Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing

Alex D. Breslow* Dong Ping Zhang Joseph L. Greathouse Nuwan Jayasena Dean M. Tullsen
AMD Research *AMD Research* *AMD Research* *AMD Research* *UC San Diego*

Abstract

Hash tables are important data structures that lie at the heart of important applications such as key-value stores and relational databases. Typically bucketized cuckoo hash tables (BCHTs) are used because they provide high-throughput lookups and load factors that exceed 95%. Unfortunately, this performance comes at the cost of reduced memory access efficiency. Positive lookups (key is in the table) and negative lookups (where it is not) on average access 1.5 and 2.0 buckets, respectively, which results in 50 to 100% more table-containing cache lines to be accessed than should be minimally necessary.

To reduce these surplus accesses, this paper presents the Horton table, a revamped BCHT that reduces the expected cost of positive and negative lookups to fewer than 1.18 and 1.06 buckets, respectively, while still achieving load factors of 95%. The key innovation is remap entries, small in-bucket records that allow (1) more elements to be hashed using a single, primary hash function, (2) items that overflow buckets to be tracked and rehashed with one of many alternate functions while maintaining a worst-case lookup cost of 2 buckets, and (3) shortening the vast majority of negative searches to 1 bucket access. With these advancements, Horton tables outperform BCHTs by 17% to 89%.

1 Introduction

Hash tables are fundamental data structures that are ubiquitous in high-performance and big-data applications such as in-memory relational databases [12, 17, 40] and key-value stores [20, 24]. Typically these workloads are read-heavy [4, 62]: the hash table is built once and is seldom modified in comparison to total accesses. A hash table that is particularly suited to this behavior is a bucketized cuckoo hash table (BCHT), a type of open-addressed hash table.[1] BCHTs group their cells into buckets: associative blocks of two to eight slots, with each slot capable of storing a single element.

When inserting an element, BCHTs typically select between one of two independent hash functions, each of which maps the key-value pair, call it KV, to a different *candidate bucket*. If one candidate has a free slot, KV is inserted. In the case where neither has spare slots, BCHTs resort to *cuckoo hashing*, a technique that resolves collisions by evicting and rehashing elements when too many elements vie for the same bucket. In this case, to make room for KV, the algorithm selects an element, call it KV', from one of KV's candidate buckets, and replaces it with KV. KV' is then subsequently rehashed to its alternate candidate using the remaining hash function. If the alternate bucket for KV' is full, KV' evicts yet another element and the process repeats until the final displaced element is relocated to a free slot.

Although these relocations may appear to incur large performance overheads, prior work demonstrates that most elements are inserted without displacing others and, accordingly, that BCHTs trade only a modest increase in average insertion and deletion time in exchange for high-throughput lookups and load factors that often exceed 95% with greater than 99% probability [19], a vast improvement over the majority of other techniques [60, 64].

BCHTs, due to this space efficiency and unparalleled throughput, have enabled recent performance breakthroughs in relational databases and key-value stores on server processors [20, 60, 64] as well as on accelerators such as GPUs [71], the Xeon Phi [15, 60], and the Cell processor [34, 64]. However, although BCHTs are higher-performing than other open addressing schemes [60, 64], we find that as the performance of modern computing systems becomes increasingly constrained by memory bandwidth [1, 11, 52, 70], they too suffer from a number of inefficiencies that originate from how data is fetched when satisfying queries.

Carefully coordinating table accesses is integral to throughput in hash tables. Because of the inherent randomness of hashing, accesses to hash tables often exhibit poor temporal and spatial locality, a property that causes hardware caches to become increasingly less effective as tables scale in size. For large tables, cache lines containing previously accessed hash table buckets are frequently evicted due to capacity misses before they are touched again, degrading performance and causing applications to become memory-bandwidth-bound once the table's working set cannot be cached on-chip. Given these concerns, techniques that reduce accesses to additional cache lines in the table prove invaluable when optimizing hash table performance and motivate the need to identify and address the data movement inefficiencies that are prevalent in BCHTs.

Consider a BCHT that uses two independent hash functions to map each element to one of two candidate buckets. To load balance buckets and attain high load factors, recent work on BCHT-based key-value stores inserts each element into the candidate bucket with the least load [20, 71], which means that we expect half of the elements to be hashed by each function. Consequently, on positive lookups, where the queried key is in the table, 1.5 buckets are expected to be examined. Half of

*This author is also a PhD student at UC San Diego. Send correspondence regarding the work to abreslow@cs.ucsd.edu.

[1] Under open addressing, an element may be placed in more than one location in the table. Collisions are resolved by relocating elements within the table rather than spilling to table-external storage.

the items can be retrieved by examining a single bucket, and the other half require accessing both. For negative lookups, where the queried key is not in the table, 2 lookups are necessary. Given that the minimum number of buckets that might need to be searched (for both positive and negative lookups) is 1, this leaves a very significant opportunity for improvement.

To this end, this paper presents Horton tables,[2] a carefully retrofitted bucketized cuckoo hash table, which trades a small amount of space for achieving positive and negative lookups that touch close to 1 bucket apiece. Our scheme introduces **remap entries**, small and succinct in-bucket records that enable (1) the tracking of past hashing decisions, (2) the use of many hash functions for little to no cost, and (3) most lookups, negative and positive alike, to be satisfied with a single bucket access and at most 2. Instead of giving equal weight to each hash function, which leads to frequent fetching of unnecessary buckets, we employ a single **primary hash function** that is used for the vast majority of elements in the table. By inducing such heavy skew, most lookups can be satisfied by accessing only a single bucket. To permit this biasing, we use several secondary hash functions (7 in our evaluations) to rehash elements to alternate buckets when their preferred bucket lacks sufficient capacity to directly store them. Rather than forgetting our choice of secondary hash function for remapped elements, we convert one of the slots in each bucket that overflows into a **remap entry array** that encodes which hash function was used to remap each of the overflow items. It is this ability to track all remapped items at low cost, both in terms of time and space, that permits the optimizations that give Horton tables their performance edge over the prior state-of-the-art.

To achieve this low cost, instead of storing an explicit tag or fingerprint (a succinct hash representation of the remapped object) as is done in cuckoo filters [21] and other work [8, 13], we instead employ implicit tags, where the index into the remap entry array is a tag computed by a hash function H_{tag} on the key. This space optimization permits all buckets to use at most 64 bits of remap entries in our implementation while recording all remappings, even for high load factors and tables with billions of elements. As a further optimization, we only convert the last slot of each bucket into a remap entry array when necessary. For buckets that do not overflow, they remain as standard buckets with full capacity, which permits load factors that exceed 90 and 95 percent for 4- and 8-slot buckets, respectively.

Our main contributions are as follows:

- We develop and evaluate Horton tables and demonstrate speed improvements of 17 to 89% on graphics processors (GPUs). Although our algorithm is not specific to GPUs, GPUs represent the most efficient platform for the current state of the art, and thus it is important to demonstrate the effectiveness of Horton tables on the same platform.
- We present algorithms for insertions, deletions, and lookups on Horton tables.
- We conduct a detailed analysis of Horton tables by deriving and empirically validating models for their expected data movement and storage overheads.
- We investigate additional optimizations for insertions and deletions that further reduce data movement when using multiple hash functions, reclaiming remap entries once their remapped elements are deleted, even when they are shared by two or more table elements.

[2]Named for elephants' remarkable recall powers [18].

This paper is organized as follows: In Section 2 we elaborate on the interplay between BCHTs and single instruction multiple data (SIMD) processors, in Section 3 we describe BCHTs, in Section 4 we provide a high-level overview of Horton tables, in Section 5 we describe the lookup, insertion, and deletion algorithms for Horton tables, and then in Section 6 we present our models for Horton tables that include the cost of insertions, deletions, and remap entry storage. Section 7 covers our experimental methodology, and Section 8 contains our performance and model validation results. Related work is described in Section 9 and Section 10 concludes.

2 The Role of SIMD

In key places in this paper, we make references to SIMD and GPU architectures. Although not necessary for understanding our innovations, these references are present due to SIMD's importance for high-throughput implementations of in-memory hash tables and BCHTs in particular.

Recent work in high-performance databases that leverages BCHTs has shown that SIMD implementations of BCHTs, as well as larger data processing primitives, are often more than 3× faster than the highest performing implementations that use scalar instructions alone [6, 44, 60]. These SIMD implementations enable billions of lookups per second to be satisfied on a single server [60, 71], an unimaginable feat only a few years ago. At the same time, SIMD implementations of BCHTs are faster than hash tables that use other open addressing methods [60, 64]. Such implementations are growing in importance because both CPUs and GPUs require writing SIMD implementations to maximize performance-per-watt and reduce total cost of ownership.

For these reasons, we focus on a SIMD implementation of BCHTs as a starting point and endeavor to show that all further optimizations provided by Horton tables not only have theoretical models that justify their performance edge (Section 6) but also that practical SIMD implementations deliver the performance benefits that the theory projects (Section 8).[3]

3 Background on BCHTs

In this section, we describe in detail BCHTs and the associated performance considerations that arise out of their interaction with the memory hierarchy of today's systems.

To begin, we illustrate two common scenarios that are triggered by the insertion of two different key-value pairs KV_1 and KV_2 into the hash table, as shown in Figure 1. Numbered rows correspond to buckets, and groups of

[3]For a primer on SIMD and GPGPU architectures, we recommend these excellent references: H&P (Ch. 4) [30] and Keckler et al. [39].

