SimdHT-Bench: Characterizing SIMD-Aware Hash Table Designs on Emerging CPU Architectures*

Dipti Shankar, Xiaoyi Lu, Dhabaleswar K. (DK) Panda
Department of Computer Science and Engineering, The Ohio State University
{shankar.50, lu.932, panda.2}@osu.edu

***Abstract*—With the emergence of modern multi-core CPU architectures that support data parallelism via vectorization, several storage systems have been employing SIMD-based techniques to optimize data-parallel operations on in-memory structures like hash-tables. In this paper, we perform an in-depth characterization of the opportunities for incorporating AVX vectorization-based SIMD-aware designs for hash table lookups on emerging CPU architectures. We analyze the challenges and design dimensions involved in exploiting vectorization-based parallel key searching over cache-optimized non-SIMD hash tables. Based on this, we design a comprehensive micro-benchmark suite, SimdHT-Bench, that enables evaluating the performance and applicability of CPU SIMD-aware hash table designs for accelerating different read-intensive workloads. With SimdHT-Bench, we study five different use-case scenarios with varied workload patterns, on the latest Intel Skylake and Intel Cascade Lake multi-core CPU nodes. Further, to validate the applicability of SimdHT-Bench, we employ these performance studies to design a high-performance SIMD-aware RDMA-based in-memory key-value store to accelerate the Memcached 'Multi-Get' workload. We demonstrate that the SIMD-integrated designs can achieve up to 1.45x–2.04x improvement in server-side Get throughput and up to 34% improvement in end-to-end Multi-Get latencies over the state-of-the-art CPU-optimized non-SIMD MemC3 hash table design, on a high-performance compute cluster with Intel Skylake processors and InfiniBand EDR interconnects.**

***Index Terms*—AVX, SIMD, CPU, Hash Table, Key-Value Store**

I. Introduction

With the emergence of modern multi-core CPU architectures that support data parallelism via vectorization, there have been several studies directed towards leveraging CPU-SIMD for accelerating compute for data-intensive workloads. Using 'Single-Instruction-Multiple-Data' (SIMD) instructions to enable data parallelism has been studied for accelerating database operators like scan, join, aggregation [1], [2], and bloom filters [3], [4]. SIMD instructions have also been leveraged to enable data-parallel key lookups over hash tables for join operations [1], [5]–[7]. Similarly, network application scenarios like packet processing, that deal with batched hash table lookups [8], [9], also exploit high-performance CPU-optimized hash tables with SIMD-aware accelerations. These studies make it evident that leveraging SIMD vectorization for parallelizing key searches across hash tables has immense potential for accelerating application workloads that need to facilitate fast in-memory lookup operations.

This research is supported in part by NSF grants #CCF-1822987, #CNS-1513120, #ACI-1450440, #CCF-1565414, and NSF ACI1664137.

On the other hand, distributed and high-performance key-value stores (KVS) utilize hash tables in their backend, as a fast index to store (Set) and lookup key-value pairs (Get/Multi-Get) [10], [11]. They play a vital role in accelerating today's data-intensive workloads in multi-tiered data center architectures. Many studies [12], [13] have shown that the performance of key-value store-based applications is dominated by reads, i.e., GETs. For instance, based on the real-workload traces from Facebook [14], [15], we see that a single web page request from a user can generate up to 521 distinct key-value pair items that need to be fetched from the remote Memcached server cluster. The key-value store applications attempt to minimize the number of network round trips necessary to fetch all the key-value pairs corresponding to the user requests, by batching multiple Gets (24 – 96 keys/request) into a single request. Since most typical key-value stores employ CPU-centric hash tables [11], [16], this motivates us to consider if such a 'Multi-Get' (MGet) workload can leverage SIMD-aware designs explored in the literature.

A. Motivation and Challenges

Towards answering the above question, we first explore SIMD-aware hash table designs in the literature. We find that the state-of-the-art hash table vectorization approaches [1], [5], [6], [8], [9], [17] are focused on accelerating batched key-value pair searches (batched lookups), which is very similar to the Memcached 'Multi-Get' scenario. To better understand these state-of-the-art CPU-optimized hash table designs, we summarize them in Table I-B. From Table I-B, we can see that each of the works propose an SIMD-aware acceleration that employs a different hash table design (memory layout). Therefore, it is evident that these hash table designs are tightly bound to their specific use-case scenarios. Thus, they cannot be directly leveraged to determine if they fit a newer and vastly different usage scenario. To be more precise, there are no studies or benchmarks which provide design guidance on how to maximally exploit the SIMD-aware hash design capabilities for varying workload patterns. This leads us to define our first challenge: Challenge①: *Can we design a micro-benchmark platform that can enable us to study the different state-of-the-art (or any new) SIMD-aware hash table designs on emerging multi-core CPU architectures for varied application scenarios?*

Typically, for non-SIMD variants, the rule-of-thumb is to choose the hash table layout that enables minimal memory

978-1-7281-4045-2/19/$31.00 ©2019 IEEE

and/or cache-line accesses. This has led to heavy reliance on bucketized cuckoo hash tables, that facilitate cache optimal designs for CPUs [12], [17], [18] to support better hash table occupancy (i.e., load factor[1]). In addition to the application-specific memory layouts, the research works in Table I-B also widely differ in how they exploit CPU vectorization capabilities for accelerating batched hash table lookups. For instance, networking applications [8], [9], [17] leverage bucketized cuckoo hash tables with 4-way or 8-way set-associative designs[2]. They mainly leverage SIMD to compare multiple hashes stored in a hash table bucket in parallel. Similarly, analytical database workloads [1], [6] propose leveraging SIMD to search multiple distinct input keys across the hash table in parallel. This defines our second challenge: Challenge②: *How can we determine which SIMD-aware hash table vectorization approach (software approach or algorithm) can best fit our application workload? How much performance can they gain over their non-SIMD hash table counterparts?*

Finally, today's latest Intel Skylake and Cascade Lake nodes [19], [20] support SSE (128-bit), AVX2 (256-bit), and AVX-512 (512-bit) vector instructions. Specifically, with 512-bit extensions to the Advanced Vector Extensions (AVX) SIMD instructions (AVX-512) for x86 ISA, we have the opportunity to operate on an entire cache-line in a single instruction. Thus, with this increasing vector-processing capability on modern hardware, we have the option to leverage different CPU-SIMD vector widths to enable different degrees of data parallelism for a given SIMD vectorization approach. This is also evident across the different works summarized in Table I-B. Based on this, we define our third challenge: Challenge③: *How much SIMD parallelism (hardware capability or CPU vector instructions) can we exploit to accelerate hash table lookups for a given application workload?*

B. Contribution

To address the above research challenges, we first characterize and define SIMD-aware design dimensions that need to be considered for any application workload. Since we are mainly interested in enabling faster performance for read-dominated workloads, we focus our studies on cuckoo hash tables that enable near-constant lookup time [21], [22]. Based on these characteristics, we design "SimdHT-Bench", to help researchers study the performance of different SIMD-aware hash table designs in-depth for any use-case involving batched reads. This proposed micro-benchmark suite presents a:

1) Comprehensive design, that takes hash table memory layout, workload data access pattern and the CPU-SIMD capabilities into account, to address Challenge①,
2) SIMD algorithm validation engine that can help determine which hash table designs and CPU-SIMD vector lengths can be leveraged for a given application workload and CPU hardware, to address Challenge③, and,
3) Performance engine that evaluates different viable SIMD-aware hash table designs, and presents a performance-centric compare-and-contrast with its non-SIMD equivalents, towards answering Challenge②.

