Scaling Up Clustered Network Appliances with ScaleBricks

Dong Zhou, Bin Fan, Hyeontaek Lim, David G. Andersen, Michael Kaminsky[†],
Michael Mitzenmacher[**], Ren Wang[†], Ajaypal Singh[‡]

Carnegie Mellon University, [†]Intel Labs, [**]Harvard University, [‡]Connectem, Inc.

ABSTRACT

This paper presents ScaleBricks, a 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. ScaleBricks applies a new, compact lookup structure to route packets directly to the appropriate handling node, without incurring the cost of multiple hops across the internal interconnect. Its lookup structure is many times smaller than the alternative approach of fully replicating a forwarding table onto all nodes. As a result, ScaleBricks is able to improve throughput and latency while simultaneously increasing the total number of flows that can be handled by such a cluster. This architecture is effective in practice: Used to optimize packet forwarding in an existing commercial LTE-to-Internet gateway, it increases the throughput of a four-node cluster by 23%, reduces latency by up to 10%, saves memory, and stores up to 5.7x more entries in the forwarding table.

CCS Concepts

•**Networks** → **Middle boxes / network appliances;**

Keywords

network function virtualization; scalability; hashing algorithms

1. INTRODUCTION

Many clustered network appliances require *deterministic* partitioning of a flat key space among a cluster of machines. 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 the next section, the cellular network-to-Internet gateway [1] in the core network of Long-Term Evolution (LTE).

In this paper, we explore a less-examined aspect of scalability for such clustered network appliances: can we create a design in which the forwarding table ("FIB" or Forwarding Information Base) 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? We ask this question because in a typical design, such as RouteBricks [13], adding another node to a cluster does not increase the total number of keys that the cluster can support; it increases only the total throughput and number of ports. In this paper, we explore a design that allows the FIB to continue to scale through 8, 16, or even 32 nodes, increasing the FIB capacity by up to 5.7x.

We focus on three properties of cluster scaling in this work:

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 3, 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 [22] requires multiple hops across the fabric.

The contribution of this paper is two-fold. First, we present the design, implementation, and theoretical underpinning of an architecture called ScaleBricks that achieves these goals (Section 3). The core of ScaleBricks is a new data structure SetSep that represents the mapping from keys to nodes in an extremely compact manner (Section 4). As a result, each *ingress* node is able to forward packets directly to the appropriate *handling* node without needing a full copy of the FIB at all nodes. This small global information table requires only

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.
Copyright is held by the owner/author(s).
SIGCOMM'15, August 17-21, 2015, London, United Kingdom
ACM 978-1-4503-3542-3/15/08.
http://dx.doi.org/10.1145/2785956.2787503

$O(\log N)$ bits per key, where N is the number of nodes in the cluster, enabling positive—though sublinear—FIB scaling for realistically-sized clusters. We believe this data structure will prove useful for other applications outside ScaleBricks.

Second, we use ScaleBricks to improve the performance of a commercial cellular LTE-to-Internet gateway, described in more detail in the following section. Our prototype shows that a 4-node ScaleBricks cluster can nearly quadruple the number of keys managed compared with single node solutions, while simultaneously improving packet forwarding throughput by approximately 23% and cutting latency up to 10% (Section 6).

2. DRIVING APPLICATION: CELLULAR NETWORK-TO-INTERNET GATEWAY

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 [1]; Figure 1a 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 [18].

- 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).[1]
- 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.
- 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 paper, we focus our improvements on a commercially-available software EPC stack from Connectem [10]. 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 1b 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 in this paper 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.

3. DESIGN OVERVIEW

In this section, we explain the design choices for ScaleBricks and compare those choices to representative alternative designs to illustrate why we made those choices. 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.
- *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

[1]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.