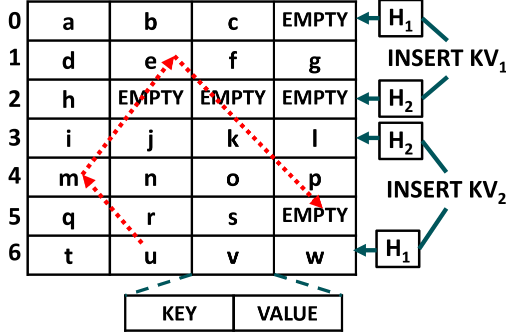


Figure 1: *Inserting items* $\mathbf{KV_1}$ *and* $\mathbf{KV_2}$ *into a BCHT*

four cells within a row to slots. In this example, H_1 and H_2 correspond to the two independent hash functions that are used to hash each item to two candidate buckets (0 and 2 for KV_1, 3 and 6 for KV_2). Both H_1 and H_2 are a viable choice for KV_1 because both buckets 0 and 2 have free slots. Deciding which to insert into is at the discretion of the algorithm (see Section 3.3 for more details).

For KV_2, both H_1 and H_2 hash it to buckets that are already full, which is resolved by evicting one of the elements (in this case u), and relocating it and other conflicting elements in succession using a different hash function until a free slot is found.[4] So e moves to the empty position in bucket 5, m to e's old position, u to m's old position, and KV_2 to u's old position. Li, et al. demonstrated that an efficient way to perform these displacements is to first conduct a breadth-first search starting from the candidate buckets and then begin moving elements only once a path to a free slot is discovered [47].

3.1 State-of-Practice Implementation

A number of important parameters affect the performance of BCHTs. In particular, the number of hash functions (***f***) and the number of slots per bucket (***S***) impact the achievable load factor (i.e., how full a table can be filled) as well as the expected lookup time. A hash table with more slots per bucket can more readily accommodate collisions without requiring a rehashing mechanism (such as cuckoo hashing) and can increase the table's load factor. Most implementations use four [20,60,64] or eight [71] slots per bucket, which typically leads to one to two buckets per hardware cache line. Using more slots comes at the cost of more key comparisons on lookups, since the requested element could be in any of the slots.

Increasing f, the number of hash functions, allows a key-value pair to be mapped to more buckets, as each hash function maps the item to one of f different buckets. This improved flexibility when placing an item permits the hash table to achieve a higher load factor. However, on lookups, more buckets need to be searched because the element could be in more locations. In practice, $f = 2$ is used most often because it permits sufficient flexibility in where keys are mapped without suffering from having to search too many buckets [20, 54, 63].

BCHTs are primarily designed for fast lookups. The get operation on any key requires examining the contents of at most f buckets. Because buckets have a fixed width, the lookup operation on a bucket can be unrolled and efficiently vectorized. These traits allow efficient SIMD implementations of BCHTs that achieve lookup rates superior to linear probing and double-hashing-based schemes on past and present server architectures [60, 64] and accelerators such as Intel's Xeon Phi [60].

3.2 Memory Traffic on Lookups

Like prior work, we divide lookups into two categories: (1) positive, where the lookup succeeds because the key is found in the hash table, and (2) negative where the lookup fails because the key is not in the table.

Prior work diverges on the precise method of accessing the hash table during lookups. The first method, which we term **latency-optimized**, always accesses all buckets where an item may be found [47, 64]. Another technique, which we call **bandwidth-optimized**, avoids fetching additional buckets where feasible [20, 71].

Given f independent hash functions where each of them maps each item to one of f candidate buckets, the latency-optimized approach always touches f buckets while the bandwidth-optimized one touches, on average, $(f+1)/2$ buckets on positive lookups and f buckets on negative lookups. For our work, we compare against the **bandwidth-optimized** approach, as it moves less data on average. Reducing such data movement is a greater performance concern on throughput-oriented architectures such as GPUs, since memory latency is often very effectively hidden on these devices [23]. Thus, we compare against the more bandwidth-oriented variant of BCHT, which searches 1.5 buckets instead of 2 (or more, if there are more hash functions) for positive lookups.

3.3 Insertion Policy and Lookups

Unlike the latency-optimized scheme, the bandwidth-optimized algorithm searches the buckets in some defined order. If an item is found before searching the last of the f candidate buckets, then we can reduce the lookup's data movement cost by skipping the search of the remaining candidate buckets. Thus if f is 2, and we call the first hash function H_1 and the second H_2, then the average lookup cost across all inserted keys is `1 * (fraction of keys that use` H_1`) + 2 * (fraction of keys that use` H_2`)`. Therefore, the insertion algorithm's policy on when to use H_1 or H_2 affects the lookup cost.

Given that hash tables almost always exhibit poor temporal and spatial locality, hash tables with working sets that are too large to fit in caches are bandwidth-bound and are quite sensitive to the comparatively limited off-chip bandwidth. In the ideal case, we therefore want to touch as few buckets as possible. If we can strongly favor using H_1 over H_2 during insertions, we can reduce the percentage of buckets that are fetched that do not contain the queried key, which reduces per-lookup bandwidth requirements as well as cache pollution, both of which improve lookup throughput.

Existing high-throughput, bandwidth-optimized BCHT implementations [20, 71] attempt to load-balance buckets on insertion by examining all buckets the key can map to and placing elements into the buckets with the most free slots. As an example, in Figure 1, KV_1 would be placed in the bucket hashed to by H_2. The

[4] So in this example, elements on the chain that were originally hashed with H_1 would be rehashed using H_2 and vice versa.

intuition behind this load balancing is that it both reduces the occurrence of cuckoo rehashing, which is commonly implemented with comparatively expensive atomic swap operations, and increases the anticipated load factor. Given this policy, H_1 and H_2 are both used with equal probability, which means that 1.5 buckets are searched on average for positive lookups. We refer to this approach as the **load-balancing baseline**.

An alternative approach is to insert items into the first candidate bucket that can house them. This technique reduces the positive lookup costs, since it favors the hash functions that are searched earlier. We refer to this as the **first-fit** heuristic. As an example, in Figure 1, KV_1 would be placed in the final slot of the top bucket of the table even though bucket 2 has more free slots. This policy means that items can be located with fewer memory accesses, on average, by avoiding fetching candidate buckets that follow a successful match. When the table is not particularly full, most elements can be inserted and retrieved by accessing a single table cache line.

Although prior work mentions this approach [19, 64], they do not evaluate its performance impact on lookups. Ross demonstrated its ability to reduce the cost of inserts but does not present data on its effect on lookups, instead opting to compare his latency-optimized lookup algorithm that always fetches f buckets to other open addressing methods and chaining [64]. Erlingsson et al. use the first-fit heuristic, but their results focus on the number of buckets accessed on insertions and feasible load factors for differing values of f and S (number of slots per bucket) and not the heuristic's impact on lookup performance [19]. For the sake of completeness, we evaluate both the load-balancing and first-fit heuristics in Section 8.

One consequence of using first-fit is that, because it less evenly balances the load across buckets, once the table approaches capacity, a few outliers repeatedly hash to buckets that are already full, necessitating long, cuckoo displacement chains when only 2 hash functions are used. Whereas we were able to implement the insertion routine for the load-balancing baseline and attain high load factors by relocating at most one or two elements, the first-fit heuristic prompted us to implement a serial version of the BFS approach described by Li et al. [47] because finding long displacement chains becomes necessary for filling the table to a comparable level. One solution to reducing these long chains is to use more hash functions. However, for BCHTs, this increases both the average- and worst-case lookup costs because each item can now appear in more distinct buckets. In the sections that follow, we demonstrate that these tradeoffs can be effectively circumvented with our technique and that there are additional benefits such as fast negative lookups.

4 Horton Tables

Horton tables are an extension of bucketized cuckoo hash tables that largely resolve the data movement issues of their predecessor when accessing buckets during lookups. They use two types of buckets (Figure 2): one is an unmodified BCHT bucket (**Type A**) and the other bucket flavor (**Type B**) contains additional in-bucket metadata to track elements that primarily hash to the bucket but have to be remapped due to insufficient capacity.

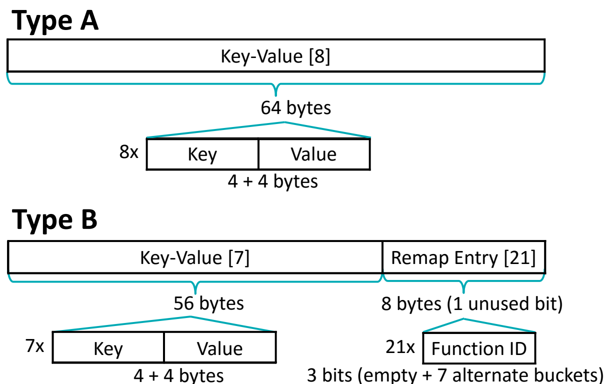


Figure 2: *Horton tables use 2 bucket variants: Type A (an unmodified BCHT bucket) and Type B (converts final slot into remap entries)*

All buckets begin as Type A and transition to Type B once they overflow, enabling the aforementioned tracking of displaced elements. This ability to track all remapped items at low cost, both in terms of time and space, permits the optimizations that give Horton tables their performance edge over the prior state of the art.

Horton tables use $H_{primary}$, the **primary hash function**, to hash the vast majority of elements so that most lookups only require one bucket access. When inserting an item $KV = (K,V)$, it is only when the bucket at index $H_{primary}(K)$ cannot directly store KV that the item uses one of several **secondary hash functions** to remap the item. We term the bucket at index $H_{primary}(K)$ the **primary bucket** and buckets referenced by secondary hash functions **secondary buckets**. Additionally, **primary items** are key-value pairs that are directly housed in the bucket referenced by the primary hash function $H_{primary}$, and **secondary items** are those that have been remapped. There is no correlation between the bucket's type and its primacy; Type A and B buckets can simultaneously house both primary and secondary elements.

Type B buckets convert the final slot of Type A buckets into a **remap entry array**, a vector of k-bit[5] elements known as **remap entries** that encode the secondary hash function ID used to rehash items that cannot be accommodated in their primary bucket. Remap entries can take on one of 2^k different values, 0 for encoding an unused remap entry, and 1 to $2^k - 1$ for encoding which of the secondary hash functions R_1 to R_{2^k-1} was used to remap the items. To determine the index at which a remapped element's remap entry is stored, a tag hash function known as H_{tag} is computed on the element's key which maps to a spot in the remap entry array.