We present five different use-cases with SimdHT-Bench, including: (a) two case studies contrasting horizontal and vertical SIMD-aware parallel lookup approaches for typical database workloads, (b) performance of different SIMD-aware hash tables on Intel Skylake and Intel Cascade Lake nodes, (c) performance with variable length keys/payloads, and, (d) leveraging different SIMD vector widths to enable different parallelisms over the same underlying hash table design. From these experiments, we observe that, irrespective of the hash table vectorization approach being leveraged, it is vital to employ SIMD vector widths that enable minimal memory/-cache accesses for optimal performance. We also observe that: (a) the vertical SIMD approach over a 3-way cuckoo hash table with AVX-512 (512-bit vectors), and, (b) the horizontal AVX2-based SIMD approach (256-bit vectors) over a 2-way bucketized cuckoo hash table with 4 slots-per-bucket, can provide the best lookup performance benefits across uniform and skewed access patterns.

We validate our performance studies with SimdHT-Bench by integrating these SIMD-aware designs into a high-performance in-memory key-value store like RDMA-Memcached [23], [24], and contrast it with the state-of-the-art non-SIMD CPU-optimized MemC3 [12] backend design. Our performance evaluations show that 'SIMD+RDMA-Memcached' can achieve about 1.45x–2.04x improvement in server-side Get throughput and 10%–34% improvement in the end-to-end 'Multi-Get' latencies, over 'MemC3+RDMA-Memcached', on a high-performance compute (HPC) cluster with Intel Skylake CPUs and InfiniBand EDR interconnects. The rest of the paper is organized as follows. Section II presents the necessary background and related work. Section III characterizes SIMD-aware cuckoo hash table designs. Section IV describes SimdHT-Bench and Section V presents performance studies. Section VI discusses our key-value store use-case. We conclude in Section VII with future work.

Research Work	Design Dimensions		
	Memory Layout #Slots-Per-Bucket x (Key Size, Payload Size)	N-way Hashing	SIMD-aware Design
MemC3 [12]	4 x (1 B, 8 B)	2-way	No
SILT [18]	4 x (2 B, 4 B)	2-way	No
CuckooSwitch [17]	4 x (6 B, 2 B)	2-way	No
Vectorized BCHT [1], [25]	2 x (4 B, 4 B) 8 x (4 B ,4 B)	2-way	Yes (SSE for CPU, AVX-512 for Phi)
Vectorized Cuckoo HT [1]	1x(4 B, 4 B)	2-way	Yes (AVX2 for CPU, AVX-512 for Phi)
Cuckoo++ [8]	8x 2B, 48B *payload = per-bucket-metadata	2-way	Yes (SSE)
DPDK [9]	8 x (4 B, 8 B)	2-way	Yes (SSE)

TABLE I
STATE-OF-THE-ART RESEARCH WORKS EMPLOYING CPU-OPTIMIZED CUCKOO HASH TABLE VARIANTS

II. BACKGROUND & RELATED WORK

In this section, we provide an overview of the basics of cuckoo hashing and the state-of-the-art SIMD-aware hash table

[1]load factor (LF) = (number-of-items-that-can-be-inserted / hash-table-size).
[2]no. slots-per-bucket = set-associativity of a cuckoo hash table bucket

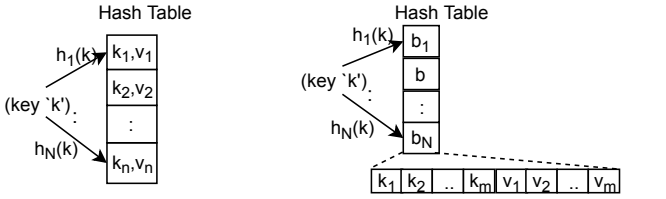
designs in the literature.

A. Cuckoo Hashing: The Basics

Since we are interested in read-dominated workloads, we focus on cuckoo hashing [21], [26], which enables a key-value pair to be located[3], i.e, a hash table lookup, using a constant number of memory accesses (unlike other collision resolution hash table schemes such as chaining, linear probing, etc.). Cuckoo hashing is a well-known open-addressing based hashing scheme, that maintains two hash functions (h1 and h2), such that, a key can be found in exactly one of two locations (or hash buckets). It achieves this simplicity by shifting the complexity from lookup to insertion.

New insertions can potentially relocate an existing key-value pair to its alternative bucket if both h1 and h2 are occupied. The "re-hashing" continues until an empty hash table entry is found. This process is time-consuming and is amortized only as long as the load factor[1,4] is below 50%. However, a small load factor hurts the hit rate and memory occupancy of the hash table. To overcome this, practical high-performance variants [12], [17], [18] employ two approaches:

1) N-way Cuckoo Hashing (N-way): This approach enables a higher load factor by employing more than two hash functions. This generic variant of the basic cuckoo hash table is referred to as N-way hashing, as it provides 'N' potential hash buckets to locate a given key. It follows a vertical layout design, as presented in Figure 1(a).



(a) N-way Cuckoo Hash Table (b) (N,m) Bucketized Cuckoo Hash Table

Fig. 1. High-Performance Cuckoo Hash Table Variants

2) Bucketized Cuckoo Hash Table (BCHT): Bucketized cuckoo hash table is a variant of the N-way cuckoo hash table that enables more than one key-value pair to be stored into each hash bucket. By doing so, it lowers the probability of re-location and improves load factor without needing to increase 'N'. It follows an (N, m) set-associative hash table design, with N-way hashing and 'm' slots-per-hash bucket as presented in Figure 1(b). The hash table size is (N*m) for a bucketized (N,m) cuckoo hashing (NOTE: an N-way cuckoo HT is intuitively an (N,1) BCHT).

We summarize the load factor improvements observed with these two variants, based on studies from [22], in Figure 2. For our discussions, load factor indicates the maximum load factor (i.e., max. number-of-items that can be inserted) for a given hash table design. From Figure 2, we can see that different cuckoo hash table variants enable different hash table memory occupancy characteristics. This 'load factor' determines the hit rate of the system whose index it maintains, and can, in turn, affect performance. For instance, as compared to the non-bucketized 2-way cuckoo hash table (LF = 0.5):

(a) increasing slots-per-bucket to 4, i.e., a (2,4) BCHT can increase the load factor to 93%. Additionally, if we can fit each bucket into fewer cache-lines, as in [8], [12], we can enable high throughput for look-ups by reducing the number of memory accesses.

(b) increasing N from two to three, i.e., 3-way cuckoo hashing, we can improve the load factor to 91%. However, this comes at the cost of an additional hash bucket access over 2-way cuckoo hashing (three memory access in the worst-case), spanning multiple cache-lines.

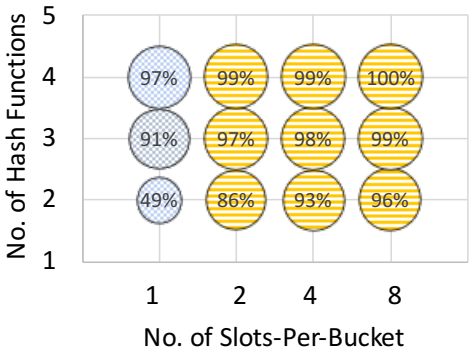


Fig. 2. Cuckoo Hash Table Variants [22], [27]: Load Factor vs. N-way Hashing vs. BCHT; 1 slot-per-bucket = non-bucketized 'N-way' cuckoo hash table (represented in blue); For BCHT, N = 2, 3, 4 and #slots-per-BCHT-bucket = 2, 4, 8 (represented in yellow)

B. SIMD-Aware Hash Table Designs: Related Work

The idea of employing SIMD vectorization to improve the performance of critical data structures has been well studied in the field of database systems. We provide an overview of some of these related research works, that has served as a motivation for this paper.

For accelerating hash table lookups for database and networking application use-cases, various non-SIMD CPU-optimized and SIMD-aware research works have been proposed, as summarized in Table I-B. Similarly, using SIMD instructions to enable other data-parallel database operators such as scan, join, aggregation [1], [2], [28], etc., and for data structures like bloom filters [3], [4], [6] have also been well studied. Specific to key-value stores, recent work by Pilman et. al [28] focuses on improving scan performance for complex analytical queries. As an orthogonal approach, SIMD-aware designs have also been studied over offload-based accelerators like GPGPUs [29], [30]. They leverage the high memory bandwidth of modern GPUs and its data hiding capabilities to achieve a higher hash table lookup performance at the server.