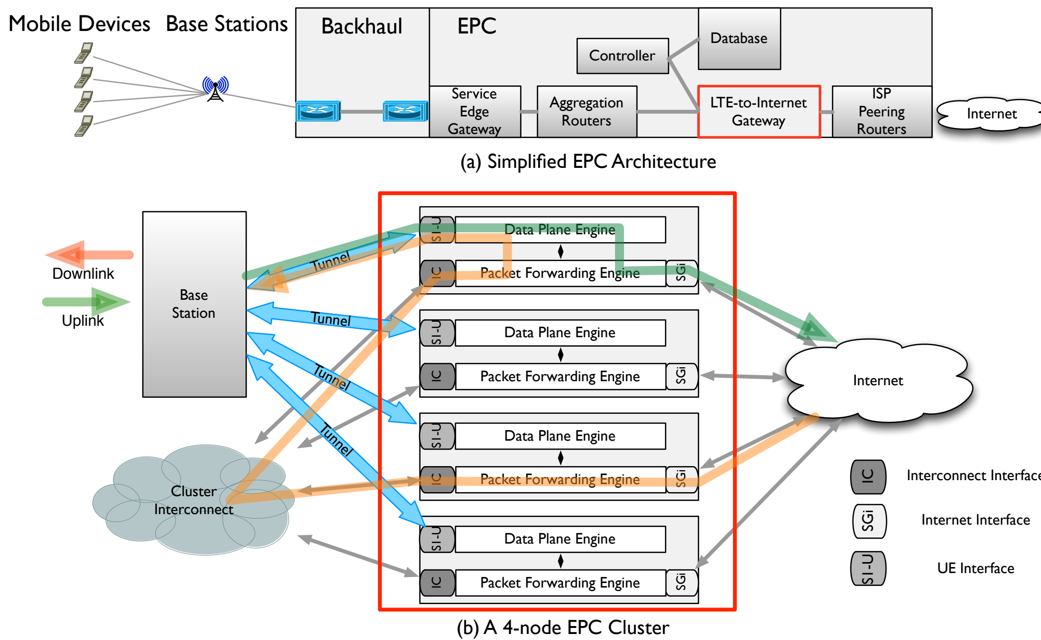
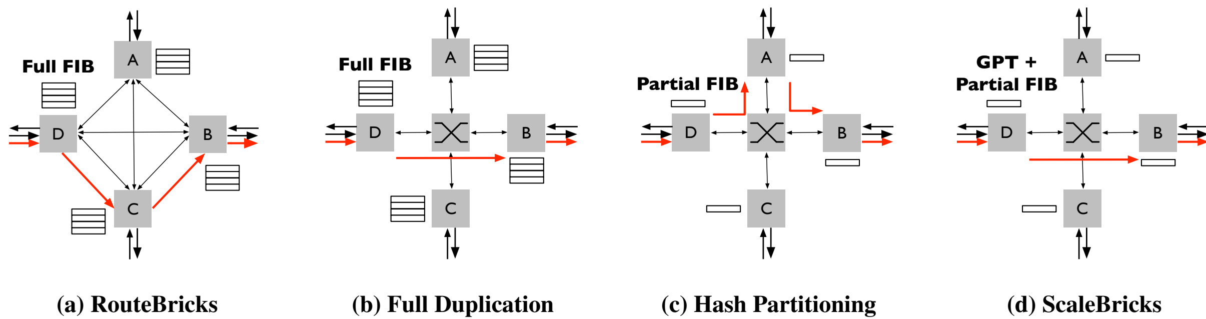


Figure 1: (a) Simplified Evolved Packet Core (EPC) architecture and (b) a 4-node EPC cluster



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

Figure 2: Packet forwarding in different FIB architectures

unknown key, or more than one lookup node if the FIB entry has been replicated to more than one node.

3.1 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 [13]. In such systems, the servers are connected in a full mesh or a butterfly-like topology, as shown in Figure 2a. On top of this topology, load-balancing routing algorithms—e.g., Valiant Load Balancing (VLB) [32]—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× 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 2b–2d). 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 paper. 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.

3.2 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 2b). 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 2c). 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 paper, we present a design that forwards directly from ingress to handling nodes, but uses substantially less memory than a typical fully-replicated FIB (Figure 2d). 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 prior work in space-efficient, high-performance hash tables for read-intensive workloads [34]. We omit in-depth discussion here because we used the prior design nearly unchanged, only extending 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 paper, we extend prior work by Fan et al. [15] 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 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 4.5 provides further details about RIB partitioning.

4. 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 [15] to provide memory-speed lookups on billions of entries, while requiring only a few bits per entry. As discussed in the previous section, 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.

4.1 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 4.3.

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

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

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 [23]. 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

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

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.