Remap entries are a crucial innovation of Horton tables, as they permit all secondary items to be tracked so that at most one primary and sometimes one secondary hash function need to be evaluated during table lookups regardless of whether (1) the lookup is positive or negative and (2) how many hash functions are used to rehash secondary items. At the same time, their storage is compactly allocated directly within the hash table bucket that overflows, boosting the locality of their accesses while still permitting high table load factors.

[5] k could range from 1 to the width of a key-value pair in bits, but we have found $k = 3$ to be a good design point.

8	5	EMPTY	EMPTY
33	EMPTY	15	2
35	18	22	23
EMPTY	EMPTY	4	37
⋮			
EMPTY	EMPTY	EMPTY	7

8	5	EMPTY	EMPTY
33	7	15 R_7	2
35	18	22	7 E 5 2
EMPTY	EMPTY	23 R_2	37
⋮			
EMPTY	EMPTY	EMPTY	4

R_5

Figure 3: *Comparison of a bucketized cuckoo hash table (L) and a Horton table (R). E = empty remapy entry.*

With Horton tables, most lookups only require touching a single bucket and a small minority touch two. At the same time remap entries typically use at most several percent of the table's capacity, leaving sufficient free space for Horton tables to achieve comparable load factors to BCHTs.

Figure 2 shows the Type A and Type B bucket designs given 4-byte keys and values and 8-slot buckets. The bucket type is encoded in one bit of each bucket. For Type A buckets, this costs us a bit from just one of the value fields (now 31 bits). For Type B buckets, we encode 21 3-bit remap entries into a 64-bit slot, so we have a bit to spare already. If we have a value that requires all 32 bits in the last slot of a Type A bucket, we can move it to another slot in this bucket or remap to another bucket.

Because Type B buckets can house fewer elements than Type A buckets, Type A buckets are used as much as possible. It is only when a bucket has insufficient capacity to house all primary items that hash to it that it is converted into a Type B bucket, a process known as **promotion**. To guarantee that elements can be found quickly, whenever possible we enforce that primary elements are not displaced by secondary items. This policy ensures both that more buckets remain Type A buckets and that more items are primary elements.

4.1 A Comparison with BCHTs

Figure 3 shows a high-level comparison of Horton tables with an $f = 2$, traditional BCHT that stores the same data. Buckets correspond to rows and slots to individual cells within each row. In the Horton table (right), each item maps to its primary bucket by default. Bucket 2 (zero indexing) has been **promoted** from Type A to Type B because its 4 slots are insufficient to directly house the 6 key-value pairs that $H_{primary}$ has hashed there: 35, 18, 22, 7, 23, and 4. Because there is insufficient space to house 7, 23, and 4 directly in Bucket 2, they are remapped with hash functions R_7, R_2, and R_5, respectively, and the function IDs are stored directly in the remap entry array at the indices referenced by $H_{tag}(7) = 0$, $H_{tag}(23) = 3$, and $H_{tag}(3) = 2$. If we contrast the Horton table and the associated cuckoo hash table, we find that, of the shown buckets, the Horton table has a lookup cost of 1 for elements 8, 5, 33, 15, 2, 35, 18, 22, and 37 and a lookup cost of 2 for 7, 23, and 4, which averages out to 1.25. By contrast the bucketized cuckoo hash table has an expected lookup cost of 1.5 [20, 71] or 2.0 [47, 64], depending on the implementation.

① Compute $\mathbf{H_{primary}(K)} \rightarrow \mathbf{x}$ ④ Compute tag hash **t** with $\mathbf{H_{tag}(K)} \rightarrow \mathbf{t}$
② Determine if **K** matches an item in bucket **x**
⑤ If remap entry at index **t** is empty, return **KNF** Else go to ⑥
③ If match found, return it. Else if **x.isTypeA()**, return **KNF** Else go to ④
K = b.key?
REA[t].isEmpty()?
1 5 3
0 14 20
K → $H_{primary}$ → x: a b c d e f g REA
K → R_3 → w: h i j k l m n REA
⑥ Compute the secondary hash function specified by the remap entry (**R_3**) and check if **K** matches an item in **w**
⑦ If there is match, return it. Else return **KNF**

Figure 4: *Horton table lookups. KNF and REA are abbreviations for key not found and remap entry array.*

5 Algorithms

In this section, we describe the algorithms that we use to look up, insert, and delete elements in Horton tables. We begin each subsection by detailing the functional components of the algorithms and then, where relevant, briefly outline how each algorithm can be efficiently implemented using SIMD instructions.

5.1 Lookup Operation

Our lookup algorithm (Figure 4) works as follows. ① Given a key K, we first compute $H_{primary}(K)$, which gives us the index of K's primary bucket. ② We next examine the first $S - isTypeB()$ slots of the bucket, where $isTypeB()$ returns 1 if Bucket x is Type B and 0 if it is Type A by checking the bucket's most significant bit. ③ If the key is found, we return the value. In the case that the key is not found and the bucket is Type A, then the element cannot appear in any other bucket, and so we can declare the key not found.

④ If, however, the bucket is Type B, then we must examine the remap entry array when the item is not found in the first $S - 1$ slots of K's primary bucket (Bucket x). We first compute $H_{tag}(K)$ to determine the index into the remap entry array (shown as $t = 14$ in the figure). ⑤ If the value at that slot is 0, which signifies empty, then we declare the key not found because the key cannot appear in any other bucket. However, if the remap entry is set, then ⑥ we evaluate the secondary function R_i specified by the remap entry (R_3 in Figure 4) on a combination of the primary bucket and remap entry index (see Section 5.3.4) to get the index of the secondary bucket (Bucket w). We then compare K to the first $S - isTypeB()$ elements of w. ⑦ If we get a match, then we return it. If we do not get a match, then because an element is never found outside of its primary bucket or the secondary bucket specified by its remap entry, then we declare the key not found. It cannot be anywhere else.

5.2 SIMD Implementation of Lookups

Our approach leverages bulk processing of lookups and takes a large array of keys as input and writes out a corresponding array of retrieved values as output. We implement a modified version of Zhang et al.'s lookup algorithm [71], which virtualizes the SIMD unit into groups of S lanes (akin to simple cores that co-execute the same instruction stream) and assigns each

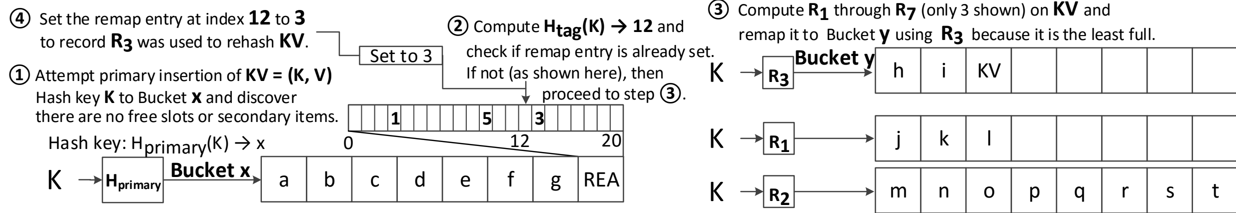


Figure 5: *Common execution path for secondary inserts. REA is an abbreviation of remap entry array.*

group a different bucket to process. When an element is found in the first $S - isTypeB()$ slots, we write the value out to an in-cache buffer. For the minority of lookups where more processing is necessary, e.g. computing the tag hash, indexing into the remap entry array, computing the secondary hash function, and searching the secondary bucket, we maintain a series of additional in-cache buffers where we enqueue work corresponding to these less frequent execution paths. When there is a SIMD unit's worth of work in a buffer, we dequeue that work and process it. Once a cache line worth of contiguous values have been retrieved from the hash table, we write those values back to memory in a single memory transaction.[6]

5.3 Insertion Operation

The primary goal of the insertion algorithm is to practically guarantee that lookups remain as fast as possible as the table's load factor increases. To accomplish this, we enforce at all levels the following guidelines:

1. Secondary items never displace primary items.
2. Primary items may displace both primary and secondary items.
3. When inserting into a full Type A bucket, only convert it to Type B if a secondary item in it cannot be remapped to another bucket to free up a slot.[7]

These guidelines ensure that as many buckets as possible remain Type A, which is important because converting a bucket from Type A to Type B can have a cascading effect: both the evicted element from the converted slot and the element that caused the conversion may map to other buckets and force them to convert to Type B as well. Further, Type B buckets house fewer elements, so they decrease the maximum load factor of the table and increase the expected lookup cost.

5.3.1 Primary Inserts

Given a key-value pair KV to insert into the table, if the primary bucket has a spare slot, then insertion can proceed by assigning that spare slot to KV. Spare slots can occur in Type A buckets as well as Type B buckets where a slot has been freed due to a deletion, assuming that Type B buckets do not atomically pull items back in from remap entries when slots become available. For the primary hash function, we use one of Jenkins' hash functions that operates on 32-bit inputs and produces 32-bit outputs [35]. The input to the function is the key, and we mod the output by the number of buckets to select a bucket to map the key to.

In the case where the bucket is full, we do not immediately attempt to insert KV into one of its secondary buckets but first search the bucket for secondary elements. If we find a secondary element KV' that can be remapped without displacing primary elements in other buckets, then we swap KV with KV' and rehash KV' to another bucket. Otherwise, we perform a secondary insert (see Sections 5.3.2, 5.3.3, and 5.3.4).

5.3.2 Secondary Inserts

We make a secondary insert when the item that we want to insert, $KV = (K,V)$, hashes to a bucket that is full and in which all items already stored in the bucket are primary. Most of the time, secondary inserts occur when an element's primary bucket has already been converted to Type B (see Section 5.3.3 and Figure 6 for the steps for converting from Type A to B); Figure 5 shows the most common execution path for a secondary insert.