III. Characterizing SIMD-Aware HT Designs

In this section, we focus on understanding existing high-performance cuckoo hash table variants, and define newer

[3] 'value' associated with a key-value pair is also referred to as 'payload'

[4] For load factor, 'number-of-items-that-can-be-inserted' is typically less than the 'hash-table-size' due to probability of cuckoo re-hashing.

dimensions introduced by SIMD-aware designs.

A. Basic Design Considerations

From the research works presented in Section II-B, we observe that several application workloads [8], [12], [25], [30] leverage cuckoo hash table variants to enable faster lookups. As seen from Table I-B, these high-performance cuckoo hash table implementations employ different in-memory layouts that cater to their specific application needs. These workload-specific design dimensions can be characterized as follows:

1) Memory Layout: If we map the different hash table layouts in Table I-B to Figure 2, we observe that almost all use-cases explore hash table designs that ensure a high load factor (>90%). As detailed in Section II-A and Figure 2, this can be enabled using two approaches for (N,m) cuckoo hashing: (a) increasing the number of slots-per-bucket (i.e., 'm'), or, (b) increasing the buckets a key is mapped to, i.e., number of hash functions (i.e., 'N'); each of which has a different memory access cost. Based on the key/payload sizes, the choice of 'N' and 'm' defines a unique 'memory layout'. This specifies our first generic design dimension; one that has been well-studied in the non-SIMD scenario, as seen from Table I-B.

2) Workload Data Access Pattern: While we focus our studies on read-dominated workloads, various hash table application scenarios have vastly different access patterns. For instance, key-value store workloads are dominated by skewed data accesses (where some keys are more frequently queried over others) [12], [15]. Conversely, hash table accesses in network packet processing applications follow a more uniform access pattern [8], [17]. Thus, the workload access pattern defines our second basic design dimension.

B. Defining SIMD-Aware Dimensions

From the research works in Table I-B, that present designs to exploit CPU's vectorization capabilities to accelerate hash table lookups, we observe that leveraging CPU vectorization introduces two new SIMD-specific hash table design dimensions to consider.

1) SIMD Vectorization Approach – Horizontal vs. Vertical: For leveraging SIMD to accelerate cuckoo hash tables, the first question that needs to be answered is: **In what ways can we leverage SIMD vectorization to accelerate hash table lookups?** This defines our first SIMD-Aware design dimension. As discussed in [1], we find that this can be achieved along with two software/algorithmic approaches:

(a) Horizontal SIMD Vectorization: Consider a bucketized cuckoo hash table with two or more slots-per-bucket. Rather than individually comparing a given key against every single key in the designated hash bucket(s), we can load an entire bucket onto a CPU vector and probe all possible hash locations in parallel. For example, consider the 2-way bucketized cuckoo hash table with 4 slots-per-bucket with 32-bit keys/payloads, as depicted in Figure 3(a). With this, we can load both hash buckets designated for a given key 'k' into a 256-bit vector and search all 8 possible hash table locations in one AVX compare instruction. Since this approach exploits SIMD for a single key-value pair lookup across multiple slots in a bucket, it can be considered as a reduction operation and is referred to as the horizontal vectorization approach [5], [8], [30].

(b) Vertical SIMD Vectorization: The fundamental principle involved in this approach is to process a different key per SIMD lane. This enables a true SIMD approach as we can lookup 'w' keys in the hash table in parallel[5]. For instance, consider the 2-way cuckoo hash table with 32-bit key and payload, as depicted in Figure 3(b). With this approach, we can gather distinct keys from 8 different hash locations (i.e., $[h(k_1), h(k_2), ..., h(k_w)]$ with 'w'=8) corresponding to keys in the input (i.e., $[k_1, k_2, ..., k_w]$) , and thus search for 8 different keys in parallel with one AVX compare instruction. Since this approach exploits SIMD to process multiple keys in parallel and returns a vector of values corresponding to the matching keys, it is referred to as the vertical vectorization approach (and explored in [1], [6]).

2) SIMD Parallelism – SSE vs. AVX2 vs. AVX-512: Today's latest Intel Skylake and Cascade Lake nodes [19], [20] support SSE (128-bit), AVX2 (256-bit), and AVX-512 (512-bit) vector instructions. Thus, for a given SIMD vectorization approach, we have the option to leverage different CPU-SIMD vector widths. This leads us to our second SIMD-centric question: **How much SIMD parallelism can we exploit from the underlying CPU hardware to accelerate hash table lookups?**

For example, for the horizontal vectorization approach presented in Figure 3(a) for a 2-way bucketized cuckoo hash table with 4 slots-per-bucket, we could use: (a) 256-bit vectors to lookup all 8 locations in parallel (w=8) in a pessimistic fashion, or, (b) use 128-bit vectors to lookup a single bucket at a time (i.e., 4 hash locations in parallel; w=4) in an optimistic fashion. Similarly, for the 2-way cuckoo hash table in Figure 3(b) that employs vertical vectorization via AVX2 to lookup 8 keys (w=8) in parallel, we could also employ AVX-512 to lookup 16 keys (w=16) in parallel instead. Thus, the choice of 'w' is our second SIMD-aware design dimension.

To facilitate studying the different SIMD-aware and workload-centric hash table design choices, we present the "SimdHT-Bench" micro-benchmark suite.

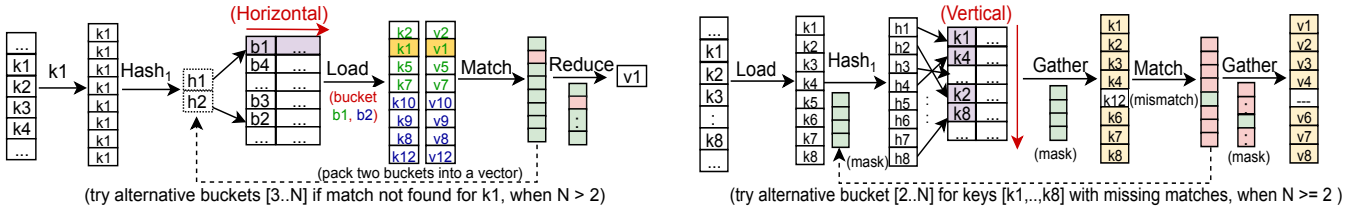
IV. Benchmarking SIMD-Aware HT Workloads

Our goals for SimdHT-Bench are two-fold: First, based on design dimensions discussed in Section III-B, we determine which SIMD-aware designs fit the desired application workload characteristics. Second, we build and study the lookup performance benefits that can be enabled by each of SIMD-aware design variants calculated. With such a framework, we can help determine if SIMD-aware parallel approaches for the hash table (HT) lookups can benefit the application workload characteristics in question.

A. Overview of SimdHT-Bench Micro-benchmark Suite

To meet the above goals, we design our proposed SimdHT-Bench to capture the essence of various aspects of cuckoo

[5] 'w' is referred to as the SIMD width



(a) Horizontal Vectorization Example: First Iteration with key 'k1' over (2,4) BCHT; Repeats (n-1) times for 'n' keys in input array

(b) Vertical Vectorization Example: First Iteration with keys k1,..,k4 over 2-way Cuckoo HT; Repeats (n/w-1) times for 'n' keys in input array

Fig. 3. Vectorization-based Parallel Key Lookups for SIMD CPUs: Illustration with 2-way cuckoo hash table with bucketized layout with 4 slots-per-bucket and non-bucketized 2-way Cuckoo HT); key/payload size (bits) = (32, 32) and length of vector = 256 bits; 'pink' in mask refers to a 'match'

HT workloads. As shown in Figure 4, our proposed micro-benchmark suite contains the following modules:

1) Configurable Input Parameters: The benchmark's input interface enables the user to specify the workload characteristics, like, (a) HT layout and size, (b) key/value sizes (key and payload stored in the HT), and (c) workload access pattern. It optionally enables specifying the SIMD vector lengths[5] and vectorization approaches (horizontal or vertical) to consider.

