FIBs *could* be implemented in a practical manner.

Our work is motivated by several trends in network architecture: First, enterprise networks [22] and datacenter networks [16, 25, 7] have been rapidly growing in scale, both because of the adoption of high-bisection topologies and with the emergence of low-cost, high-data-rate switches from new 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, building. The addition of virtualization, where machines may have multiple addresses for hosted VMs, further increases this scale.

Second, 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-based 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 desire lookup tables that contain an entry for every cache-able chunk of content. While these latter motivations are still undergoing research, we hope that by showing that huge FIBs can be practically implemented, the designers will have more freedom to choose the best overall approach.

In all of these contexts, line speeds continue to increase, with 10 Gbps and 40 Gbps links now becoming both common and affordable. The consequence is that we are now asking network switches to handle not just more lookups per second, but more lookups per second into increasingly larger tables. Existing solutions have been unable to achieve all of these goals.

Perhaps the most common approach to building fast Ethernet switches is to use a custom ASIC coupled with specialized, high-speed memory (e.g., TCAM) to store the forwarding table. These memories, unfortunately, are expensive, power hungry, and very limited in size. For example, the midrange Mellanox SX1016 64-Port 10GbE switch [2] supports only 48K Layer-2 forwarding entries.

Software-based switches on commodity hardware, in contrast, can affordably store larger tables in SRAM (CPU cache) or DRAM, and we take this approach in this paper. Conventional hash-table based lookup tables, however, are typically memory-inefficient, which becomes important when attempting to achieve large scale or fit lookup tables into fast, expensive SRAM. There are two general reasons for this inefficiency: First, extra space is required to avoid collisions in the hash tables, often increasing their size by 2× or more. Second, for high performance, to avoid locking overhead while allowing multiple threads to read from the forwarding table,

CoNEXT'13, Dec 09–12 2013, Santa Barbara, CA, USA.
ACM 978-1-4503-2101-3/13/12.
http://dx.doi.org/10.1145/2535372.2535379

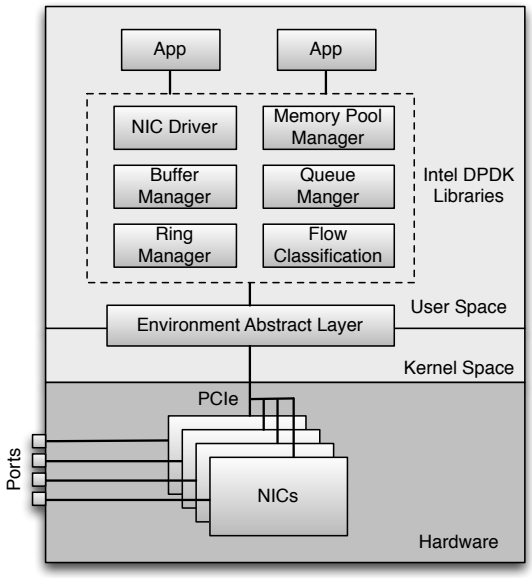

Figure 1: Architecture of Intel DPDK.

pointer-swapping approaches such as Read-Copy Update (RCU) [5] are often used to update the FIB, which require storing two full copies of the table instead of one.

This paper therefore proposes 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 Intel's just-released high-performance packet processing architecture (the Intel Data Plane Development Kit [20], or DPDK for short) to create a best-of-breed software switch. The forwarding table design is based upon memory-efficient, high-performance concurrent cuckoo hashing [14], 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.

Our evaluation (Section 4) shows that CUCKOOSWITCH can achieve the full 64-byte packet forwarding throughput of our eight-port 10GbE switch with one *billion* FIB entries.

2. BUILDING BLOCKS

In this section, we provide the salient details about two important components that our system builds upon: First, the Intel Data Plane Development Kit, which we use unmodified for high-throughput packet I/O in user-space; and second, *optimistic concurrent cuckoo hashing*, which we extend and optimize as described in Section 3 to create a high-throughput forwarding table for switching applications.