4.2 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.

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 3a 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× when $m = 6$, and when $m \geq 12$, it needs (on average) fewer than 100 trials, i.e., it is 100× faster.

Figure 3b 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×.

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 paper.) 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.

4.3 Representing Non-Boolean Values

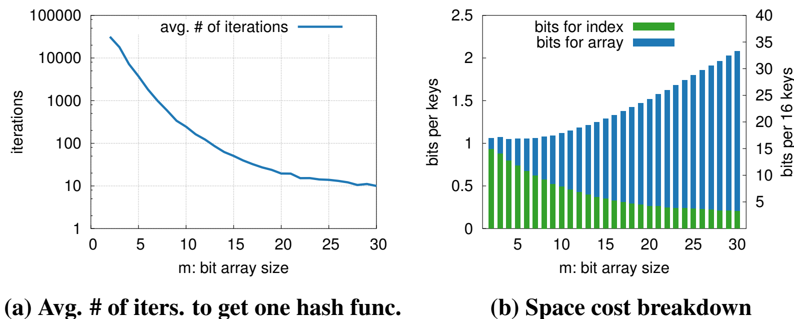
We have described the construction for two disjoint subsets (two possible values). For $V > 2$ different subsets, a trivial extension is to look for *one hash function* that outputs the right value in $\{1, \ldots, V\}$ for each key. However, this approach is not practical because it must try $O(V^n)$ 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 4 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 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 suit-



(a) Avg. # of iters. to get one hash func. **(b) Space cost breakdown**

Figure 3: Space vs. time, as the function of bit array size m

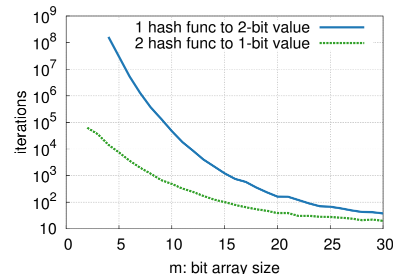


Figure 4: One hash func. vs. multiple hash func.

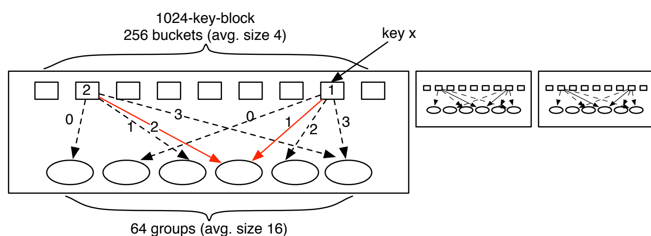


Figure 5: Illustration of two-level hashing

able 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 [29]. 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 5 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 5. 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.

4.5 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 safe in-place updates, techniques analogous to those proposed in CuckooSwitch [34] and MemC3 [14] could be applied, although we have not designed such a mechanism yet.

5. IMPLEMENTATION / OPTIMIZATIONS

ScaleBricks is implemented in C. It uses Intel's Data Plane Development Kit (DPDK) for x86 platforms [21] as a fast user-space packet I/O engine.

5.1 Global Partition Table using SetSep

SetSep uses several optimizations to improve performance.

Efficient Use of Memory Bandwidth and CPU Cycles Sequentially issuing memory fetches one at a time cannot saturate the bandwidth between CPU and memory. ScaleBricks instead uses *batched lookups* and *prefetching* [27, 34] (Algorithm 1). Each lookup request is divided into three stages and a subsequent stage accesses a memory location determined by its previous stage. Immediately fetching these memory locations would stall CPU pipelines for many cycles while waiting for the load instruction to complete. Instead, our algorithm first issues a *prefetch* instruction for a set of addresses, which causes the CPU to start loading the data from these addresses into cache. Then, at the beginning of the next stage, it executes normal reads at those prefetched addresses. These loads are then likely to hit in L1/L2 cache and thus complete much faster. Prefetching significantly improves efficiency: Section 6 shows that with appropriate batch sizes, this optimization improves microbenchmark lookup throughput of SetSep by up to 1.8×.