2) Workload/Table Generator: The table generator creates an HT with layout [N = no. of hash functions for cuckoo hashing, m = slots-per-bucket, layout = bucketized/non-bucketized, k = key's hash size, v = payload size, b = HT size] and a desired load factor (LF). The workload generator is designed as a pluggable module that creates a list of keys to query, to enable evaluating user-specified access patterns. For this paper, we focus our design framework for read-only workloads.

We currently support skewed and uniform distributions. For skewed, we plug-in "mutilate" workload generator [15], [31], that emulates an access pattern typical to key-value stores like Memcached. The workload pattern parameter also determines the number of worker threads (i.e., working cores) and if the table is shared or dedicated per-core.

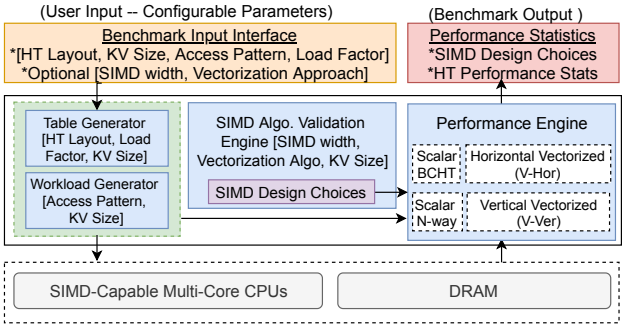


Fig. 4. Micro-benchmark Suite Design

3) SIMD Algorithm Validation Engine: The SIMD algorithm validation engine determines which CPU-SIMD vector lengths and SIMD vectorization approaches fit the given user-specific HT layout. It creates a list of these viable SIMD-aware HT design choices that can be used to study the lookup performance. We describe it in Section IV-B.

4) Performance Engine: The performance engine loads and queries the cuckoo HT for each of the design choices listed, using the workload pattern generated. For each of the SIMD-based design choices, it does a compare-and-contrast with the corresponding non-SIMD aware (scalar) version of the vectorization-based lookup algorithm, and computes the HT throughput performance. Finally, the SIMD-aware design choices and their performance comparison stats are returned to the user.

B. Determining SIMD-Centric Design Choices

From Figure 2, we observe that, for a given load factor, there are various bucketized (N,m) and non-bucketized (N-way) cuckoo HT variants. Also, from Section III-B, we see that the two vectorization-based parallel lookup approaches can be enabled by varying SIMD vector widths. Based on these, we can easily determine a list of possible SIMD-aware design combinations. However, we need to determine if each of these combinations can be supported by: (a) the application workload in question (specified by the configurable input parameters), and, (b) the underlying CPU architecture. Towards enabling this validation, we present two validators to help filter out viable SIMD design candidates: (a) Horizontal-over-BCHT Validator for the bucketized table layout, and, (b) Vertical-over-CuckooHT Validator for the non-bucketized cuckoo HT layout.

The 'Horizontal-over-BCHT' validator, presented as function 'HorV-Valid' in Algo. 1, checks if one or more hash buckets (containing keys and payloads) can fit into the CPU vector of width 'w'. Similarly, the 'Vertical-over-CuckooHT' validator, presented as function 'VerV-Valid' in Algo. 2, determines if it can fit two or more keys to probe in-parallel onto the CPU vector of width 'w'. For the corresponding non-SIMD versions, all vector instructions are replaced with scalar load/store/compare operations. Additionally, for these scalar counterparts, we have: (a) for the non-SIMD BCHT, 'bucks-per-vec'=1 in Algo. 1, and, (b) for the non-SIMD N-way cuckoo HT, 'keys-per-iter'=1 in Algo. 2.

C. Implementing Generic Templates for SIMD-Aware Designs

To study SIMD-aware HT lookup operations, we design generic templates for the horizontal and vertical vectorization approaches discussed in Section III-B, to support varying CPU vector widths (based on [1]). We generically define the different vector operations as vec_<operation>$_{x,W}$(), wherein, 'W' signifies the length of the CPU vector (in bits), and,

'x' the width of each lane within the vector (such that 'W'/'x'= SIMD width 'w'). We define function wrappers for the underlying vector instructions supported to enable running our experiments. For instance, for a 256-bit vector over 32-bit keys, we have vec_cmpeq$_{32,256}$, i.e., we need to employ the AVX2 compare instruction `_mm256_cmpeq_epi32`.

Now, while these templates give a generic representation of the vectorization-based lookups, similar to [1], we have:
(a) for vertical, we try to reduce the number of cache accesses by packing the `vec_gather_key` and `vec_gather_val` gathers into fewer wider gathers (e.g., 16-way 32-bit gathers vs. 8-way 64-bit gathers for AVX-512), and,
(b) for horizontal, we try to leverage vector instructions to calculate the hash buckets of multiple keys in parallel (i.e., `calc_N_hash_buckets`).

Algorithm 1: Horizontal Vectorization Template

Result: Returns Lookup Throughput (Lookups/s)
Input : SIMD width 'w', HT Layout '(N,m)', Key Size 'ks', Val Size 'vs', Workload 'p_k[n]'
Output: Throughput

Function `HorV-Valid(`w, (N,m), k, v`)`**:**
  assert($m > 1$) ▷ check if layout is BCHT
  if $(w >= (k+v) * m)$ **then**
    Buckets-Per-Vector = $ceil(N * ((k+v) * m)/w)$ ▷ calculate no. of buckets fit into vector width 'w'
    return True, Buckets-Per-Vector
  else
    return False, 0
  end

declare V[1..n]
start_timer(t);
assert(bucks-per-vec = **HorV-Valid**(w, N, m, ks, vs)) ▷ HOR-Vec
for (*each k in p_k[]*) {
  $H[1..N]$ = calc_N_hash_buckets(k)
  for (*each i in N / bucks-per-vec*) {
    $\overrightarrow{k}$ = vec_set_lanes$_{ks,w}$(k); ▷ replicated key k onto all lanes of vector
    $\overrightarrow{b}$ = vec_load_buckets$_{ks,w}$(H, w, bucks-per-vec) $\overrightarrow{t_k}, \overrightarrow{t_v}$ = vec_shuffle_and_blend$_{ks,w}$($\overrightarrow{b}$, m); ▷ extract and separate keys and values from bucket
    $\overrightarrow{match}$ = vec_cmpeq$_{ks,w}$($\overrightarrow{t_k}$, $\overrightarrow{k}$)
    V_k = vec_reduce$_{vs,w}$($\overrightarrow{match}$, $\overrightarrow{t_v}$ ▷ reduce vector to find matching payload
  }
}
end_timer(t);
return calc_thr(t, n);

Algorithm 2: Vertical Vectorization Template

Result: Returns Lookup Throughput (Lookups/s)
Input : SIMD width 'w', HT Layout 'N', Key Size 'ks', Val Size 'vs', Workload 'p_k[n]'
Output: Throughput

Function `VerV-Valid(`w, k, v`)`**:**
  assert($is\_power\_of\_two(k)$) ▷ check if layout is N-way HT
  if $(w > (k+v))$ **then**
    Keys-Per-Iteration = w/k ▷ calculate number of keys that can be probed in parallel
    return True, Keys-Per-Iteration
  else
    return False, 0
  end

declare V[1..n]
start_timer(t);
assert(keys-per-iter = **VerV-Valid**(w, N, ks, vs)) ▷ VER-Vec
for (*each k[] in p_k[]; steps of 'keys-per-iter'*) {
  $\overrightarrow{k}$ = vec_load_lanes$_{ks,w}$(k); ▷ replicated key k onto all lanes of vector
  $\overrightarrow{h}$ = vec_calc_hash$_{ks,w}$($\overrightarrow{k}$); ▷ calculate hash with one key-per-lane from k[]
  $\overrightarrow{t_k}$ = vec_gather_key$_{ks,w}$($\overrightarrow{k}$); ▷ gather key from hash buckets per lane
  $\overrightarrow{match}$ = vec_cmpeq$_{ks,w}$($\overrightarrow{t_k}$, $\overrightarrow{k}$)
  $\overrightarrow{t_v}$ = vec_gather_val$_{vs,w}$($\overrightarrow{match}$, $\overrightarrow{h}$); ▷ gather key from hash buckets per lane
  vec_store_val$_{vs}$(V_k, $\overrightarrow{t_v}$) ▷ store value results computed from vector to memory
}
end_timer(t);
return calc_thr(t, n);

D. Leveraging SimdHT-Bench for Application Studies

To enable a new application to leverage SimdHT-Bench, we require the following two steps: (a) plug-in a new workload pattern that mimics the application into the framework's pluggable workload generator, and, (b) design new templates for any additional functionality beyond lookups. For instance, if we want to support HT updates in addition to parallel-lookups, a new access pattern with HT insert/delete operations can be easily accommodated into the pluggable workload generator.