2.1 Intel Data Plane Development Kit

Intel's Data Plane Development Kit, or 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 hands batches of packets to processing threads to be processed together. These techniques combine to provide developers

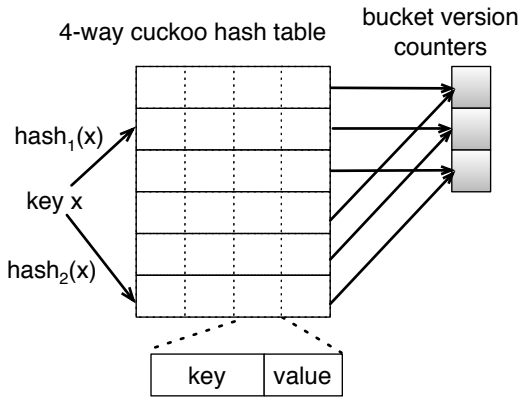

Figure 2: A 2,4 cuckoo hash table.

with a straightforward programming environment for constructing extremely high-performance packet processing applications. Figure 1 shows the high level architecture of Intel DPDK. Our system uses the DPDK for all of its packet I/O. Readers familiar with prior work such as RouteBricks [12] and the PacketShader IO-engine [18] 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 system:

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 paper, and as a consequence, they benefit from this NUMA awareness, alignment, and huge page table support.

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, as in prior work such as the PacketShader's I/O engine, RouteBricks and Netmap [18, 12, 28], the DPDK delivers packets in large batches for efficiency. As a result, many of our techniques (Section 3) also emphasize batching for efficiency in a way that is aligned with the DPDK's batching.

2.2 Optimistic Concurrent Cuckoo Hashing

Our second building block is our recent multiple-reader, single writer "optimistic concurrent cuckoo hashing" [14]. As described in Section 3, 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 [26], an open addressing hashing scheme. It achieves high memory efficiency, and ensures expected O(1) retrieval time and insertion time. As shown in Figure 2, cuckoo hashing maps each

item to multiple candidate locations by hashing[1] 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% [13].

Optimistic concurrent cuckoo hashing [14] 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.[2] 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.

Before the changes we make in this paper, "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. DESIGN AND IMPLEMENTATION

Our goals for CUCKOOSWITCH are twofold: First, achieve extremely high packet forwarding rates; as close to the limit of the hardware as possible. And second, store the switching FIB in as little space as possible in order to minimize the physical size, cost, and power consumption of a software-based switch using this design.

CUCKOOSWITCH achieves high throughput and memory efficiency through architectural and application-specific improvements to its lookup data structure – optimistic concurrent cuckoo hashing. This section first presents the overall view of packet processing and how CUCKOOSWITCH uses the DPDK to perform packet I/O between user-level threads and the underlying hardware. It then discusses the architectural and algorithmic improvements we make to the FIB structure to achieve fast lookups and high-performance packet switching.

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 4. 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 Packet Processing Overview

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

3.2 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

[1] We use terms "location" and "bucket" interchangeably in the paper.

[2] 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.

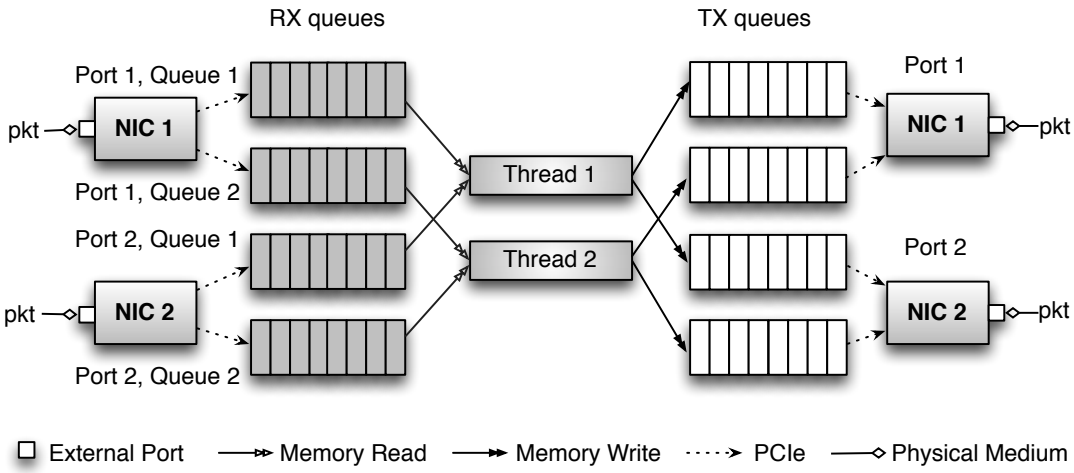

Figure 3: Packet processing pipeline of CUCKOOSWITCH.

can be found in Section 8.2.2 of the Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A [21]. 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 [19] 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 [1] 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 [21].

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 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.

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)
        v'1, v'2 ← b1, b2's version (by atomic read)
        if v1 != v'1 or v2 != v'2 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)
```

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,

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)
```

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[3] between the version counter reads, data read, and subsequent version counter reads.[4] 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.

3.3 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 4, 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 4.