① We first determine that a primary insert is not possible. ② We then compute the tag hash function on the key. If the remap entry at index $H_{tag}(K)$ is not set, we proceed to step ③. Otherwise, we follow the remap entry collision management scheme presented in Section 5.3.4 and Figure 7. ③ At this point, we need to find a free cell in which to place KV. We check each candidate bucket referenced by the secondary hash functions R_1 through R_7, and we place the remapped element in the bucket with least load, Bucket y in Figure 5. Alternatively, we could have placed KV into the candidate bucket with the first free slot – we chose the load-balancing approach because it reduced the prevalence of expensive cuckoo chains for relocating elements. ④ Lastly, we update the remap entry to indicate which secondary hash function was used. In this example, R_3 was used and H_{tag} on K evaluated to 12, so we write 3 in the 12th slot of the remap entry array of x, KV's primary bucket.

5.3.3 Conversion from Type A to Type B

Figure 6 shows the series of steps involved for converting a bucket from Type A to Type B. ① – ② If there are no secondary elements that can be displaced in the primary bucket, then the bucket evicts one of the items (h) to make room for the remap entry array, ③ – ⑤ rehashes it to another bucket, and ⑥ – ⑦ then proceeds by rehashing the element that triggered the conversion. As in the algorithm in Section 5.3.2, we attempt to relocate both items to their least loaded candidate buckets. ⑧ – ⑩ Once moved, the numerical identifier of each

[6] A simpler algorithm can be used when S is a multiple of the number of lanes, as all lanes within a SIMD unit process the same bucket.

[7] An item's primacy can be detected by evaluating $H_{primary}$ on its key. If the output matches the index where it is stored, then it is primary.

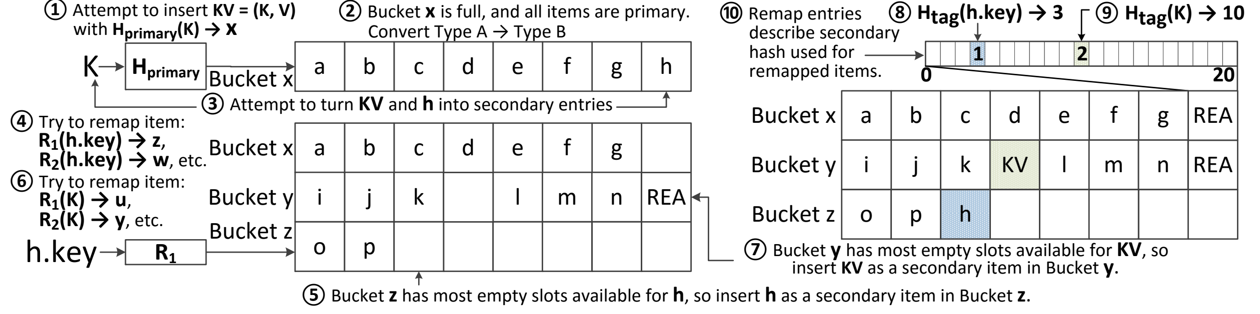


Figure 6: *Steps for converting from Type A to Type B. REA is an abbreviation of remap entry array.*

secondary hash function that remapped each of the two items (KV and h) is stored in the remap entry array of the primary bucket at the index described by H_{tag} of each key.

5.3.4 Remap Entry Collision Management

A major challenge of the remap entry approach is when two or more items that require remapping map to the same remap entry. Such collisions can be accommodated if all items that share a remap entry are made to use the same secondary hash function. However, if the shared secondary hash function takes the key as input, it will normally map each of the conflicting items to a different bucket. While this property poses no great challenge for lookups or insertions, it makes deletions of remap entries challenging because without recording that a remap entry is shared, we would not know whether another item is still referenced by it. Rather than associating a counter with each remap entry to count collisions as is done in counting Bloom filters [13, 22], we instead modify the secondary hash function so that items that share a remap entry map to the same secondary bucket. Since they share the same secondary bucket, we can check if it is safe to delete a remap entry on deletion of an element KV that the entry references by computing the primary hash function on each element in KV's secondary bucket. If none of the computed hashes reference KV's primary bucket for any of the elements that share KV's secondary bucket, then the remap entry can be deleted.

To guarantee that items that share remap entries hash to the same secondary bucket, we hash on a combination of the primary bucket index and the implicit tag as computed by $H_{tag}(key)$. Since this tuple uniquely identifies the location of each remap entry, we can create a one-to-one function from tuples to unique secondary hash function inputs, shown in Equation 1, where i is a number that uniquely identifies each secondary hash function and which ranges from 1 to $2^k - 1$ for k-bit remap entries (e.g. R_3 is the third secondary function out of 7 when k is 3), H_{L1} and H_{L2} are hash functions, k_{sec} is the secondary key derived from the tuple, and n is the number of remap entries per remap entry array. The uniqueness of these tuples as inputs is important for reducing collisions.

$$\begin{aligned} R_i(k_{sec}) &= (H_{L1}(k_{sec}) + H_{L2}(k_{sec}, i)) \ \% \ Total\ Buckets \\ where\ k_{sec}&(bucket\ index, tag) = bucket\ index * n + tag \end{aligned} \tag{1}$$

By modifying the characteristics of H_{L1} and H_{L2}, we are able to emulate different hashing schemes. We employ modified double hashing by using Jenkins' hash [35] for H_{L1} and Equation 2 for H_{L2} where KT is a table of 8 prime numbers. We found this approach preferable because it makes it inexpensive to compute all of the secondary hash functions, reduces clustering compared to implementing H_{L2} as linear probing from H_{L1}, and, as Mitzenmacher proved, there is no penalty to bucket load distribution versus fully random hashing [55].

$$H_{L2}(k_{sec}, i) = KT[k_{sec} \% 8] * i \tag{2}$$

Sometimes the secondary bucket cannot accommodate additional elements that share the remap entry array. If so, we swap the item that we want to insert with another element from its primary bucket that can be rehashed. Because both elements in the swap are primary elements, this swap does not adversely affect the lookup rate.

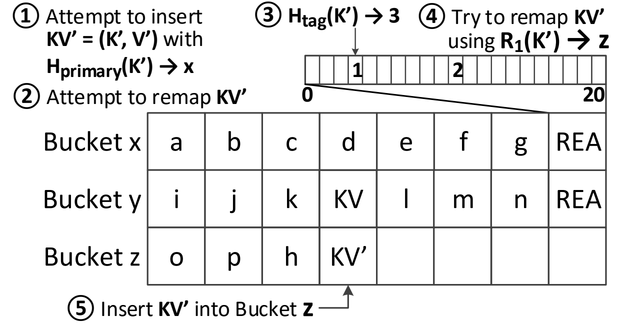


Figure 7: *Resolution of a remap entry collision*

Figure 7 presents a visual depiction of a remap entry collision during insertion that is resolved by having the new item map to the same bucket as the other items referenced by the remap entry. It continues from the example in Figure 6 and follows with inserting a new item KV'. ① When inserting KV', we first check for a free slot or an element that can be evicted from Bucket x because it is a secondary item when in x. ② If no such item exists, then we begin the process of remapping KV' to another bucket. ③ We first check the remap entry, and if it is set, ④ we proceed to the bucket it references, z in Figure 7. ⑤ We check for a free slot or a secondary item in z that can be displaced. If it is the former, we immediately insert KV'. Otherwise, we recursively evict elements until a free slot is found within a certain search tree height. Most of the time, this method works, but when it does not, we resort to swapping KV' with another element in its primary bucket that can be recursively remapped to a secondary bucket.

5.4 Deletion Operation

Deletions proceed by first calculating $H_{primary}$ on the key. If an item is found in the primary bucket with that key, that item is deleted. However, if it is not found in the primary bucket, and the bucket is Type B, then we check to see whether the remap entry is set. If it is, then we calculate the secondary bucket index and examine the secondary bucket. If an item with a matching key is found, then we delete that item. To determine whether we can delete the remap entry, we check to see if additional elements in the secondary bucket have the same primary bucket as the deleted element. If none do, we remove the remap entry.

5.4.1 Repatriation of Remapped Items

On deletions, slots that were previously occupied become free. In Type A buckets, there is no difference in terms of lookups regardless of how many slots are free. However, with Type B buckets, if a slot becomes free, that presents a performance opportunity to move a remapped item back into its primary bucket, reducing its lookup cost from 2 to 1 buckets. Similarly, if a Type B bucket has a combined total of fewer than $S+1$ items stored in its slots or remapped via its remap entries, it can be upgraded to a Type A bucket, which allows one more item to be stored and accessed with a single lookup in the hash table. Continual repatriation of items is necessary for workloads with many deletes to maximize lookup throughput and the achievable load factor. Determining when best to perform this repatriation, either via an eager or lazy heuristic, is future work.

6 Feasibility and Cost Analysis

In this section, we investigate the feasibility of using remap entries, the associated costs in terms of storage overhead, and the expected cost of both positive and negative hash table lookups.

6.1 Modeling Collisions

One of the most important considerations when constructing a Horton table is that each bucket should be able to track all items that initially hash to it using the primary hash function $H_{primary}$. In particular, given a hash table with B_T buckets and n inserted items, we want to be able to compute the expected number of buckets that have exactly x elements hash to them, for each value of x from 0 to n inclusive. By devising a model that captures this information, we can determine how many remap entries are necessary to facilitate the remapping and tracking of all secondary items that overflow their respective primary buckets.