Based on SimdHT-Bench, we present detailed evaluations over different CPU architectures in the following section.

V. Performance Studies

In this section, we present the results of our in-depth analysis of the SIMD-aware cuckoo hash table (HT) designs over the latest multi-core CPU architectures, with SimdHT-Bench. We divide our evaluations into the following categories: (1) Stand-alone HT performance with vectorization-based parallel searching algorithms, and, (2) Evaluations with RDMA-based Memcached using Memslap micro-benchmarks with integrated SIMD-aware HT.

A. Experimental Setup

We use the following three HPC clusters for our evaluations:
Cluster A: Each node in this testbed is provisioned with 40-core Intel Skylake, with dual 20-cores Gold 6148 processors, 192 GB DRAM, and connected via Dual Port EDR Mellanox InfiniBand EDR interconnects (100 Gbps).
Cluster B: Each node in this testbed is provisioned with Intel Skylake dual fourteen-core processors (28-cores), 128 GB DRAM, connected via Mellanox InfiniBand EDR interconnects (100 Gbps).
Cluster C: In this testbed, each node is provisioned with Intel

Cascade Lake-SP, with 48 cores on two sockets (24 cores-per-socket), supporting a total of 96 hardware threads per node (48 x 2 threads-per-core). It is equipped with 192 GB DRAM.

For all our experiments, we use full-subscription mode, i.e., we run one-process-per-core for all cores on the test node over a 'shared' HT, unless specified otherwise. For each experiment, we present the average of five runs and measure lookup performance as average throughput per core (per process) in billion lookups/sec. We present studies with different cuckoo HT variants discussed in Section II-A. We denote the bucketized cuckoo HT as '(N, m) BCHT' ('m'>1) and the non-bucketized cuckoo HT as 'N-way cuckoo HT' ('m' = 1) in our analysis, and compare each of them with their non-SIMD counterparts. We refer to the SIMD-aware (N, m) BCHT and N-way cuckoo HT as 'Vector', and, its non-SIMD equivalents as 'Scalar' in the experimental figures to follow.

B. Case Study ①: Horizontal vs. Vertical SIMD Approaches

For our first case study (Case Study①(a)), we base our experiments on the SIMD-aware cuckoo HT studies in [1] for database workloads. While they restrict the load factor to 50% to explore 2-way cuckoo HT and (2, 2) BCHT, we extend these studies to a load factor (LF) of 90% to generate our table and configure the data access pattern with a 90% hit rate[6] for both skewed and uniform distributions. This study also attempts to compare the two different SIMD approaches (horizontal-vs.-vertical) and also contrasts different access patterns (uniform-vs.-skew) and SIMD widths. To study SimdHT-Bench's validation and performance engine, we supply these input parameters to our benchmark interface, and enable it to generate different cuckoo HT suitable for this workload.

```
*(k,v) = (32, 32); 'w' = 128, 256, 512
*****skylake
*(2,1) -> V-Ver, Opts: 256 bit - 8 keys/it, Opts:
    512 bit - 16 keys/it
*(3,1) -> V-Ver, Opts: 256 bit - 8 keys/it, Opts:
    512 bit - 16 keys/it
*(4,1) -> V-Ver, Opts: 256 bit - 8 keys/it, Opts:
    512 bit - 16 keys/it
*(2, 2) -> V-Hor, Opts: 128 bit - 1 bucket/vec, Opts
    : 256 bit - 2 bucket/vec
*(2, 4) -> V-Hor, Opts: 256 bit - 1 bucket/vec, Opts
    : 512 bit - 2 bucket/vec
*(2, 8) -> V-Hor, Opts: 512 bit - 1 bucket/vec
*(3, 2) -> V-Hor, Opts: 128 bit - 1 bucket/vec, Opts
    : 256 bit - 2 bucket/vec
*(3,4) -> V-Hor, Opts: 256 bit - 1 bucket/vec, Opts:
     512 bit - 2 bucket/vec
*(3,8) -> V-Hor, Opts: 512 bit - 1 bucket/vec
```

Listing 1. SIMD-Aware Cuckoo HT Design Choices Output

For this experiment, for each (N, m) cuckoo HT design, we: (a) vary 'N' between 2–4 for the non-bucketized cuckoo HT, and, (b) vary 'N' between 2–3 and 'm' between 2–8. We run these experiments on a single 40 core node on Cluster A. Figure 5 presents the performance comparisons for various SIMD-aware designs based on the Listing 1. For each (N, m) variant, we present the performance w.r.t the top listed SIMD approach that utilizes the smallest SIMD width vector. We study the performance for both uniform and skewed key data access distributions. From this figure, for small HT size, the non-SIMD variants with the smaller 'm' seems to give the best performance. Most importantly, we observe that the two vectorization-based approaches can benefit up to 2.97x for uniform distribution.

For skewed distribution, the benefit pattern seems different:

- The non-SIMD (scalar) variants can maintain a higher performance for a skewed pattern as compared to a uniform distribution. It can leverage the basic temporal cache locality for frequently accessed keys. Thus, for BCHT, the benefit of horizontal vectorization is small for 'm=2', and increases from 1.23x to 1.98x with 'm=4'.
- For N-way hashing, however, we observe benefits of about 1.67x–1.97x to maintain load factor at around 90% with 'N=3 or 4'. It can enable up to 2.45x benefits if we choose to support a very small load factor with 'N=2'.

For the second set of experiments (Case Study①(b)), we extend the above studies to varying key-value pair sizes and out-of-cache HT sizes. Figure 6 presents the performance results for varying HT sizes (between 256 KB that fits in L2 cache of 1 MB to 4 MB and 64 MB). We employ a load factor of 90% and a hit rate of 90% with the uniform data access pattern. From this figure, we can observe that as the HT size increases from 256 KB to 64 MB the benefits of both horizontal and vertical vectorization approaches reduces from an average 3.5x to about 1.5x on Intel Skylake nodes.

From an overall perspective, we observe that:

Observation①: For HT that fits into the cache, contrasting the different (N, m) variants across bucketized and N-way hashing, we observe that 3-way cuckoo HT and (2, 4) BCHT provide the best sustainable benefits of about 1.4x – 1.95x for a maximum load factor of about 90%. For a load factor less than 50%, employing 2-way cuckoo HT with a vertical approach can give the best performance as compared to the horizontal approach. The latter is also observed in [1], that compares 2-way cuckoo HT with AVX2 (V-Ver) in contrast to (2, 2) BCHT (V-Hor) with SSE for LF=50%.