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

3.4 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[5]. 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

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

[4] 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.

[5] 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 [4].

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]
```

also ensures that all of the lookups will be processed with only two 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 4 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.

3.5 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 [4]. The factor of 256× 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.

4. EVALUATION

The evaluation proceeds in three parts. First, we evaluate the "raw" packet forwarding throughput of our experimental platform, with no switching or FIB lookups involved. These results give us a baseline for the forwarding capacity of our hardware platform when using the DPDK.

Second, we examine 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.

Third, we compare CUCKOOSWITCH with other popular hash table implementations, for both hash table micro-benchmarks and full system evaluation.

The optimized cuckoo hash table can serve over 400 million small key/value (key+value fits in 64 bits) lookups per second with 16 threads if the hash table fits in L3 cache. When the hash table is too large for cache, the optimizations we applied successfully mitigate the high DRAM access latency, maintaining the throughput at over 300 million lookups per second, even with *one billion* entries in the table, while remaining completely thread-safe. The end-to-end system evaluation shows that CUCKOOSWITCH achieves maximum 64-byte packet throughput (it is bottlenecked by the PCIe interface to the network cards) even with one billion forwarding entries.

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.

4.1 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 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-

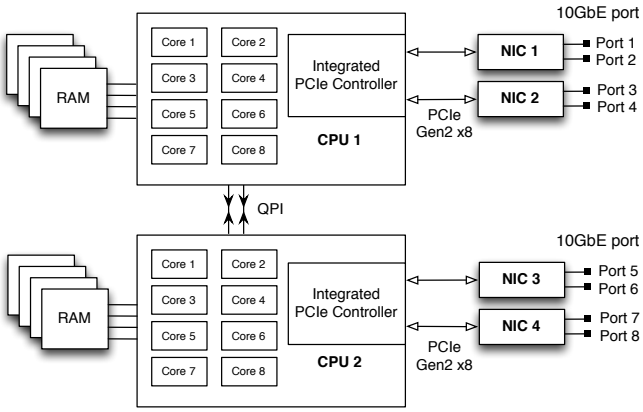

Figure 4: Topology of the evaluation platform.

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.

sized 64 byte packets. This is the peak packet rate achievable on 10GbE [28].

Other Hash Tables Used for Comparison As one component of our evaluation, we compare both the hash table and packet forwarding throughput of optimized optimistic concurrent cuckoo hashing with several popular hash tables.

One of optimistic cuckoo's major benefits is that it can be read concurrently by many threads while being simultaneously updated. Few existing concurrent hash tables have good performance under these circumstances. We evaluate the Intel TBB (Threading Building Blocks) [6]'s `concurrent_hash_map` as one thread-safe example. Its performance is poor in our scenario, because it is poorly-matched to the read-intensive nature of a switch FIB. We therefore also compare to three *non-thread-safe* hash tables: the STL's `hash_map` and Google's `sparse_hash_map` and `dense_hash_map`.

It is worth noting that `sparse_hash_map`, `dense_hash_map` and STL's `hash_map` use the identity function to compute the hash for integer keys, whereas ours uses the stronger CRC32c hash. We believe this difference accounts for part of the higher throughput achieved by `dense_hash_map` when the FIB is sufficiently small to fit in L3 cache.

These non-thread-safe tables do not support concurrent reads and writes. In our evaluation, they are given the benefit of the doubt and used as an immutable store that multiple threads read from at a time. Such a design is not uncommon in switches, and is often handled using "pointer swapping": In order to update a read-only hash table, all updates are directed to a writable copy of the table, and the readable and writable copies of the tables are periodically exchanged. This technique is similar to RCU (read-copy-update) synchronization that is popular in the Linux kernel. It provides a way to achieve the throughput of these fast read-only hash tables, but requires doubling their memory use to have two copies of the table at a time. Because DRAM is an expensive and power-hungry component, designs that support in-place updates have a substantial advantage, *if* they can provide comparably high throughput.