Algorithm 1: Batched SetSep lookup with prefetching

```
BatchedLookup(keys[1..n])
begin
    for i ← 1 to n do
        bucketID[i] ← keys[i]'s bucket ID
        prefetch(bucketIDToGroupID[bucketID[i]])
    for i ← 1 to n do
        groupID[i] ← bucketIDToGroupID[bucketID[i]]
        prefetch(groupInfoArray[groupID[i]])
    for i ← 1 to n do
        groupInfo ← groupInfoArray[groupID[i]]
        values[i] ← LookupSingleKey(groupInfo, keys[i])
    return values[1..n]
```

Hardware Accelerated Construction Constructing SetSep is amenable to parallelization: Each group of keys can be computed independently, and within a group, the computation of the hash function across the sixteen keys can be parallelized using SIMD. In this work, we only explore using multiple hardware threads across groups, which provides sufficiently fast construction for our application. We plan to evaluate the SIMD and GPU-derived speedups in future work.

5.2 Partial FIB using Cuckoo Hashing

Each node in the cluster has a slice of the FIB to provide an exact mapping from keys to application-specific data. In ScaleBricks, this table is implemented using concurrent cuckoo hashing [34], which achieves high occupancy and line-rate lookup performance for read-intensive workloads.

CuckooSwitch [34] used a concurrent cuckoo hash table to build a FIB that maps MAC addresses to output ports. That prior work was optimized to fetch the entire key-value pair in a single cache line read and thus stored values adjacent to their keys. Our target application of ScaleBricks, however, requires storing arbitrarily large application-specific data about each key (instead of a single output port as in CuckooSwitch). We therefore apply the following optimization.

When the table is initialized at run-time, the value size is fixed for all entries based on the application requirements. We assign each slot in the cuckoo hash table a logical "slot number." Instead of storing key/value pairs in an interleaved form, we create a separate *value array* in which the k-th element is the value associated with the k-th slot in the hash table. To lookup the value of a key, we simply index into the value array at the position corresponding to the key's slot number in the hash table. When moving a key from one slot to another during insertion, we need to move the value as well.

The apparent drawback of this approach is an extra memory read during lookup. In practice, however, as we will show in the evaluation, this extra memory read has minimal impact on the lookup throughput.

6. EVALUATION

Our evaluation addresses three questions:

1. How fast can we construct and lookup with SetSep?
2. How does moving to ScaleBricks improve the throughput of the Packet Forwarding Engine of the LTE Evolved Packet Core? What limitations does it add?
3. How does ScaleBricks scale with the number of servers?

We omit the evaluation of the cuckoo hashing-based FIB; prior work has demonstrated that the table is fast enough to serve over 300 million lookups per second [34].

6.1 Micro-Benchmark: SetSep

This section presents micro-benchmark results for SetSep construction and lookup performance on modern hardware.

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.

6.1.1 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 1 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 paper*. Its speed, $\frac{1}{2}$ million keys per second per core, is adequate for the read-intensive workloads we target.

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.

6.1.2 Lookup

Figure 7 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 as discussed in Section 5, 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 *dynamic batching* policy from CuckooSwitch [34]: 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.

6.2 Macro-Benchmark: ScaleBricks Cluster

As a concrete application, we optimized the Packet Forwarding Engine of Connectem's EPC stack by migrating it to ScaleBricks. Importantly, the stack already used the Intel DPDK, so structural changes were small. For context, the Packet Forwarding Engine typically runs on three of the cores

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

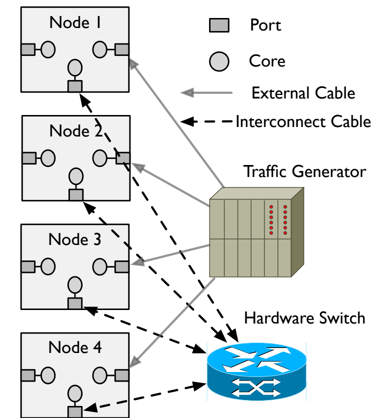


Figure 6: Cluster setup

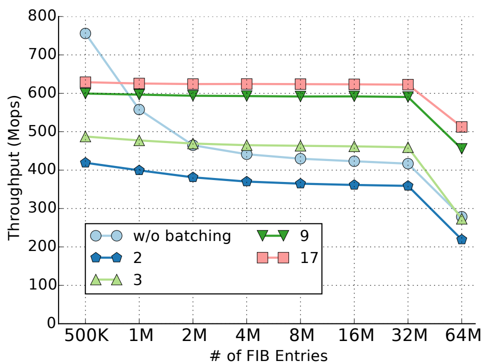


Figure 7: Local lookup throughput of SetSep (GPT)