C. Case Study ②: Supporting 16-bit and 64-bit Hash Keys

Various workloads [8], [12] need to support variable-length keys and payloads. These variable-length keys are stored in HT as using 32-bit/64-bit/16-bit hashes. We try to illustrate a scenario and contrast it with Case Study ①. Further, this case study contrasts the two SIMD approaches (horizontal-vs.-vertical) with the HT size. Figure 7(a) presents the performance results for varying key-value pair sizes: (a) for larger-than-integer, i.e, (K,V) = (64 bits, 64 bits) over 3-way cuckoo HT, and, (b) mixed payload sizes scenario, i.e., K,V = (16 bits, 32 bits) over (2, 8) BCHT. We employ load factor of 90%, hit rate of 90%, 512 KB HT size, to showcase both skewed and uniform data access patterns.

From Figure 7(a), we can see that, for non-SIMD variants of both (a) and (b), there are no significant variations in performance as compared to (K,V) = (32, 32) (Figure 5), as the

[6]hit rate (or selectivity) refers to the number of input keys that are likely to be found in the hash table

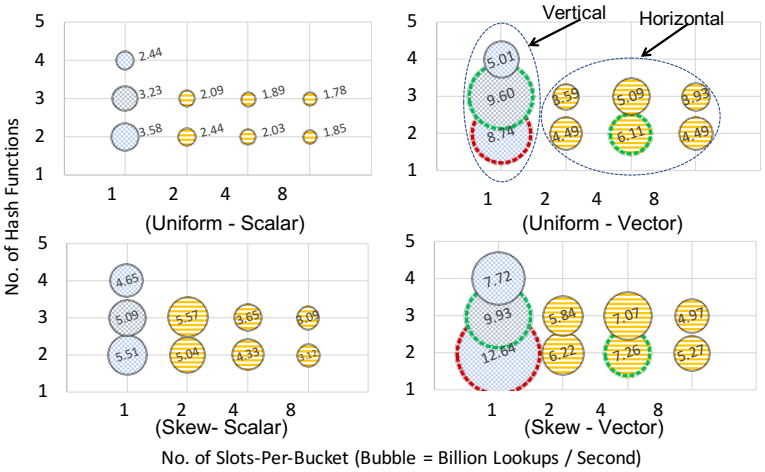


Fig. 5. Case Study ①(a): Contrasting SIMD Vectorization Approaches on Intel Skylake: 1 MB HT Size, (K,V) = (32 bits, 32 bits) LF=90% and Hit Rate 90%; Each 'bubble' in this graph represents the lookup performance in billion lookups/sec; 'Blue' represents vertical vectorization over N-way cuckoo HT (1 slot-per-bucket / non-bucketized) and 'Yellow' represents (N, m) bucketized cuckoo HT variants; Best performance marked in 'Red' and best LF-performance combination in 'Green'

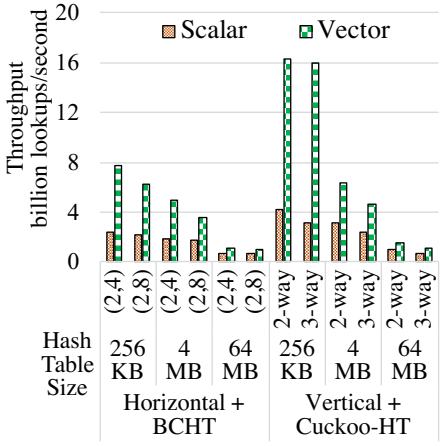


Fig. 6. Case Study ①(b): Performance with Varying Hash Table Sizes on Intel Skylake for Uniform Data Access Pattern

number of cache-lines accessed does not change. For (K, V) = (16, 32), the horizontal SIMD approach over (2, 8) BCHT observes about 4.16x improvement over its non-SIMD variant with AVX-256. However, for (K, V) = (64, 64), we observe that the lookup performance gains only 1.37x with the vertical approach, i.e., a 40% slower compared to running over (K, V) = (32, 32). While we do perform the same number of gather operations (up to three for keys and one for value with 3-way cuckoo), we lose performance as AVX2/AVX-512 support a maximum gather size of 64-bits per lane (i.e., eight different cache-lines per gather) on the latest CPU architectures. Thus, we can no longer leverage fewer wider gather to minimize the number of cache-line accesses as in the case of 32-bit keys, as discussed in Section IV and [1].

Specifically for vertical SIMD, we find that:

Observation②: For the vertical SIMD approach over N-way cuckoo HTs, enabling fewer wider gathers is critical for performance, irrespective of SIMD vector width. Thus, there is a need for either: (a) hardware-optimized 'gather' intrinsics can take some prefetching hints, or, (b) wider-than-64 bit 'gather' operations.

D. Case Study ③: AVX2 vs. AVX-512

From Figure 5 and Figure 7(a) , we observe that enabling a larger 'm' (e.g., (2, 4) vs. (2, 8)) for BCHT via wider SIMD vectors does not demonstrate any considerable benefits. To understand the impact of SIMD vector widths further, Figure 7(b) extends the above experiment to run performance comparisons with 3-way cuckoo HT and (2, 8) BCHT, with 256-bit and 512-bit vector instructions. For 3-way cuckoo HT, this means contrasting the SIMD parallelism of 8 keys/iteration with AVX2 and 16 keys/iteration with AVX-512. For (2, 8) BCHT, this means looking up each of the two buckets one at a time with AVX2 vs. loading both designated (N=2) hash buckets for probing in parallel with AVX-512. We also vary the number of concurrent processes (20 cores and 40 cores) probing the HT, to study the corresponding performance impact. From Figure 7(b), we observe that:

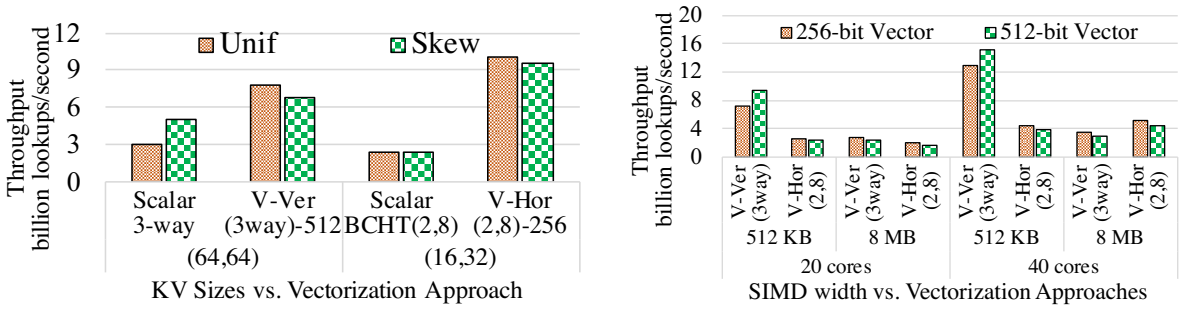
Observation③: For N-way cuckoo hashing, increasing SIMD vector width by two improves the performance by as little as 25%, for HTs that fit into the cache. No improvements are observed for larger HTs, especially for a load factor >90%. For BCHT, probing multiple hash buckets per-key in parallel does not demonstrate significant performance benefits over probing one bucket per vector instruction.

E. Case Study④: Intel Skylake vs. Intel Cascade Lake

To contrast performance on the latest CPU architectures, across different access patterns (uniform-vs.-skew) and HT sizes, we study the (2, 4) BCHT with horizontal SIMD approach and 3-way cuckoo HT with vertical SIMD support on an Intel Cascade Lake node on Cluster C running 68 processes and contrast it with Intel Skylake node on Cluster A running 40 processes. Figure 8 presents results for the same, with a 90% load factor and a 90% hit rate, over 1 MB and 16 MB HT sizes. From this figure, we can observe that Cascade Lake maintains a gain of about 1.5x over Skylake across both SIMD-aware designs. However, for skewed workloads, 3-way vertical SIMD can enable visible gains while (2, 4) BCHT performs similar to its non-SIMD equivalent.