Packet Size (Bytes)	Throughput (Mpps)	Throughput (Gbps)	Bottleneck
64	92.22	61.97	PCIe bandwidth
128	66.24	78.43	PCIe bandwidth
192	47.17	80.00	Network bandwidth
256	36.23	80.00	Network bandwidth

Table 2: Raw packet I/O throughput.

4.2 Raw Packet I/O Throughput

We first examine the raw packet I/O achieved by our evaluation platform when handling packets of different sizes. Here, raw packet I/O is defined by forwarding a packet back out its incoming port without performing any processing. Because this experiment requires no forwarding table lookup, its throughput provides the upper bound/maximum throughput that CUCKOOSWITCH can possibly achieve.

Table 2 shows the results. When doing raw packet I/O with 64-byte Ethernet packets, our evaluation system can only process 61.97 Gbps (or 92.22 Mpps) in total, which is 20% lower than the theoretical maximum throughput (80 Gbps). When the packets are larger than 128 bytes, the 8 ports can be saturated and achieve 80 Gbps throughput. To identify the bottleneck, we measured the throughput of each individual port with 64-byte packets. Our hardware *can* saturate each individual *port*, but can only achieve 23.06 Mpps through two ports of the same NIC. All four NICs behave the same, and throughput scales linearly when going from one to four NICs.

We believe the NICs themselves are limited by their PCIe Gen2 x8 connections when the overhead of sending small packets over PCI is taken into account. These x8 links provide 32 Gbps in each direction (more than the 20 needed). However, once PCIe transaction level packets and the PCIe ACK/NACK packets are considered, we believe it is insufficient. Intel's own DPDK results are very similar to our throughput [3], achieving 93 Mpps with 4 dual-port NICs, and they note that their system is PCI-limited.

The good news for the scaling of our software-based switch is that the inter-processor QPI is *not* a bottleneck. We perform a similar experiment in which packets are forwarded to a predetermined *remote* outgoing port based upon its incoming port ID, i.e., the outgoing port is always on the other CPU socket. Our throughput results are identical in this cross-CPU case, confirming that QPI bandwidth is plentiful.

A second important result of this experiment is that it shows that the system achieves full hardware throughput when forwarding a single high-bandwidth flow per input port. In this experiment, packets arriving at a single port were sent to a single RX queue processed by a single thread, as would happen with one flow under RSS. Thus, in the rest of our evaluation, we focus on experiments with addresses drawn uniformly at random from the entire FIB. This is the worst-case workload for the hash table, as caching is ineffective.

Summary: With minimum-sized packets, the upper bound of CUCKOOSWITCH's throughput on our testbed is 61.97 Gbps (92.22 Mpps), and this throughput is limited by PCIe bandwidth from

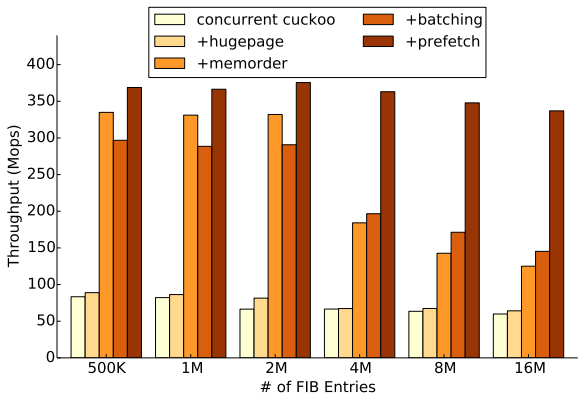

Figure 5: Contribution of optimizations to the hash table performance. Optimizations are cumulative.

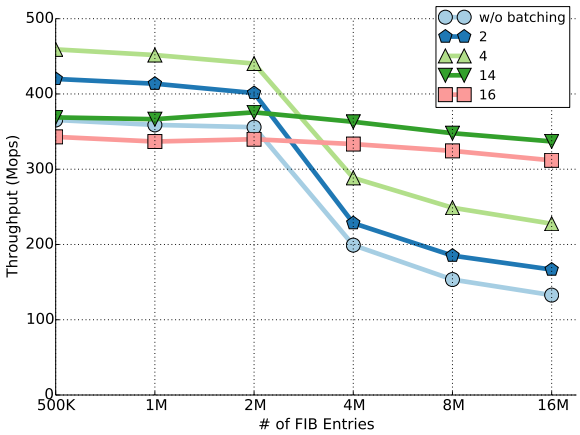

Figure 6: Lookup throughput of hash table vs. batch size.

NICs to CPUs. With larger packets, the system can achieve the full network bandwidth.

4.3 Hash Table Micro-benchmark

We next 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 described in Section 3 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, "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 in Section 3, 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 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

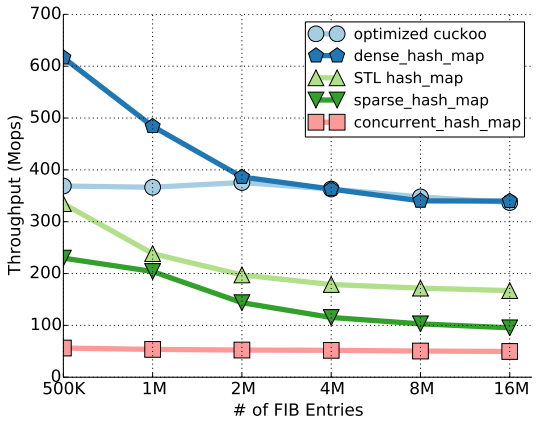

Figure 7: Lookup throughput comparison between optimized cuckoo hashing and other hash table implementations.

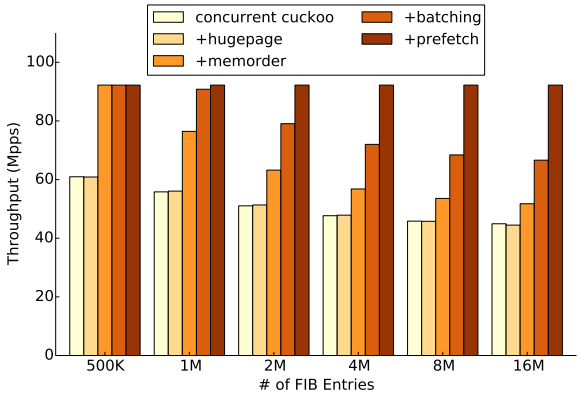

Figure 8: Full system packet forwarding throughput factor analysis. Packet size = 64B.

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.