If we assume that $H_{primary}$ is a permutation (i.e., it is invertible and the domain is the codomain), and that it maps elements to bins in a fashion that is largely indistinguishable from sampling a uniform random distribution, then given a selection of random keys and a given table size, we can precisely compute the expected number of buckets to which $H_{primary}$ maps exactly x key-value pairs by using a Poisson distribution based model [36]. The expected number of buckets with precisely x such ele-

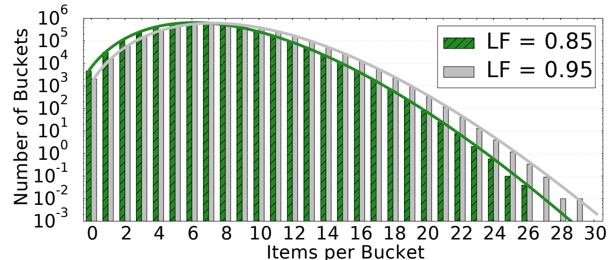


Figure 8: *Histogram of the number of buckets to which $H_{primary}$ assigns differing amounts of load in elements for two load factors. Curves represent instantiations of Equation 3 and bars correspond to simulation.*

ments, B_x, is given by Equation 3.

$$\begin{aligned} &B_x(\lambda,x) = Total\ Buckets * P(\lambda,x) \\ &\quad where\ P(\lambda,x) = \frac{e^{-\lambda}\lambda^x}{x!} \\ &\quad where\ \lambda = Load\ Factor * Slots\ Per\ Bucket \\ &\quad i.e.,\ \lambda = \frac{Elements\ Inserted}{Total\ Buckets} \end{aligned} \tag{3}$$

The parameter λ is the mean of the distribution. Given a load factor, the average number of items that map to a bucket is the product of the load factor and the slots per bucket. Figure 8 coplots the results of a bucketized hash table simulation with results predicted by the analytical model given a hash table with 2^{22} 8-slot buckets. In our simulation, we created n unique keys in the range [0, $2^{32}-1$] using a 32-bit Mersenne Twister pseudorandom number generator [51] and maintained a histogram of counts of buckets with differing numbers of collisions. We found little to no variation in results with different commonly utilized hash functions (e.g., CityHash, SpookyHash, Lookup3, Wang's Hash, and FNV [26, 35, 59]). Therefore, we show only the results using one of Jenkins' functions that maps 32-bit keys to 32-bit values. Figure 8 shows a close correlation between the simulation results and Equation 3 for two load factors. Bars correspond to simulation results and curves to Equation 3. In each case, the model very closely tracks the simulation.

A high-level conclusion of the model is that with billions of keys and 8-slot buckets, there is a non-trivial probability that a very small subset of buckets will have on the order of 30 keys hash to them. This analysis informs our decision to use 21 remap entries per remap entry array and also the need to allow multiple key-value pairs to share each remap entry in order to reduce the number of remap entries that are necessary per bucket.

6.2 Modeling Remap Entry Storage Costs

In our hash table design, each promoted bucket trades one slot for a series of remap entries. To understand the total cost of remap entries, we need to calculate what percentage of buckets are Type A and Type B, respectively. For any hash table with S slots per bucket, Type A buckets have no additional storage cost, and so they do not factor into the storage overhead. Type B buckets on the other hand convert one of their S slots, i.e. $1/S$ of their usable storage, into a series of remap entries. Thus the expected space used by remap entries O_{re}, on a scale of

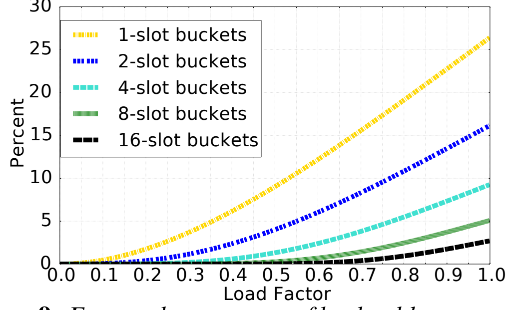


Figure 9: *Expected percentage of hash table storage that goes to remap entries as the load factor is varied*

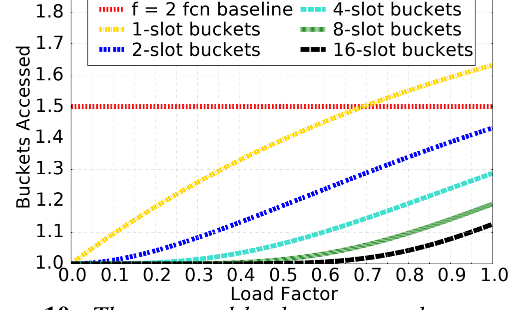


Figure 10: *The expected buckets accessed per positive lookup in a Horton table vs. a baseline BCHT that uses two hash functions*

0 (no remap entries) to 1 (entire table is remap entries), is the product of the fraction of Type B buckets and the consumed space $1/S$ (see Equation 4). For simplicity, we assume that each item that overflows a Type B bucket is remappable to a Type A bucket and that these remaps do not cause Type A buckets to become Type B buckets. This approximation is reasonable for two reasons. First, many hash functions can be used to remap items, and second, secondary items are evicted and hashed yet again, when feasible, if they prevent an item from being inserted with the primary hash function $H_{primary}$.

$$O_{re} = \frac{1}{S} \sum_{x=S+1}^{n} P(\lambda, x) \tag{4}$$

Figure 9 shows the expected percentage of hash table storage that goes to remap entries when varying the number of slots per bucket as well as the load factor. As the remap entries occupy space, the expected maximum load factor is strictly less than or equal to $1 - O_{re}$. We see that neither 1 slot nor 2 slots per bucket is a viable option if we want to achieve load factors exceeding 90%. Solving for the expected bound on the load factor, we find that 4-, 8-, and 16-slot hash tables are likely to achieve load factors that exceed 91, 95, and 96%, respectively, provided that the remaining space not consumed by remap entries can be almost entirely filled.

6.3 Modeling Lookups

The expected average cost of a positive lookup is dependent on the percentage of items that are first-level lookups, the percentage of items that are second-level lookups, and the associated cost of accessing remapped and non-remapped items. For a bucket with S slots, if $x > S$ elements map to that bucket, $x - S + 1$ elements will need to be remapped, as one of those slots now contains remap entries. In the case where $x \leq S$, no elements need to be remapped from that bucket. The fraction of items that require remapping, I_{remap}, is given by Equation 5, and the fraction that do not, $I_{primary}$, is given by Equation 6. As stated previously, lookups that use $H_{primary}$ require searching one bucket, and lookups that make use of remap entries require searching two. Using this intuition, we combine Equations 5 and 6 to generate the expected positive lookup cost given by Equation 7. Since $I_{primary}$ is a probability, and $1 - I_{remap}$ is equivalent to $I_{primary}$, we can simplify the positive lookup cost to 1 plus the expected fraction of lookups that are secondary. Intuitively, Equation 7 makes sense: 100% of lookups need to access the primary bucket. It is only when the item has been remapped that a second bucket needs to be searched.

$$I_{remap} = \frac{\sum_{x=S+1}^{n} (x - S + 1) * P(\lambda, x)}{\lambda} \tag{5}$$

$$I_{primary} = \frac{\sum_{x=1}^{S} (x) * P(\lambda, x) + \sum_{x=S+1}^{n} (S-1) * P(\lambda, x)}{\lambda} \tag{6}$$

$$\begin{aligned} Positive\ Lookup\ Cost &= I_{primary} + 2 I_{remap} \\ &= 1 + I_{remap} \end{aligned} \tag{7}$$

Figure 10 shows the expected positive lookup cost in buckets for 1-, 2-, 4-, 8-, and 16-slot bucket designs. Like before, buckets with more slots are better able to tolerate collisions. Therefore, as the number of slots per bucket increases for a fixed load factor, so does the ratio of Type A buckets to total buckets, which reduces the number of second-level lookups due to not needing to dereference a remap entry. In the 1- and 2-slot bucket cases, the benefit of remap entries is less pronounced but is still present. For the 1-slot case, there is a point at LF = 0.70 where we expect a baseline bucketized cuckoo hash table to touch fewer buckets. However, this scenario is not a fair comparison as, for a baseline BCHT with 1-slot buckets and two functions, the expected load factor does not reach 70%. To reach that threshold, many more hash functions would have to be used, increasing the number of buckets that must be searched. For the 4-, 8-, and 16-slot cases, we observe that the expected lookup cost is under 1.1 buckets for hash tables that are up to 60% full, and that even when approaching the maximum expected load factor, the expected lookup cost is less than 1.3 buckets. In the 8-slot and 16-slot cases, the expected costs at a load factor of 0.95 are 1.18 and 1.1 buckets, which represents a reduced cost of 21% and 27%, respectively, over the baseline.

The cost of a negative lookup follows similar reasoning. On a negative lookup, the secondary bucket is only searched on a false positive tag match in a remap entry. The expected proportion of negative lookups that exhibit tag aliasing, I_{alias}, is the product of the fraction of Type B buckets and the mean fraction of the tag space that is utilized per Type B bucket (Equation 8). In the implicit tag scheme, for a 64-bit remap entry array with 21 3-bit entries, the tag space is defined as the set $\{i \in \mathbb{N} | \ 0 \leq i \leq 20\}$ and has a tag space cardinality, call it C_{tag}, of 21. Alternatively, with explicit t-bit tags, C_{tag} would be 2^t minus any reserved values for designating

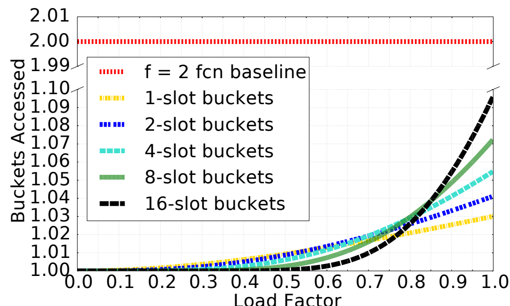


Figure 11: *The expected buckets accessed per negative lookup in a Horton table vs. a baseline BCHT that uses two hash functions*

states such as `empty`. For our model, we assume that there is a one-to-one mapping between remapped items and remap entries (i.e., each remap entry can only remap a single item). We further assume that conflicts where multiple items map to the same remap entry can be mitigated with high probability by swapping the element that would have mapped to an existing remap entry with an item stored in one of the slots that does not map to an existing remap entry, then rehashing the evicted element, and finally initializing the associated remap entry. These assumptions allow for at most $S-1+C_{tag}$ elements to be stored in or remapped from a Type B bucket.

$$I_{alias} = \frac{\sum_{x=S+1}^{S-1+C_{tag}} ((x - S + 1) * P(\lambda, x))}{C_{tag}} \quad (8)$$

$$\begin{aligned} Negative\ Lookup\ Cost &= I_{no\ alias} + 2I_{alias} \\ &= 1 + I_{alias} \end{aligned} \quad (9)$$

Like before, we can simplify Equation 9 by observing that all lookups need to search at least one bucket, and it is only on a tag alias that we search a second one. Because secondary buckets are in the minority and the number of remapped items per secondary bucket is often small relative to the tag space cardinality, the alias rate given in Equation 8 is often quite small, meaning that negative lookups have a cost close to 1.0 buckets.