F. Case Study⑤: Can we use Vertical SIMD on BCHTs and Horizontal SIMD on N-way Cuckoo HT?

Now, having restricted vertical SIMD to N-way cuckoo HTs and horizontal SIMD to BCHTs for the previous case studies, we now to explore if any hybrid vectorization approaches are viable. Running horizontal SIMD on N-way cuckoo HT is equivalent to its non-SIMD version as 'm=1'. However, we



(a) Case Study ②: (K,V) = (64, 64) and (K,V) = (16, 32)

(b) Case Study ③: SIMD widths (AVX2 vs AVX-512)

Fig. 7. Lookup Performance with varying Key-Value Pair Sizes

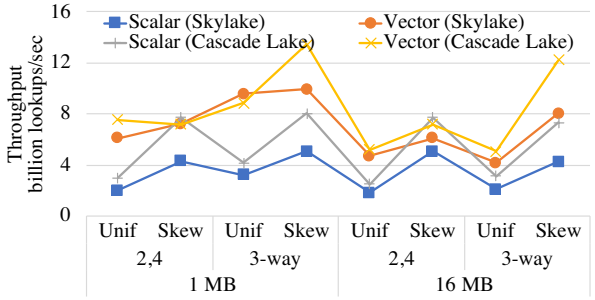


Fig. 8. Case Study④: Contrasting Lookup Performance on Intel Cascade Lake and Intel Skylake; Horizontal SIMD on (2, 4) BCHT and Vertical SIMD on 3-way Cuckoo HT

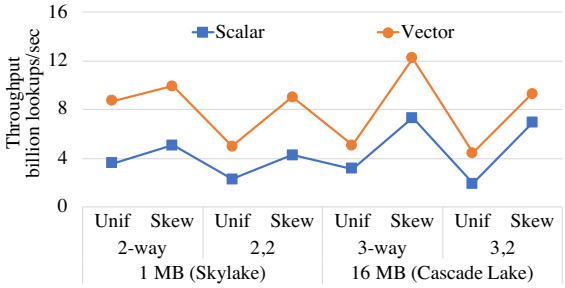


Fig. 9. Case Study⑤: Applying Vertical Vectorization to (2, 2) BCHT on Intel Skylake (1 MB HT size) and (3, 2) BCHT on Intel Cascade Lake (16 MB HT size)

can enable vertical SIMD over BCHT by looping over the 'm' buckets for selective gathers (only gather those keys that have not matched). We attempt to contrast the 2-way cuckoo HT with the (2, 2) BCHT using vertical SIMD, on a node on Cluster A, for 1 MB HT, and 3-way cuckoo HT with (3, 2) BCHT with vertical SIMD on a node on Cluster C for 16 MB HT. From Figure 9, we observe that, while the performance drops by 1.45x when the no. of slots-per-bucket is increased, it can still outperform the corresponding non-SIMD designs.

Thus, SimdHT-Bench's unified benchmark platform enables us to study various use-case scenarios arising in real-world HT workloads, and extend them to emerging CPU architectures.

VI. Benchmark Validation with In-Memory Key-Value Store Use-Case

To validate the applicability of SimdHT-Bench, we present the following key-value store (KVS) use-case. As discussed in Section I, we focus on workloads with 'Multi-Get' requests, i.e., MGet(K1, K2,..,Kn), which batches together access to several key-value pairs, towards parallelizing read operations across the key-value store server cluster.

A. Accelerating KVS Server with SIMD-Aware HT

As shown in Figure 10, the client-to-server pipeline for an MGet operation can be broken down into three basic phases:

1) *Request Phase:* In this phase, each key in MGet(K1,..,Kn) is mapped to a specific Memcached server using consistent hashing, and requests are batched by their respective servers, with key sizes between 200 B to 12 KB.

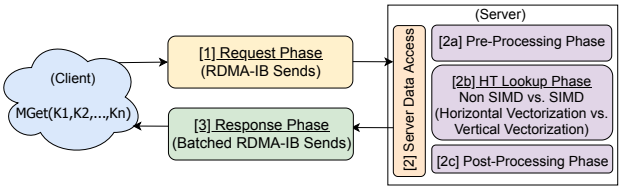
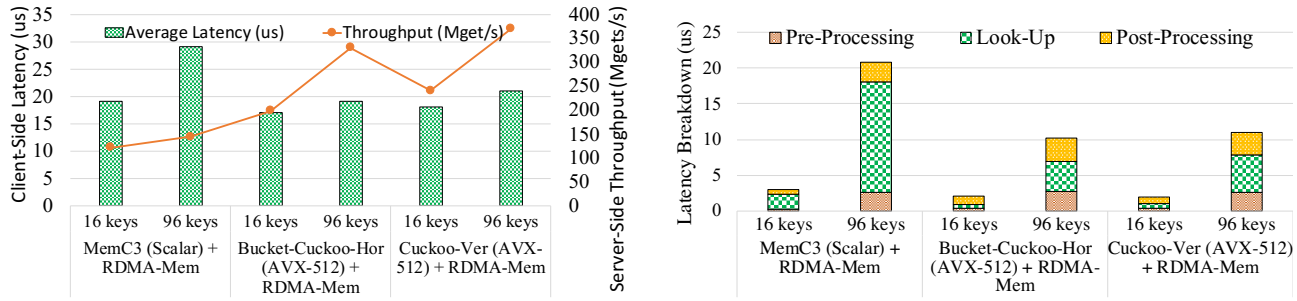


Fig. 10. State-of-the-Art End-to-End Flow for MGet

2) *Server Data Access Phase*: Upon receipt of an 'MGet' request batch from the server's communication engine, the Memcached workers undertake the following steps: *(1) Pre-Processing:* The incoming request of 'N' keys (where 'N' <= mget size 'n') is parsed to extract the individual keys. For each key, a corresponding 32-bit hash value is computed to enable hash table (HT) lookups (i.e., for probing the HT).
(2) Hash Table Lookup: In this phase, the HT is probed to locate the payload (e.g., a key-value pair memory pointer) corresponding to the 32-bit key hash. In this case, we can potentially leverage CPU-SIMD data parallelism to accelerate key lookups. Once probing is successful, the key-value pair identified is located and read from backend memory slabs (or data cache), and is verified against the client-supplied key string to ensure a full match. These matched key-value pairs are returned



(a) End-to-End Multi-Get Latency and Server-Side Get Throughput

(b) Server-Side Timewise Breakdown per Multi-Get request

Fig. 11. Integrating SIMD-Aware Cuckoo Hash Table Variants into RDMA-based Key-Value Store (RDMA-Memcached) [23], [32], [33]; Contrasting with non-SIMD CPU-Optimized MemC3 [12] with Batch-Size (#Keys/Multi-Get) = 16 keys and 96 keys

to the communication engine.

(3) Post-Processing: Once the key-value pairs in the batch are located at the server's memory slabs, the server updates its metadata to maintain cache freshness (e.g., LRU updates for Memcached), and prepares the response (containing located 'value' data per key or `NOT_FOUND`) to be communicated to the client.

3) *Response Phase:* In this phase, the responses are communicated to the server and processed at the client.

With RDMA-Memcached 'Get' protocol, the request/response phases batch the key/value data into multiple small message transfers and communicated to the client using fast two-sided RDMA SENDs. NOTE: The HT key and payload data are different from the actual key-value pair data stored in the server memory, which is typically a variable string of binary data. The HT is indexed to locate this variable-length key-value pair object, based on the hash value of the variable-length key (i.e., hash(key) –>payload –>key-value pair data).