Comparing with Other Hash Tables Figure 7 shows that our optimized concurrent cuckoo hashing outperforms the other hash tables except Google's `dense_hash_map` with small tables, which uses more than 4× the memory of ours. Importantly, as noted earlier, `dense_hash_map` uses the identity function has its default hash function for 64-bit unsigned integers (all the MAC addresses are encoded using this type), while ours uses CRC32c hash, which is more expensive in computation but ensures better hashing results for general workloads. Moreover, `dense_hash_map` is *not* thread-safe, and so in practice, making use of it would likely require doubling its memory use to store a writable copy of the table.

In contrast, optimized cuckoo hashing supports *in-place, concurrent* updates that eliminate the necessity of such techniques. It achieves nearly 350 Mops even if the table does not fit in cache, roughly 5.6× faster than the original optimistic cuckoo hashing scheme.

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.

4.4 Full System Forwarding Evaluation

These experiments evaluate: (1) how each optimization improves the full system packet forwarding throughput; (2) how CUCKOOSWITCH compares with software switches using other hash tables to store their FIBs; and (3) how CUCKOOSWITCH performs with different rates of FIB updates. 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 8 shows the results across different FIB sizes. In total, the hash table optimizations boost throughput by roughly 2×. CUCKOOSWITCH achieves maximum 64 byte packet throughput with even *one billion* FIB entries.

Comparing with Other Hash Tables As before, we now compare the packet forwarding throughput of CUCKOOSWITCH with one variant implemented using *immutable* `dense_hash_map` as its forwarding table. The reason why we only compare CUCKOOSWITCH with `dense_hash_map` is that among all the other hash tables, `dense_hash_map` offers the best lookup performance in the hash table micro-benchmarks, at the cost of using 4× more space and being non-thread-safe. Figure 9 shows that CUCKOOSWITCH always achieves higher throughput than the `dense_hash_map`-based variant. This is perhaps surprising, given that in the pure hash table benchmarks, `dense_hash_map` outperforms optimized cuckoo hash table. The difference arises because stock `dense_hash_map` does not do any batching, which becomes more significant when performing substantial amounts of work (sending packets) between lookups to the hash table.[6]

To be more fair in our comparison, we implemented *dynamic batched* immutable `dense_hash_map` using the same batching policy as optimized cuckoo hashing. As we expected, the dynamic batched version outperforms the original and achieves nearly maximum throughput with up to 16 million forwarding entries. When the forwarding entries become more than 16 million, due to the increase of cache misses and TLB misses, we observe roughly a linear decrease in throughput.

Table 3 shows the memory consumption of CUCKOOSWITCH and batched `dense_hash_map` at FIB sizes up to one billion entries. Two details stand out: 1) immutable dynamic batched `dense_hash_map` requires about 4× more memory than CUCKOOSWITCH; and 2)

[6] An out-of-order CPU such as those used in experiments can only look a limited number of instructions ahead to find additional memory references to issue.

# FIB Entries	32M	64M	125M	250M	500M	1B
optimized cuckoo hashing						
Size (GiB)	0.25	0.50	1.00	2.00	4.00	8.00
immutable dynamic batched dense_hash_map						
Size (GiB)	1.18	2.17	4.21	8.38	16.32	N/A

Table 3: Full system memory consumption w/ large FIBs.

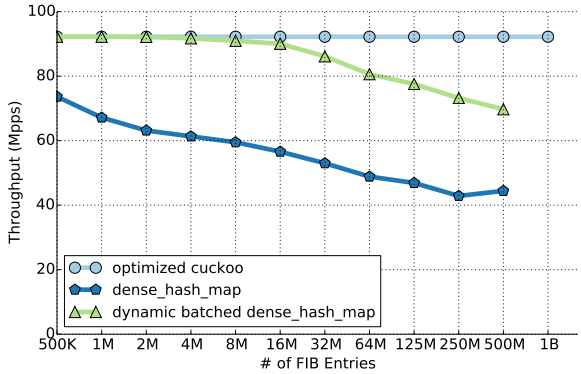

Figure 9: Full system packet forwarding throughput w/ different lookup tables. Packet size = 64B

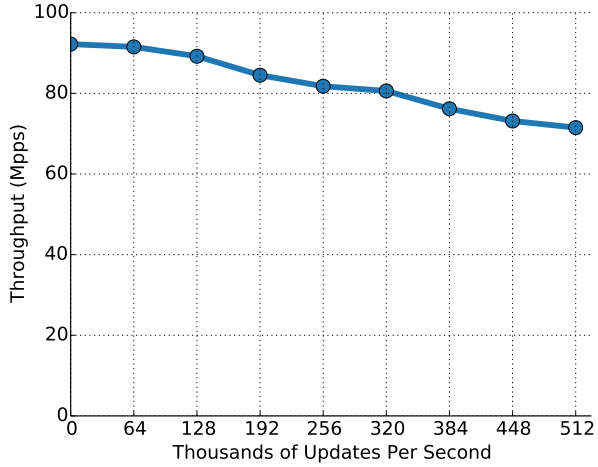

Figure 10: Full system packet forwarding throughput vs. update rate. # FIB entries = 2M, packet size = 64B.

it cannot support 1 billion entries (it runs out of memory on our evaluation machine).

Larger Packets As shown in Table 2, our evaluation platform can saturate the full 80 Gbps of bandwidth for 192 byte or larger packets. CUCKOOSWITCH can achieve the same throughput for these larger packet workloads while performing FIB lookups, even with one billion entries in the FIB.