In Figure 11, we plot the expected negative lookup cost for differing numbers of slots per bucket under progressively higher load factors with a tag space cardinality of 21. In contrast to positive lookups, adding more slots has a tradeoff. At low load factors, a table with more slots has a smaller proportion of elements that overflow and less Type B buckets, which reduces the alias rate. However, once the buckets become fuller, having more slots means that buckets have a greater propensity to have more items that need to be remapped, which increases the number of remap entries that are utilized. However, despite these trends, we observe that for 1-, 2-, 4-, 8-, and 16-slot buckets, aliases occur less than 8% of the time under feasible load factors, yielding an expected, worst-case, negative lookup cost of 1.08 buckets. Thus we expect Horton tables to reduce data movement on negative lookups by 46 to 50% versus an $f = 2$ BCHT.

7 Experimental Methodology

We run our experiments on a machine with a 4-core AMD A10-7850K with 32GB of DDR3 and an AMD Radeon™ R9-290X GPU with a peak memory bandwidth of 320 GB/s and 4GB of GDDR5. The L2 cache of the GPU is 1 MiB, and each of the 44 compute units has 16 KiB of L1 cache. Our system runs Ubuntu 14.04LTS with kernel version 3.16. Results for performance metrics are obtained by aggregating data from AMD's CodeXL, a publicly available tool that permits collecting high-level performance counters on GPUs when running OpenCL™ programs.

For the performance evaluation, we compare the lookup throughput of the load-balanced baseline, first-fit, and Horton tables. Our baseline implementation is most similar to Mega-KV [71], where the greatest difference is that we only use a single hash table rather than multiple independent partitions and use Jenkins' hash functions.

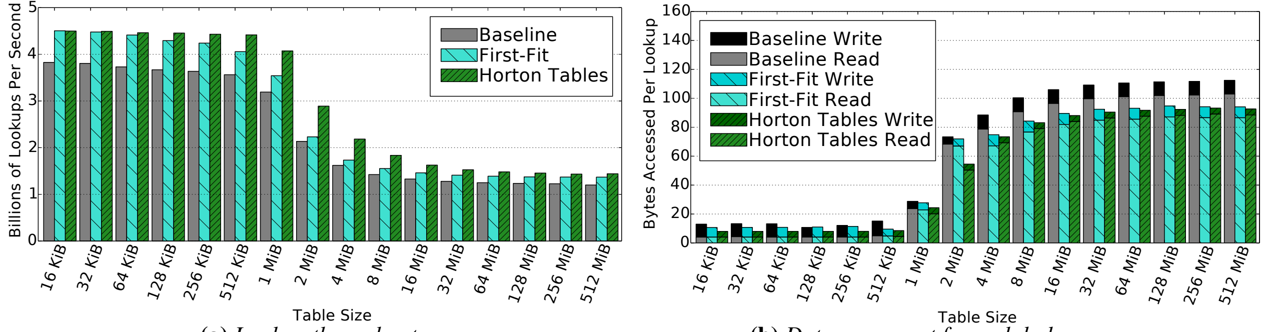
Insertions and deletions are implemented in C and run on the CPU. As our focus is on read-dominated workloads, we assume that insertion and deletion costs can largely be amortized and do not implement parallel versions. For each of the hash table variants, lookup routines are implemented in OpenCL [66] and run on the GPU, with each implementation independently autotuned for fairness of comparison. Toggled parameters include variable loop unrolling, the number of threads assigned to each compute unit, and the number of key-value pairs assigned to each group of threads to process. When presenting performance numbers, we do not count data transfer cost over PCIe because near-future integrated GPUs will have high-bandwidth, low-latency access to system memory without such overheads. This approach mirrors that of Polychroniou et al. [60] on the Xeon Phi [15].

As part of our evaluation, we validate the models presented in Section 6. We calculate the remap entry storage cost and lookup cost per item in terms of buckets by building the hash table and measuring its composition. Unless indicated elsewhere, we use 32-bit keys and values, 8-slot buckets and remap entry arrays with 21 3-bit entries. The probing array that we use for key-value lookups is 1 GiB in size. All evaluated hash tables are less than or equal to 512 MiB due to memory allocation limitations of the GPU; however, we confirmed that we were able to build heavily-loaded Horton tables with more than 2 billion elements without issue.

Keys and values for the table and the probing array are generated in the full 32-bit unsigned integer range using C++'s STL Mersenne Twister random integer generator that samples a pseudo-random uniform distribution [51]. We are careful to avoid inserting duplicate keys into the table, as that reduces the effective load factor by inducing entry overwrites rather than storing additional content. Since the probing array contains more keys than there are elements in the hash table, most keys appear multiple times in the probing array. We ensure that all such repeat keys appear far enough from one another such that they do not increase the temporal or spatial locality of accesses to the hash table. This approach is necessary to precisely lower bound the throughput of each algorithm for a given hash table size.

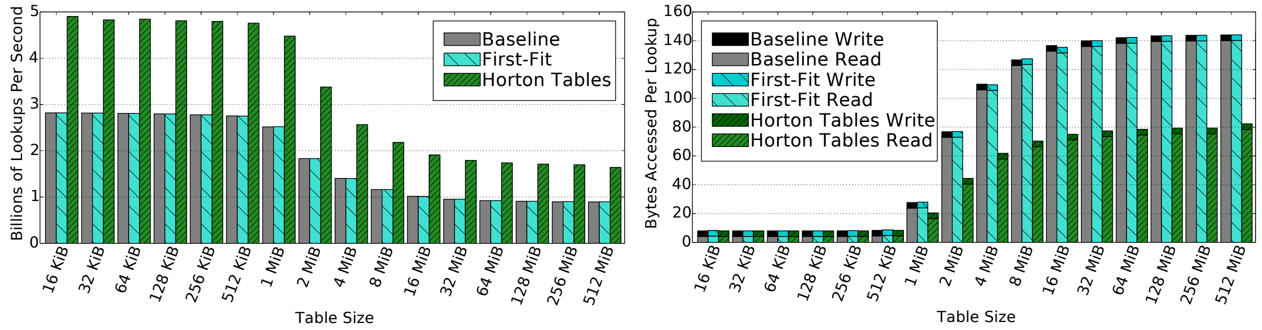
8 Results

In this section, we validate our models and present performance results. Figures 12a and 12b compare the lookup throughput and data movement (as seen by



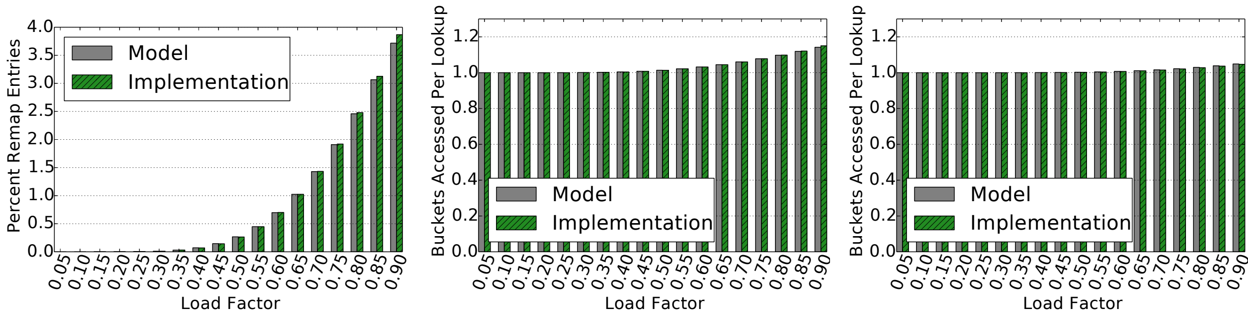
(a) *Lookup throughput* **(b)** *Data movement from global memory*

Figure 12: *Comparison of BCHTs with a Horton table (load factor = 0.9 and 100% of queried keys found in table)*



(a) *Lookup throughput* **(b)** *Data movement from global memory*

Figure 13: *Comparison of BCHTs with a Horton table (load factor = 0.9 and 0% of queried keys found in table)*



(a) *Remap entry storage cost* **(b)** *Positive lookup cost* **(c)** *Negative lookup cost*

Figure 14: *Validation of our models on an 8 MiB Horton table*

the global memory) between the load-balancing baseline (Section 3.3), BCHT with first-fit insert heuristic (Section 3.3) and Horton tables (Section 4) for tables from 16 KiB to 512 MiB in size. We see that Horton tables increase throughput over the baseline by 17 to 35% when all of the queried keys are in the table. In addition, they are faster than a first-fit approach, as Horton tables enforce primacy (Section 5.3, Guideline 1) on remapping of elements whereas first-fit does not. This discrepancy is most evident from 512 KiB to 8 MiB, where Horton tables are up to 29% faster than first-fit BCHTs.