B. Performance Evaluations

Based on the performance studies in Figure 5, we choose to integrate the following two designs into RDMA-Memcached: (a) (2,4) BCHT with horizontal SIMD support, i.e., Bucket-Cuckoo-Hor(AVX-256)+RDMA-Mem, and, (b) 3-way Cuckoo HT with vertical SIMD support over AVX-512, i.e., Cuckoo-Ver(AVX-512)+RDMA-Mem to accelerate 'Multi-Get' workloads. We employ HT key and payload[3] sizes as 32 bits (4 bytes). However, since the key-value store HT lookups need to return an object pointer (64-bit), we use the 32-bit HT payload to index a shared array of object pointers. We contrast this with RDMA-Memcached running with the CPU-optimized non-SIMD MemC3 backend [12], [23], [33], that follows a (2,4) BCHT layout with 8-bit hash keys and 64-bit pointers to key-value pair objects as the value. For our analysis, we use two nodes on Cluster B (see Section V), that is equipped with 28-core Intel Skylake nodes and IB EDR (100 Gbps) interconnects. We undertake this experiment over an RDMA-Memcached server running 26 workers with an HT of size 2 M. We use the "memslap" Multi-Get benchmark [34], configured with 26 clients threads on the client node. We use 20 B keys and 32 B values, with different MGet sizes (i.e., N keys per request = 16 or 64).

From Figure 11(a), we can observe that the SIMD-aware 'Hash Table Look-up' phase gains about 1.45x–2.04x as compared to the non-SIMD MemC3 design in server-side throughput and up to 10%-34% in end-to-end MGet latencies. To get a better perspective, Figure 11(b) presents the server-side latency breakdown, portraying the three sub-phases of the server's data access discussed in Section VI-A. From this figure, it is evident that the data-parallel SIMD-aware HT lookups can reduce the server-side processing time per-batch ('Server Data Access Phase') by up to 50%. We note that the horizontal and vertical approaches do not demonstrate any noticeable performance differences. Upon further analysis, we find that this is due to the overhead of the non-SIMD key matching step in the 'Hash Table Lookup' phase.

Thus, towards efficient co-designing, SimdHT-Bench can help us evaluate the opportunities and applicability of SIMD-aware HT designs for various application scenarios.

VII. Conclusion

In this paper, we present 'SimdHT-Bench', a micro-benchmark suite with a holistic approach to studying the applicability of CPU SIMD-aware hash table designs for varied application workloads. We analyze the design dimensions involved in exploiting vectorization-based parallel key searching over cache-optimized non-SIMD hash tables and study five different use-case scenarios that evaluate varied data access patterns involving read-dominated workloads on latest Intel Skylake and Intel Cascade Lake multi-core CPU nodes. To validate the applicability of SimdHT-Bench, we extend these performance studies to design a high-performance SIMD-aware RDMA-based in-memory key-value store to accelerate the Memcached 'Multi-Get' workload. We demonstrate that our SIMD-integrated designs can achieve about 2x improvement in server-side Get throughput and 34% gain in end-to-end Multi-Get latencies over CPU-optimized non-SIMD designs like MemC3, on an HPC cluster equipped with Intel Skylake CPUs and InfiniBand EDR interconnects.

In the future, we plan to expand our proposed benchmark to study and model mixed workloads that involve concurrent reads and updates to the SIMD-aware hash table. We also plan to extend SimdHT-Bench to other SIMD-friendly hash table designs beyond cuckoo hashing.

REFERENCES

[1] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking SIMD Vectorization for In-Memory Databases," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 1493–1508.

[2] J. Zhou and K. A. Ross, "Implementing Database Operations using SIMD Instructions," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 2002, pp. 145–156.

[3] O. Polychroniou and K. A. Ross, "Vectorized Bloom filters for Advanced SIMD Processors," in *Proceedings of the Tenth International Workshop on Data Management on New Hardware*. ACM, 2014, p. 6.

[4] J. Lu, Y. Wan, Y. Li, C. Zhang, H. Dai, Y. Wang, G. Zhang, and B. Liu, "Ultra-Fast Bloom Filters using SIMD techniques," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[5] K. A. Ross, "Efficient Hash Probes on Modern Processors," in *IEEE 23rd International Conference on Data Engineering (ICDE '07)*. IEEE, 2007, pp. 1297–1301.

[6] T. Behrens, V. Rosenfeld, J. Traub, S. Breß, and V. Markl, "Efficient SIMD Vectorization for Hashing in OpenCL," *Positions*, vol. 3, no. 4, 2018.

[7] T. Gubner and P. A. Boncz, "Exploring Query Compilation Strategies for JIT, Vectorization and SIMD," in *ADMS@VLDB*, 2017.

[8] N. L. Scouarnec, "Cuckoo++ Hash Tables: High-Performance Hash Tables for Networking Applications," in *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*. ACM, 2018, pp. 41–54.

[9] "Hash Library - Documentation - DPDK," https://doc.dpdk.org/guides/prog_guide/hash_lib.html.

[10] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast Remote Memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 401–414.

[11] "Memcached: High-Performance, Distributed Memory Object Caching System," http://memcached.org/, 2003.

[12] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 371–384.

[13] Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Characterizing Facebook's Memcached Workload," *IEEE Internet Computing*, vol. 18, no. 2, pp. 41–49, 2014.

[14] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 385–398. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[15] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-Scale Key-Value Store," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1. ACM, 2012, pp. 53–64.

[16] "Redis," https://redis.io/.

[17] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, High Performance Ethernet Forwarding with CUCKOOSWITCH," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. ACM, 2013, pp. 97–108.

[18] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: A Memory-efficient, High-performance Key-value Store," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, ser. SOSP '11, Cascais, Portugal, October 2011, pp. 1–13.

[19] J. Doweck, W. Kao, A. K. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake," *IEEE Micro*, vol. 37, no. 2, pp. 52–62, March 2017.

[20] M. Arafa, B. Fahim, S. Kottapalli, A. Kumar, L. P. Looi, S. Mandava, A. Rudoff, I. M. Steiner, B. Valentine, G. Vedaraman *et al.*, "Cascade Lake: Next Generation Intel Xeon scalable processor," *IEEE Micro*, vol. 39, no. 2, pp. 29–36, 2019.

[21] R. Pagh and F. F. Rodler, "Cuckoo Hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[22] U. Erlingsson, M. Manasse, and F. McSherry, "A Cool and Practical Alternative to Traditional Hash Tables," in *Proceedings of the 7th Workshop on Distributed Data and Structures (WDAS'06)*, 2006.

[23] D. Shankar, X. Lu, N. Islam, M. Wasi-Ur-Rahman, and D. K. Panda, "High-Performance Hybrid Key-Value Store on Modern Clusters with RDMA Interconnects and SSDs: Non-blocking Extensions, Designs, and Benefits," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 393–402.

[24] NOWLAB, "High-Performance Big Data (HiBD)," http://hibd.cse.ohio-state.edu, 2019.

[25] S. Blanas, Y. Li, and J. M. Patel, "Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 2011, pp. 37–48.

[26] R. Panigrahy, "Efficient Hashing with Lookups in two Memory Accesses," *CoRR*, vol. cs.DS/0407023, 2004. [Online]. Available: http://arxiv.org/abs/cs.DS/0407023

[27] M. Mitzenmacher, "Some Open Questions Related to Cuckoo Hashing," in *European Symposium on Algorithms*. Springer, 2009, pp. 1–10.

[28] M. Pilman, K. Bocksrocker, L. Braun, R. Marroquín, and D. Kossmann, "Fast Scans on Key-Value Stores," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1526–1537, 2017.

[29] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang, "Mega-KV: A Case for GPUs to Maximize the Throughput of In-Memory Key-Value Stores," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1226–1237, 2015.

[30] A. D. Breslow, D. P. Zhang, J. L. Greathouse, N. Jayasena, and D. M. Tullsen, "Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, 2016, pp. 281–294.

[31] Mutilate: High-Performance Memcached Load Generator, https://github.com/leverich/mutilate.

[32] X. Lu, D. Shankar, and D. K. Panda, "Scalable and Distributed Key-Value Store-based Data Management Using RDMA-Memcached," *IEEE Data Eng. Bull.*, vol. 40, no. 1, pp. 50–61, 2017.

[33] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur *et al.*, "Memcached design on high performance rdma capable interconnects," in *2011 International Conference on Parallel Processing*. IEEE, 2011, pp. 743–752.

[34] "memslap," http://docs.libmemcached.org/bin/memslap.html.