FIB Updates We have emphasized that one of the benefits of CUCKOOSWITCH is that it supports *in-place* updates, which eliminates the need for pointer-swapping techniques such as Read-Copy-Update (RCU) to update the FIB. Thus, CUCKOOSWITCH can respond instantly to FIB updates and can avoid the need to store multiple copies of the forwarding table. In this experiment we evaluate how high rates of FIB updates affect packet forwarding throughput.

Figure 10 shows the decrease in packet forwarding throughput using 2 million FIB entries with increasingly fast update rates. Note that the second point on the graph is 64,000 updates per second, a rate that corresponds to completely filling the FIB of a conventional ASIC-based switch in a single second. Higher rates of FIB updates do, however, reduce forwarding performance modestly. In the optimized scheme, each update must acquire and release the global lock, and increment at least two version counters (the exact number of increments depends on the length of cuckoo path). These updates increase inter-processor cache coherence traffic, and contend for CPU time with threads that are forwarding packets. On the whole, we believe that these results are quite positive: CUCKOOSWITCH supports tens of thousands of updates per second with little slowdown, and can support extremely high update rates (over half a million per second) with only a 25% slowdown in minimum-size packet forwarding rates.

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, which cannot be achieved by other forwarding table designs. At the same time, by reducing space consumption to only one fourth of batched `dense_hash_map`, CUCKOOSWITCH significantly improves the energy/cost efficiency.

5. DISCUSSION

Going beyond Software Switches. While we have explored our FIB design primarily in the context of a software switch, we believe it is likely to have lessons for hardware as well. For example, the overall lookup throughput results for the hash table itself shows that with one billion entries, the table achieves about 350 million lookups/second. This lookup rate far exceeds what our PCIe-based platform can achieve. Integrating an SRAM lookup version of this with a hardware switching platform is an intriguing (if potentially difficult and expensive) avenue of future work. Similarly, we believe it is worth exploring if a lookup architecture such as ours could augment an OpenFlow-style switch to act as a large, fast local cache of FIB entries. With sufficient fine-grained locality in packets, by avoiding the need to go off-switch to a controller for many packet destinations, an OpenFlow switch may be able to handle a larger number of flows without needing a larger TCAM.

Better Update Support Although the half-million updates per second that our table supports exceeds the needs of conventional Ethernet, we believe it should be possible to apply similar batching and

prefetching optimizations to the write path as well for applications that require higher update rates.

Beyond simply processing updates, the huge table sizes enabled by CUCKOOSWITCH immediately raise the question of how to handle inter-node route/switch updates in an efficient manner, a question we leave for future exploration.

6. RELATED WORK

Memory Efficient Forwarding Prior work has explored alternative memory-efficient data structures for packet forwarding. Dharmapurikar et al. [11] proposed longest prefix matching for IP addresses using Bloom filters [9] to avoid using expensive and power-hungry TCAMs. Their approach uses Bloom filters to decide which conventional hash table to use for the final lookup, so this latter portion of their scheme could benefit by using CUCKOOSWITCH. BUFFALO [30] takes an alternative approach to FIB lookup by testing each packet against one Bloom filter for each output port. The Bloom filters occasionally generate false positives; therefore, a certain number of packets are forwarded to incorrect ports, increasing the path stretch experienced by packets, and thus their end-to-end latency. Because standard Bloom filters do not support updates without an expensive rebuilding operation, they handle FIB updates using counting Bloom filters [15], which take significantly more memory. Although BUFFALO uses substantially less memory than conventional techniques, it is only slightly more memory efficient than CUCKOOSWITCH, using roughly 7.5 bytes per FIB entry vs. 8.5 bytes for CUCKOOSWITCH, and must use 4× additional off-lookup-path memory for the counting Bloom filters.