These performance wins are directly proportional to the data movement saved. Initially, there is no sizeable difference in the measured data movement cost between the baseline, first-fit, and Horton tables, as the hash tables entirely fit in cache. Instead, the bottleneck to performance is the cache bandwidth. However, at around 1 MiB, the size at which the table's capacity is equal to the size of the last level cache (L2), the table is no longer fully cacheable in L2, and so it is at this point that the disparity in data movement between the three approaches becomes visible at the off-chip memory.

Figures 13a and 13b show the opposite extreme where none of the queried keys are in the table. In this case, Horton tables increase throughput by 73 to 89% over the baseline and first-fit methods because, unlike a BCHT, Horton tables can satisfy most negative searches with one bucket access. These results and those for positive lookups from Figures 12a and 12b align very closely with the reduction in data movement that we measured with performance counters. For a workload consisting entirely of positive lookups, baseline BCHTs access 30% more cache lines than Horton tables. At the opposite extreme, for a workload of entirely negative lookups, both first-fit and baseline BCHTs access 90% more hash table cache lines than Horton tables.

If we examine the total data movement, we find that both our BCHT and Horton table implementations move

an amount of data close to what our models project. At a load factor of 0.9, our model predicts 1.15 and 1.05 buckets accessed per positive and negative query, respectively. Since cache lines are 64 bytes, this cost corresponds to 74 and 67 bytes per query worth of data movement. On top of that, for each lookup query we have an additional 8 bytes of data movement for loading the 4-byte query key and 4 bytes for storing the retrieved value, which puts our total positive and negative lookup costs at 82 and 75 bytes, respectively. These numbers are within 10% of the total data movement that we observe in Figures 12b and 13b once the hash table is much larger than the size of the last-level cache. Similarly, we found that our models' data movement estimates for BCHTs were within similar margins of our empirical results.

Figures 14a, 14b, and 14c show that each of our models accurately capture the characteristics of our implementation. On average, our table requires fewer than 1.15 bucket lookups for positive lookups and fewer than 1.05 for negative lookups at a load factor of 0.9, and both have a cost of essentially 1.0 up to a load factor of 0.55. These results are a dramatic improvement over the current state of practice and validate the soundness of our algorithms to achieve high load factors without measurably compromising on the percentage of primary lookups.

9 Related Work

There has been a long evolution in hash tables. Two pieces of work that share commonality with our own are the cuckoo filter [21] and MemC3 [20]. MemC3 is a fast, concurrent alternative to Memcached [24] that uses a bucketized hash table to index data, with entries that consist of (1) tags and (2) pointers to objects that house the full key, value, and additional metadata. For them, the tag serves two primary functions: (1) to avoid polluting the cache with long keys on most negative lookups and (2) to allow variable-length keys. Tags are never used to avoid bringing additional buckets into cache. If the element is not found in the first bucket, the second bucket is always searched. Similarly, the cuckoo filter is also an $f = 2$ function, 4-slot bucketized cuckoo hash set that is designed to be an alternative to Bloom filters [10] that is cache friendly and supports deletions.

Another related work is Stadium Hashing [41]. Their focus is to have a fast hash table where the keys are stored on the GPU and the values in the CPU's memory. Unlike us they use a non-bucketized hash table with double hashing and prime capacity. They employ an auxiliary data structure known as a ticket board to filter requests between the CPU and GPU and also to permit concurrent put and get requests to the same table. Barber et al. use a similar bitmap structure to implement two compressed hash table variants [7].

The BCHT [19] combines cuckoo hashing and bucketization [25, 57, 58]. Another improvement to cuckoo hashing is by way of a stash—a small, software victim cache for evicted items [42].

Other forms of open addressing are also prevalent. Quadratic hashing, double hashing, Robin Hood hashing [14], and linear probing are other commonly used open addressing techniques [16]. Hopscotch hashing attempts to move keys to a preferred neighborhood of the table by displacing others [31]. It maintains a per-slot *hop information* field that is often several bits in length that tracks element displacements to nearby cells. By contrast, Horton tables only create remap entries when buckets exceed their baseline capacity.

Other work raises the throughput of concurrent hash tables by using lock free approaches [53, 67, 68] or fine-grain spinlocks [47]. Additional approaches attempt to fuse or use other data structures in tandem with hash tables to enable faster lookups [48, 61, 65].

In application, hash tables find themselves used in a wide of variety of data warehousing and processing applications. A number of in-memory key-value stores employ hash tables [20, 24, 32, 56, 71], and others accelerate key lookups by locating the hash table on the GPU [32, 33, 71]. Early GPU hash tables have been primarily developed for accelerating applications in databases, graphics and computer vision [2, 3, 27, 43, 45]. In in-memory databases, there has been significant effort spent on optimizing hash tables due to their use in hash join algorithms on CPUs [5, 6, 9, 12, 60], coupled CPU-GPU systems [28, 29, 38], and the Xeon Phi [37, 60].

This work on high-performance hash tables is complementary to additional research efforts that attempt to retool other indexes such as trees to take better advantage of system resources and new and emerging hardware [46, 49, 50, 69].

10 Conclusion

This paper presents the Horton table, an enhanced bucketized cuckoo hash table that achieves higher throughput by reducing the number of hardware cache lines that are accessed per lookup. It uses a single function to hash most elements and can therefore retrieve most items by accessing a single bucket, and thus a single cache line. Similarly, most negative lookups can also be satisfied by accessing one cache line. These low access costs are enabled by remap entries: sparingly allocated, in-bucket records that enable both cache and off-chip memory bandwidth to be used much more efficiently. Accordingly, Horton tables increase throughput for positive and negative lookups by as much as 35% and 89%, respectively. Best of all, these improvements do not sacrifice the other attractive traits of baseline BCHTs: worst-case lookup costs of 2 buckets and load factors that exceed 95%.

11 Acknowledgements

The authors thank the reviewers for their insightful comments and Geoff Kuenning, our shepherd, for his meticulous attention to detail and constructive feedback. We were very impressed with the level of rigor that was applied throughout the review and revision process. We also thank our peers at AMD Research for their comments during internal presentations of the work and Geoff Voelker for providing us invaluable feedback that elevated the paper's quality.

AMD, the AMD Arrow logo, AMD Radeon, and combinations thereof are trademarks of Advanced Micro Devices, Inc. OpenCL is a trademark of Apple, Inc. used by permission by Khronos. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND WOOD, D. A. DBMSs on a Modern Processor: Where Does Time Go? In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)* (1999).

[2] ALCANTARA, D. A., SHARF, A., ABBASINEJAD, F., SENGUPTA, S., MITZENMACHER, M., OWENS, J. D., AND AMENTA, N. Real-Time Parallel Hashing on the GPU. In *Proc. of the ACM SIGGRAPH Conf. and Exhibition on Computer Graphics and Interactive Techniques in Asia (SIGGRAPH Asia)* (2009).

[3] ALCANTARA, D. A., VOLKOV, V., SENGUPTA, S., MITZENMACHER, M., OWENS, J. D., AND AMENTA, N. Building an Efficient Hash Table on the GPU. In *GPU Computing Gems: Jade Edition*, W. W. Hwu, Ed. Morgan Kaufmann, 2011, ch. 4, pp. 39–54.

[4] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload Analysis of a Large-Scale Key-Value Store. In *ACM SIGMETRICS Performance Evaluation Review* (2012), vol. 40, ACM, pp. 53–64.

[5] BALKESEN, C., ALONSO, G., TEUBNER, J., AND ÖZSU, M. T. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proc. of the VLDB Endowment 7*, 1 (2013), 85–96.

[6] BALKESEN, C., TEUBNER, J., ALONSO, G., AND OZSU, M. T. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)* (2013).

[7] BARBER, R., LOHMAN, G., PANDIS, I., RAMAN, V., SIDLE, R., ATTALURI, G., CHAINANI, N., LIGHTSTONE, S., AND SHARPE, D. Memory-Efficient Hash Joins. *Proc. of the VLDB Endowment 8*, 4 (2014), 353–364.

[8] BENDER, M. A., FARACH-COLTON, M., JOHNSON, R., KRANER, R., KUSZMAUL, B. C., MEDJEDOVIC, D., MONTES, P., SHETTY, P., SPILLANE, R. P., AND ZADOK, E. Don't Thrash: How to Cache Your Hash on Flash. *Proc. of the VLDB Endowment 5*, 11 (2012), 1627–1637.

[9] BLANAS, S., LI, Y., AND PATEL, J. M. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)* (2011).

[10] BLOOM, B. H. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM 13*, 7 (1970), 422–426.

[11] BONCZ, P. A., KERSTEN, M. L., AND MANEGOLD, S. Breaking the Memory Wall in MonetDB. *Communications of the ACM 51*, 12 (2008), 77–85.

[12] BONCZ, P. A., MANEGOLD, S., AND KERSTEN, M. L. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)* (1999).

[13] BONOMI, F., MITZENMACHER, M., PANIGRAHY, R., SINGH, S., AND VARGHESE, G. An Improved Construction for Counting Bloom Filters. In *Proc. of the Annual European Symposium (ESA)* (2006).

[14] CELIS, P., LARSON, P.-Å., AND MUNRO, I. J. Robin Hood Hashing. In *Proc. of the IEEE Annual Symp. on Foundations of Computer Science (FOCS)* (1985).

[15] CHRYSOS, G., AND ENGINEER, S. P. Intel Xeon Phi Coprocessor (Codename Knights Corner). Presented at Hot Chips, 2012.

[16] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[17] DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M. R., AND WOOD, D. A. Implementation Techniques for Main Memory Database Systems. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)* (1984).

[18] DR. SEUSS. *Horton Hatches the Egg*. Random House, 1940.

[19] ERLINGSSON, U., MANASSE, M., AND MCSHERRY, F. A Cool and Practical Alternative to Traditional Hash Tables. In *Proc. of the Workshop on Distributed Data and Structures (WDAS)* (2006).