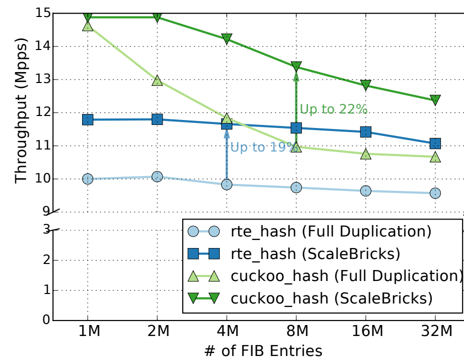


Figure 8: Single node packet forwarding throughput using 30 MiB L3 cache

in each node in the EPC cluster. The initial system was balanced to handle smaller numbers of flows, but the total system throughput drops when the number of flows (and thus the size of the FIB) grows larger. Thus, while we focus here on just the PFE throughput, improvements to the PFE do improve the total system throughput.

We measure PFE performance using 4 dual-socket servers, each with two Intel Xeon E5-2697 v2 CPUs running at 2.70 GHz, each with a 30 MiB L3 cache. Each machine has 128 GiB DDR3 RAM and two dual-port Intel 82599ES 10GbE NICs.

Each server uses three of its four 10Gb ports: one as the interface to the Internet, one as the interface to the base stations, and the other as the interconnect interface. Each port is attached to one core on CPU socket 0, using three cores in total. For all the experiments, we pre-populate the system with a number of static tunnels. As discussed in Section 2, only downstream (Internet to mobile device) packets require inter-cluster forwarding; therefore, the core assigned to handle the interface connected to the base stations is not used in our experiments. We simulate the downstream traffic using a Spirent SPT-N11U Ethernet testing platform [31]. Figure 6 depicts the configuration for the benchmark.

The forwarding engine originally implemented its FIB using a chaining hash table, the performance of which drops dramatically as the number of tunnels increases. To evaluate the benefits of ScaleBricks, we replace the original implementation with two alternative hash table designs—DPDK's `rte_hash` and our extended cuckoo hash table. Figure 8 shows the single node packet forwarding throughput using these hash tables, with and without ScaleBricks. Without ScaleBricks means full duplication, as depicted in Figure 2b.

Both the hash table and use of the SetSep GPT improve throughput measurably for the PFE. Even though our extended cuckoo hash table requires one additional memory read compared to the original design [34], it improves throughput by 50% over the DPDK's `rte_hash`. More key to this work, ScaleBricks improves the single node throughput by up to 20% and 22% within systems using `rte_hash` and extended cuckoo hash table, respectively. Two major factors contribute to this performance improvement. First, reducing the number of entries hosted by each node means smaller hash tables. In this experiment, hash table size is reduced by up to 75%. The smaller table allows many more of the table entries to fit in L3 cache, substantially increasing througput. Second, without ScaleBricks, all the packets coming from the traffic gener-