Flat Addresses Flat addresses help enterprise and datacenter network operators deploy simpler topologies (e.g., SEATTLE [22]) and facilitate host mobility in wide-area networks by making a host's address independent of its location (e.g., ROFL [10], AIP [8]). XIA [17] discusses scalability issues in using flat addresses in wide-area networking; it shows that the increase in the forwarding table size from 10K to 10M FIB entries decreases forwarding performance only by 8.3–30.9% using a software router; XIA does not show how to scale to billions of FIB entries. CUCKOOSWITCH's flat address forwarding design is not specific to Ethernet MAC addresses, and is applicable to these scenarios.

Software Routers RouteBricks [12] was one of the first software routers achieving 35 Gbps Layer-3 packet forwarding. As in CUCKOOSWITCH, RouteBricks uses the multiple queue support in modern NICs to parallelize packet processing within a server. The authors further discuss a switch cluster to scale out beyond single-node performance; we believe that the same approach can be used to build a cluster of CUCKOOSWITCH to for even higher scalability. PacketShader [18] extends software-based routing performance by using GPUs, which have high amounts of parallelism and large memory bandwidth compared to CPUs, allowing them to bring IPv6 forwarding speed close to that of IPv4. While these software routers can perform prefix routing, CUCKOOSWITCH provides higher throughput and supports much larger FIBs.

VALE [29] and Hyper-Switch [27] are two software virtual switches that target high speed communication among virtual machines collocated on the same physical server and with the outside. There are two noticeable differences between CUCKOOSWITCH and them. First, the emphasis of these virtual switch architectures is not on the scalability and performance of the forwarding table. Instead, they both assume that forwarding table lookup is cheap and that the FIB will be small. As we demonstrated, one of the contributions of CUCKOOSWITCH is a high performance, memory efficient forwarding table design. This new data structure allows us to scale the forwarding table to billions of entries, which has never been achieved before. Hyper-Switch addresses a different aspect of scalability: scaling the aggregate throughput of inter-VM communication with the number of VM pairs. Like CUCKOOSWITCH, both of these virtual switch architectures applied multi-staging packet processing, batching, and prefetching strategies to optimize the packet I/O throughput. However, CUCKOOSWITCH further extends these ideas to forwarding table lookup, which significantly mitigates the high DRAM access latency.

FPGA-based Routers Gorilla [23] generates FPGA-based network processors from a C-like language. For 1 M entries, it demonstrates up to 200 Mpps of IPv4 or MPLS lookups using an extremely high-end FPGA. Though the systems are not directly comparable, CUCKOOSWITCH can handle about 350 Mops of lookups (Figure 5), which suggests that our techniques *may* (and we emphasize that this is only a possibility) have lessons to contribute for future FPGA-based designs.

Cuckoo Hashing Cuckoo hashing [26] is a recent technique for building hash tables with high space utilization while guaranteeing O(1) expected insertion and retrieval time. SILT [24] uses *partial-key cuckoo hashing* to provide fast, memory-efficient buffer storage for newly inserted key-value items. MemC3 [14] proposes *optimistic concurrent cuckoo hashing*, which eliminates the limitation of the maximum table size of partial-key cuckoo hashing and improves performance by allowing concurrent access to the hash table by multiple readers and a single writer. CUCKOOSWITCH further improves upon optimistic concurrent cuckoo hashing by adding the optimizations described in Section 3.

7. CONCLUSION