[20] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proc. of the USENIX Symp. on Networked Systems Design and Implementation (NSDI)* (2013).

[21] FAN, B., ANDERSEN, D. G., KAMINSKY, M., AND MITZENMACHER, M. D. Cuckoo Filter: Practically Better Than Bloom. In *Proc. of the ACM Int'l Conf. on Emerging Networking Experiments and Technologies (CoNEXT)* (2014).

[22] FAN, L., CAO, P., ALMEIDA, J., AND BRODER, A. Z. Summary Cache: a Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking (TON) 8*, 3 (2000), 281–293.

[23] FATAHALIAN, K., AND HOUSTON, M. A Closer Look at GPUs. *Communications of the ACM 51*, 10 (2008), 50–57.

[24] FITZPATRICK, B. Distributed Caching with Memcached. *Linux Journal 2004*, 124 (Aug. 2004), 5.

[25] FOTAKIS, D., PAGH, R., SANDERS, P., AND SPIRAKIS, P. Space Efficient Hash Tables with Worst Case Constant Access Time. In *Proc. of the Annual Symp. on Theoretical Aspects of Computer Science (STACS)* (2003).

[26] FOWLER, G., AND CURT NOLL, L. The FNV Non-Cryptographic Hash Algorithm. `http://tools.ietf.org/html/draft-eastlake-fnv-03`. Accessed: 2015-12-01.

[27] GARCÍA, I., LEFEBVRE, S., HORNUS, S., AND LASRAM, A. Coherent Parallel Hashing. In *Proc. of the ACM SIGGRAPH Conf. and Exhibition on Computer Graphics and Interactive Techniques in Asia (SIGGRAPH Asia)* (2011).

[28] HE, B., YANG, K., FANG, R., LU, M., GOVINDARAJU, N., LUO, Q., AND SANDER, P. Relational Joins on Graphics Processors. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)* (2008).

[29] HE, J., LU, M., AND HE, B. Revisiting Co-Processing for Hash Joins on the Coupled CPU-GPU Architecture. *Proc. of the VLDB Endowment 6*, 10 (2013), 889–900.

[30] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2011.

[31] HERLIHY, M., SHAVIT, N., AND TZAFRIR, M. Hopscotch Hashing. In *Proc. of the Int'l Symp. on Distributed Computing (DISC)* (2008).

[32] HETHERINGTON, T. H., O'CONNOR, M., AND AAMODT, T. M. MemcachedGPU: Scaling-up Scale-out Key-value Stores. In *Proc. of the ACM Symp. on Cloud Computing (SoCC)* (2015).

[33] HETHERINGTON, T. H., ROGERS, T. G., HSU, L., O'CONNOR, M., AND AAMODT, T. M. Characterizing and Evaluating a Key-value Store Application on Heterogeneous CPU-GPU Systems. In *Proc. of the IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)* (2012).

[34] HOFSTEE, H. P. Power Efficient Processor Architecture and the Cell Processor. In *Proc. of the Int'l Symp. on High-Performance Computer Architecture (HPCA)* (2005).

[35] JENKINS, B. 4-byte Integer Hashing. `http://burtleburtle.net/bob/hash/integer.html`. Accessed: 2015-12-01.

[36] JENKINS, B. Some Random Theorems. `http://burtleburtle.net/bob/hash/birthday.html`. Accessed: 2015-12-01.

[37] JHA, S., HE, B., LU, M., CHENG, X., AND HUYNH, H. P. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *Proc. of the VLDB Endowment 8*, 6 (2015), 642–653.

[38] KALDEWEY, T., LOHMAN, G., MUELLER, R., AND VOLK, P. GPU Join Processing Revisited. In *Proc. of the Int'l Workshop on Data Management on New Hardware (DaMoN)* (2012).

[39] KECKLER, S. W., DALLY, W. J., KHAILANY, B., GARLAND, M., AND GLASCO, D. GPUs and the Future of Parallel Computing. *IEEE Micro*, 5 (2011), 7–17.

[40] KEMPER, A., AND NEUMANN, T. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)* (2011).

[41] KHORASANI, F., BELVIRANLI, M. E., GUPTA, R., AND BHUYAN, L. N. Stadium Hashing: Scalable and Flexible Hashing on GPUs. In *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)* (2015).

[42] KIRSCH, A., MITZENMACHER, M., AND WIEDER, U. More robust Hashing: Cuckoo Hashing with a Stash. *SIAM Journal on Computing 39*, 4 (2009), 1543–1561.

[43] KORMAN, S., AND AVIDAN, S. Coherency Sensitive Hashing. In *Proc. of the IEEE Int'l Conf. on Computer Vision (ICCV)* (2011).

[44] LEE, V. W., KIM, C., CHHUGANI, J., DEISHER, M., KIM, D., NGUYEN, A. D., SATISH, N., SMELYANSKIY, M., CHENNUPATY, S., HAMMARLUND, P., SINGHAL, R., AND DUBEY, P. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *Proc. of the Int'l Symp. on Computer Architecture (ISCA)* (2010).

[45] LEFEBVRE, S., AND HOPPE, H. Perfect Spatial Hashing. In *Proc. of the ACM SIGGRAPH Conf. and Exhibition on Computer Graphics and Interactive Techniques (SIGGRAPH)* (2006).

[46] LEVANDOSKI, J. J., LOMET, D. B., AND SENGUPTA, S. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)* (2013).

[47] LI, X., ANDERSEN, D. G., KAMINSKY, M., AND FREEDMAN, M. J. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proc. of the European Conf. on Computer Systems (EuroSys)* (2014).

[48] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)* (2011).

[49] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proc. of the USENIX Symp. on Networked Systems Design and Implementation (NSDI)* (2014).

[50] MAO, Y., KOHLER, E., AND MORRIS, R. T. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proc. of the European Conf. on Computer Systems (EuroSys)* (2012).

[51] MATSUMOTO, M., AND NISHIMURA, T. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 8*, 1 (1998), 3–30.

[52] MCKEE, S. A. Reflections on the Memory Wall. In *Proc. of the ACM Int'l Conf. on Computing Frontiers (CF)* (2004).

[53] METREVELI, Z., ZELDOVICH, N., AND KAASHOEK, M. F. CPHash: A Cache-Partitioned Hash Table. In *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)* (2012).

[54] MITZENMACHER, M. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. on Parallel and Distributed Systems (TPDS) 12*, 10 (2001), 1094–1104.

[55] MITZENMACHER, M. Balanced Allocations and Double Hashing. In *Proc. of the ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)* (2014).

[56] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In *Proc. of the USENIX Symp. on Networked Systems Design and Implementation (NSDI)* (2013).

[57] PAGH, R., AND RODLER, F. F. Cuckoo Hashing. *Journal of Algorithms 51*, 2 (2004), 122–144.

[58] PANIGRAHY, R. Efficient Hashing with Lookups in Two Memory Accesses. In *Proc. of the ACM-SIAM Symp. on Discrete Algorithms (SODA)* (2005).

[59] PIKE, G., AND ALAKUIJALA, J. Introducing CityHash. `http://google-opensource.blogspot.com/2011/04/introducing-cityhash.html`. Accessed: 2015-12-01.

[60] POLYCHRONIOU, O., RAGHAVAN, A., AND ROSS, K. A. Rethinking SIMD Vectorization for In-Memory Databases. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)* (2015).

[61] RAMABHADRAN, S., RATNASAMY, S., HELLERSTEIN, J. M., AND SHENKER, S. Prefix Hash Tree: An Indexing Data Structure Over Distributed Hash Tables. In *Proc. of the ACM Symp. on Principles of Distributed Computing (PODC)* (2004).

[62] RAMAN, V., ATTALURI, G., BARBER, R., CHAINANI, N., KALMUK, D., KULANDAISAMY, V., LEENSTRA, J., LIGHTSTONE, S., LIU, S., LOHMAN, G. M., MALKEMUS, T., MUELLER, R., PANDIS, I., SCHIEFER, B., SHARPE, D., SIDLE, R., STORM, A., AND ZHANG, L. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *Proc. of the VLDB Endowment 6*, 11 (2013), 1080–1091.

[63] RICHA, A. W., MITZENMACHER, M., AND SITARAMAN, R. The Power of Two Random Choices: A Survey of Techniques and Results. In *Handbook of Randomized Computing*, S. Rajasekaran, P. M. Pardalos, J. Reif, and J. Rolim, Eds., vol. 1. Kluwer Academic Publishers, 2001, ch. 9, pp. 255–304.

[64] ROSS, K. Efficient Hash Probes on Modern Processors. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)* (2007).

[65] SONG, H., DHARMAPURIKAR, S., TURNER, J., AND LOCKWOOD, J. Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing. In *Proc. of the Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)* (2005).

[66] STONE, J. E., GOHARA, D., AND SHI, G. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering 12*, 3 (2010), 66–73.

[67] TRIPLETT, J., MCKENNEY, P. E., AND WALPOLE, J. Scalable Concurrent Hash Tables via Relativistic Programming. *ACM SIGOPS Operating Systems Review 44*, 3 (2010), 102–109.

[68] TRIPLETT, J., MCKENNEY, P. E., AND WALPOLE, J. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proc. of the USENIX Annual Technical Conf. (USENIX ATC)* (2011), p. 11.

[69] WU, X., XU, Y., SHAO, Z., AND JIANG, S. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *Proc. of the USENIX Annual Technical Conf. (USENIX ATC)* (2015).

[70] WULF, W. A., AND MCKEE, S. A. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News 23*, 1 (1995), 20–24.

[71] ZHANG, K., WANG, K., YUAN, Y., GUO, L., LEE, R., AND ZHANG, X. Mega-KV: A case for GPUs to Maximize the Throughput of In-Memory Key-Value Stores. *Proc. of the VLDB Endowment 8*, 11 (2015), 1226–1237.