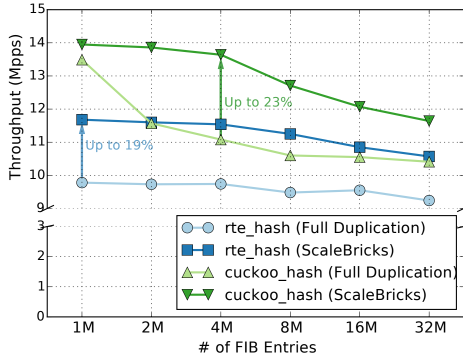


Figure 9: Single node packet forwarding throughput using 15 MiB L3 cache

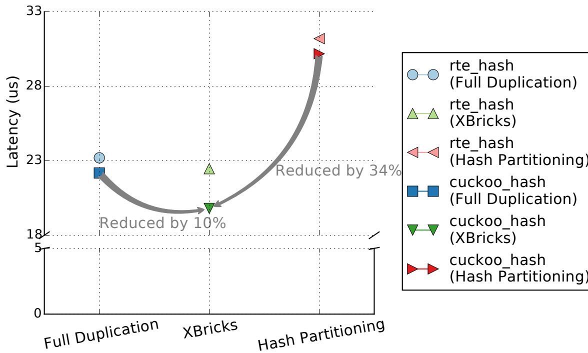


Figure 10: End-to-end latency of different approaches

ator are looked up by the core handling that port. We refer to this core as the "external core." The core processing the traffic received over the internal switch (the "internal core"), however, is mostly idle. In ScaleBricks, the external core on each server performs the global partition table lookup for all the packets, plus the hash table lookup for only those packets belonging to flows that are handled by that same server. The load is therefore more balanced, as the internal cores also perform a hash lookup, but in a smaller table. These two effects combine to improve the throughput and core utilization substantially. Although alternative designs might make better use of the idle cycles by multiplexing packet I/O and hash lookups on the internal cores, such designs—if they exist—are likely complex and introduce non-trivial overhead to switch between internal and external functionality on the internal core.

Throughput with Smaller Cache The EPC forwarding engine shares the CPU cache with other applications. To evaluate the throughput of ScaleBricks under such circumstances, we launch a *bubble* thread on a separate core to consume half of the L3 cache. Figure 9 shows the performance of different hash tables, with and without ScaleBricks, when there is only 15MiB of L3 cache available. Comparing with the results shown in Figure 8, the throughput of all tables drop with the reduced cache, but the relative benefits of ScaleBricks remain.

Latency We measure the end-to-end packet forwarding latency of six different approaches using Spirent SPT-N11U's RFC 2544 [6] benchmark tool. We create 1 M static tunnels for the latency test. Two interesting observations stand out from the average latency results reported in Figure 10. First, compared to the baseline, ScaleBricks reduces the average latency by up to 10%. We believe that ScaleBricks is able to service more of the lookups from cache because of the smaller table size, thereby reducing the memory access latency. Second, compared with the hash partitioning, the latency of ScaleBricks is lower by up to 34%. This matches our expectation. In summary, ScaleBricks improves the end-to-end latency via faster memory access and/or eliminating the extra inter-cluster hop.

Update Rate We measure the update rate of ScaleBricks as follows: a single CPU core can handle 60 K updates/sec. Using a decentralized update protocol allows ScaleBricks to distribute updates to all the nodes in a cluster. In a 4-node ScaleBricks cluster, using one dedicated core on each server provides an aggregated rate of 240 K updates/sec. Because the update is essentially parallelizable, by adding more CPU cores, we can achieve higher update rate if necessary.

6.3 Scalability of ScaleBricks

Compared to naive FIB duplication, ScaleBricks provides a practical leap in scaling by using a compact representation for the information that must be replicated on all nodes. To put its contributions in perspective, it is useful to compare to both FIB duplication and to two-hop FIB partitioning. Recall that FIB duplication and ScaleBricks require only a single hop across the internal switch fabric. Two-hop FIB partitioning, in contrast, incurs higher forwarding cost, but achieves true linear FIB scaling, which ScaleBricks does not.

Although SetSep is compact, its size (i.e., bits per entry) does increase slowly with the number of nodes in the cluster. This creates a scaling tension. At first, ScaleBricks scales almost linearly: for n nodes, each must store only $\frac{F}{n}$ of the F total FIB entries. But those FIB entries are large—perhaps 64 or 128 bits each. In contrast, the GPT must store all F entries on each node, but using only $F \log n$ bits. At first, $\log n$ is very small, and so the GPT is much smaller than the original FIB. But as n increases and more entries are added to the FIB, the GPT size begins to grow, and after 32 nodes, adding more servers actually decreases the total number of FIB entries that can be handled. Analytically, the total number of 64-bit FIB entries that can be stored in an n-node cluster scales as $Mn/(64+(0.5+1.5\log n)n)$, where M is the memory capacity per node. As a result, a ScaleBricks cluster can scale up to handle 5.7 times more FIB entries compared with a cluster using full FIB duplication.

Assuming each server uses 16 MiB of memory, Figure 11 shows analytically the total forwarding entries enabled using full FIB duplication, ScaleBricks, and hash partitioning, for clusters from 1 to 32 servers. ScaleBricks scales better when FIB entries are larger than 64 bits because the total number of FIB entries as well as the size of GPT will decrease.