This paper describes CUCKOOSWITCH, a new FIB design for software-based Ethernet switches. Using a highly-optimized I/O engine (Intel DPDK) plus a high performance, memory efficient lookup table based on cuckoo hashing, CUCKOOSWITCH can saturate 80 Gbps for packet sizes of 192 bytes or larger—even for FIB sizes as large as one billion entries. For minimum-size packets, CUCKOOSWITCH can achieve the full 64-byte packet forwarding throughput of our experiment platform with one *billion* FIB entries; furthermore, CUCKOOSWITCH allows in-place updates with modest performance impact. We believe that the large table sizes and fast forwarding provided by CUCKOOSWITCH both raise the bar for the performance of software-based switches and contribute to our understanding of the feasibility of network architectures that require or benefit from large flat forwarding tables.

8. ACKNOWLEDGMENTS

We gratefully acknowledge the CoNEXT reviewers and our shepherd, Ken Calvert, for their feedback and suggestions; and Google, Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC), and the National Science Foundation under awards CCF-0964474 and CNS-1040801 for their financial support of this research.

9. REFERENCES

[1] libcuckoo. https://github.com/efficient/libcuckoo.

[2] Mellanox sx1016 64-port 10gbe switch system. http://www.mellanox.com/page/products_dyn?product_family=125.

[3] Intel Data Plane Development Kit (Intel DPDK) Overview, .

[4] Intel 64 and IA-32 Architectures Optimization Reference Manual, . http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf.

[5] Read-Copy Update, . http://en.wikipedia.org/wiki/Read-copy-update.

[6] Intel Threading Building Blocks. http://threadingbuildingblocks.org/, 2011.

[7] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity, data center network architecture. In *Proc. ACM SIGCOMM*, Aug. 2008.

[8] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet Protocol (AIP). In *Proc. ACM SIGCOMM*, Aug. 2008.

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

[10] M. Caesar, T. Condie, J. Kannan, K. Lakshimarayanan, I. Stoica, and S. Shenker. ROFL: Routing on Flat Labels. In *Proc. ACM SIGCOMM*, Aug. 2006.

[11] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using bloom filters. In *Proc. ACM SIGCOMM*, Aug. 2003.

[12] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2009.

[13] U. Erlingsson, M. Manasse, and F. Mcsherry. A cool and practical alternative to traditional hash tables. In *Seventh Whorkshop on Distributed Data and Structures (WDAS'2006)*, pages 1–6, 2006.

[14] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proc. 10th USENIX NSDI*, Apr. 2013.

[15] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area Web cache sharing protocol. In *Proc. ACM SIGCOMM*, pages 254–265, Sept. 1998.

[16] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *Proc. ACM SIGCOMM*, Aug. 2009.

[17] D. Han, A. Anand, F. Dogar, B. Li, H. Lim, M. Machado, A. Mukundan, W. Wu, A. Akella, D. G. Andersen, J. W. Byers, S. Seshan, and P. Steenkiste. XIA: Efficient support for evolvable internetworking. In *Proc. 9th USENIX NSDI*, Apr. 2012.

[18] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *Proc. ACM SIGCOMM*, Aug. 2010.

[19] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 0123705916, 9780123705914.

[20] Intel Data Plane Development Kit (Intel DPDK). http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/packet-processing-is-enhanced-with-software-from-intel-dpdk.html, 2013.

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

[22] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: A scalable ethernet architecture for large enterprises. In *Proc. ACM SIGCOMM*, Aug. 2008.

[23] M. Lavasani, L. Dennison, and D. Chiou. Compiling high throughput network processors. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, FPGA '12, 2012.

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

[25] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: A scalable fault-tolerant layer2 data center network fabric. In *Proc. ACM SIGCOMM*, Aug. 2009.

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

[27] K. K. Ram, A. L. Cox, M. Chadha, and S. Rixner. Hyper-Switch: A Scalable Software Virtual Switching Architecture. In *Proc. USENIX ATC 2013*, June 2013.

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

[29] L. Rizzo and G. Lettieri. Vale, a switched ethernet for virtual machines. In *Proc. CONEXT '12*, 2012.

[30] M. Yu, A. Fabrikant, and J. Rexford. BUFFALO: Bloom filter forwarding architecture for large organizations. In *Proc. CoNEXT*, Dec. 2009.