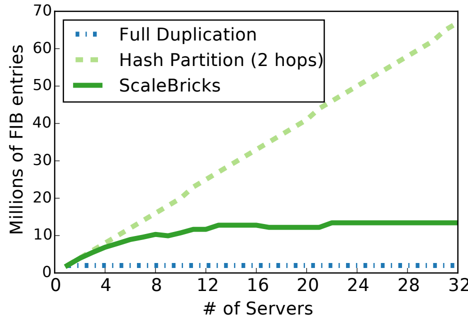


Figure 11: # of FIB entries with different # of servers

As both intuition suggests and the figure shows, traditional FIB duplication does not scale: The ensemble supports only as many FIB entries as a single node does. In contrast, hash partitioning of the FIB scales linearly, but at an extra forwarding cost. In ScaleBricks, the number of FIB entries that can be handled scales nearly linearly for a small number of nodes.

7. DISCUSSION

Skewed Forwarding Table Distribution ScaleBricks assumes the assignment of keys to handling nodes is not under its control; in the case of EPC, a separate controller assigns flows to handling nodes. If the assignment is skewed, some nodes must handle a disproportionately larger number of flows. In such a case, hash partitioning provides linear scalability (at the cost of one more hop) because it evenly distributes the FIB across intermediate lookup/indirect nodes. ScaleBricks, however, uses a combined lookup/handling node, so the FIB partitioning is skewed according to the handling node of each entry. In the ideal case, when each node has the same number of FIB entries, ScaleBricks scales well to up to 8 servers, as we have shown in the evaluation. However, when the distribution of the FIB entries is skewed, ScaleBricks can no longer achieve this scalability. The tradeoff between scalability and latency is fundamental, and ScaleBricks achieves near-optimal scalability with minimal latency (switching hops).

Isolation of Failure In general, ScaleBricks exhibits better failure tolerance properties than hash-partitioned clusters by providing failure isolation—when a server fails, the network communication from/to other servers can continue. This isolation comes from the fate sharing between each server and partial FIB hosted by itself. In a hash-partitioned cluster, however, one failing node could cause forwarding errors for the keys for which the node serves as a lookup node, even if these keys are not handled by the failing node itself.

More Applications ScaleBricks helps improve the performance of stateful, clustered network appliances which assign flows to their specific handling nodes without being able to control the assignment. We demonstrated the usefulness of ScaleBricks by using the LTE-to-Internet gateway as a driving application, and we are currently seeking for more applications that can benefit from ScaleBricks.

8. RELATED WORK

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 [30] or d-left hashing [28] 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 [12, 17, 16], Fredman and Komlós [16], 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 [26] is most similar to ours. Our approach uses both less space and fewer memory accesses.

Perfect hashing data structures can be compressed; for example, both ECT [25], as used in SILT [24], and CHD [3] use fewer than 2.5 bits per key to store the index.[2] However, these schemes must also store the value associated with each key. Nor do these compressed implementations of perfect hashing provide sufficient lookup throughput. ECT is optimized for indexing data in external storage such as SSDs, and its lookup latency is one to seven microseconds per lookup; CHD is faster, but remains several times slower than our solution.

Bloom Filters [4] are a compact probabilistic data structure used to represent a set of elements for set-membership tests, with many applications [7]. They achieve high space efficiency by allowing false positives.

Bloom filters and variants have been proposed for set separation. For example, BUFFALO [33] 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

[2] The information theoretical lower bound for minimal perfect hashing is approximately 1.44 bits per key.

(such as counting Bloom filters). Bloomier filters [9] 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 [5] similarly provide an alternative structure based on multiple choice hashing for efficiently representing partial functions. Our approach is again more scalable than these approaches.

Fast Software Routers and Switches RouteBricks [13] demonstrated that a commodity server can forward 64-byte IPv4 packets at 19 Mpps by batching packet I/O and parallelizing packet processing on modern multi-core CPUs and multi-queue NICs. PacketShader [20] exploits the parallelism and memory bandwidth of GPUs to provide fast IPv4 and IPv6 forwarding. ScaleBricks similarly exploits modern CPUs by performing batched table lookups that make more efficient use of memory bandwidth and reduce CPU cycles.

Scaling Forwarding Tables Bloom filters [4] have been used to improve memory-efficiency in packet forwarding engines. Dharmapurikar et al. [11] 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.

CuckooSwitch [34] scales the forwarding table of a *single node*. ScaleBricks adopts CuckooSwitch's per-node FIB design, but it could use any efficient FIB. The contribution of ScaleBricks is, instead, the use of SetSep to build a global partition table that is divided across switch cluster nodes and helps forward incoming packets to the handling node in a single hop.

Flat Address Routing Flat addresses enable simple network topology and easy manageability of enterprise and datacenter networks (e.g., SEATTLE [22]) as well as straightforward support for wide-area mobility (e.g., ROFL [8], AIP [2], XIA [19]). ScaleBricks provides a new, scalable implementation option for such flat designs.

9. CONCLUSION

ScaleBricks is a new mechanism for helping to "scale up" clustered network applications. Its core contribution is the design and implementation of a new data structure, SetSep, for compactly storing the mapping from keys, such as flow IDs or flat addresses, to values, such as the node ID that should handle that flow. To make this structure practical, ScaleBricks provides efficient mechanisms for partitioning the full forwarding state around a cluster and constructing and updating its SetSep-based global partitioning table. SetSep requires only 3.5 bits/key to store a mapping from arbitrary keys to 2-bit values, and provides extremely fast lookups. ScaleBricks is an effective technique for practical applications, and moving to it improved the packet forwarding throughput of a 4-node LTE-to-packet network gateway by 23% and cut its latency up to 10%.

Acknowledgments

This work was supported in part by NSF grants CNS-1345305, CNS-1314721, IIS-0964473, CNS-1228598, and CCF-1320231, and by the Intel Science and Technology Center for Cloud Computing.

10. REFERENCES

[1] 3GPP. The Evolved Packet Core. http://www.3gpp.org/technologies/keywords-acronyms/100-the-evolved-packet-core.

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

[3] D. Belazzougui, F. Botelho, and M. Dietzfelbinger. Hash, displace, and compress. In *Proceedings of the 17th European Symposium on Algorithms*, ESA '09, pages 682–693, 2009.

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

[5] F. Bonomi, M. Mitzenmacher, and R. Panigrahy. Beyond Bloom filters: From approximate membership checks to approximate state machines. In *Proc. ACM SIGCOMM*, Pisa, Italy, Aug. 2006.

[6] S. Bradner and J. McQuaid. *Benchmarking Methodology for Network Interconnect Devices*. Internet Engineering Task Force, Mar. 1999. RFC 2544.

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

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

[9] B. Chazelle, J. Kilian, R. Rubinfeld, A. Tal, and O. Boy. The Bloomier filter: An efficient data structure for static support lookup tables. In *Proceedings of SODA*, pages 30–39, 2004.

[10] Connectem. Connectem Inc. http://www.connectem.net/.

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

[12] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. M. Auf Der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[13] 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)*, Big Sky, MT, Oct. 2009.

[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*, Lombard, IL, Apr. 2013.

[15] B. Fan, D. Zhou, H. Lim, M. Kaminsky, and D. G. Andersen. When cycles are cheap, some tables can be huge. In *Proc. HotOS XIV*, Santa Ana Pueblo, NM, May 2013.

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

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

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

[19] 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*, San Jose, CA, Apr. 2012.

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

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

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

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

[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)*, Cascais, Portugal, Oct. 2011.

[25] H. Lim, D. G. Andersen, and M. Kaminsky. Practical batch-updatable external hashing with sorting. In *Proc. Meeting on Algorithm Engineering and Experiments (ALENEX)*, Jan. 2013.

[26] Y. Lu, B. Prabhakar, and F. Bonomi. Perfect hashing for network applications. In *Information Theory, 2006 IEEE International Symposium on*, pages 2774–2778. IEEE, 2006.

[27] Y. Mao, C. Cutler, and R. Morris. Optimizing ram-latency dominated applications. APSys '13, 2013.

[28] 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.

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

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

[31] Spirent. Spirent SPT-N11U. http://www.spirent.com/sitecore/content/Home/Ethernet_Testing/Platforms/11U_Chassis.

[32] 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.

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

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