Analyzing Vectorized Hash Tables Across CPU Architectures

Conference Paper

Author(s):
Böther, Maximilian ; Benson, Lawrence; Klimovic, Ana; Rabl, Tilmann

Publication date:
2023-07

Permanent link:
https://doi.org/10.3929/ethz-b-000624893

Originally published in:
Proceedings of the VLDB Endowment 16(11), https://doi.org/10.14778/3611479.3611485

Funding acknowledgement:
204620 - MLin: Machine Learning Input Data Processing as a Service (SNF)
957407 - Integrated Data Analysis Pipelines for Large-Scale Data Management, HPC, and Machine Learning (EC)

Analyzing Vectorized Hash Tables Across CPU Architectures

Maximilian Böther
ETH Zurich
Zurich, Switzerland
mboether@inf.ethz.ch

Lawrence Benson
Hasso Plattner Institute
Potsdam, Germany
lawrence.benson@hpi.de

Ana Klimovic
ETH Zurich
Zurich, Switzerland
aklimovic@ethz.ch

Tilmann Rabl
Hasso Plattner Institute
Potsdam, Germany
tilmann.rabl@hpi.de

ABSTRACT

Data processing systems often leverage vector instructions to achieve higher performance. When applying vector instructions, an often overlooked data structure is the hash table, even though it is fundamental in data processing systems for operations such as indexing, aggregating, and joining. In this paper, we characterize and evaluate three fundamental vectorized hashing schemes, vectorized linear probing (VLP), vectorized fingerprinting (VFP), and bucket-based comparison (BBC). We implement these hashing schemes on the x86, ARM, and Power CPU architectures, as modern database systems must provide efficient implementations for multiple platforms due to the continuously increasing hardware heterogeneity. We present various implementation variants and platform-specific optimizations, which we evaluate for integer keys, string keys, large payloads, skewed distributions, and multiple threads. Our extensive evaluation and comparison to three scalar hashing schemes on four servers shows that BBC outperforms scalar linear probing by a factor of more than 2x, while also scaling well to high load factors. We find that vectorized hashing schemes come with caveats that need to be considered, such as the increased engineering overhead, differences between CPUs, and differences between vector ISAs, such as AVX and AVX-512, which impact performance. We conclude with key findings for vectorized hashing scheme implementations.

PVLDB Reference Format:
Maximilian Böther, Lawrence Benson, Ana Klimovic, and Tilmann Rabl. Analyzing Vectorized Hash Tables Across CPU Architectures. PVLDB, 16(11): 2755 - 2768, 2023. doi:10.14778/3611479.3611485

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/hpides/vectorized-hash-tables.

1 INTRODUCTION

The heterogeneity of computing systems is steadily increasing. While Intel and AMD x86 CPUs have been dominating the market in the last decades, ARM-based CPUs are gaining traction and are expected to reach 22 % in cloud deployments by 2025 [78]. Many companies, e.g., Nvidia, Amazon, or Ampere, are building custom ARM chips [2, 56, 79], and due to Apple M-series processors [3], ARM already has 7 % market share in the customer segment [70]. Furthermore, IBM is continuously improving the Power architecture and released Power10 in 2021 [67].

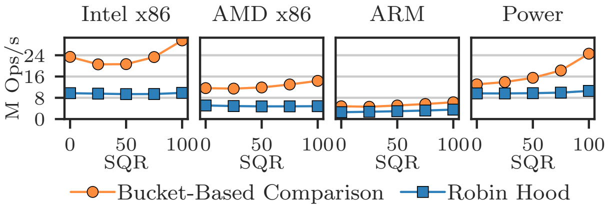

Figure 1: Lookup performance of best scalar (RH) and vectorized (BBC) hashing schemes for load factor 90 %.

Computer scientists have built many systems to process the gigantic volume of data produced steadily [53]. These systems are increasingly designed with a focus on hardware efficiency [14, 30, 77, 83]. They leverage modern hardware capabilities, such as modern storage tiering [43, 58], target-specific code generation [29, 58, 59], or CPU-GPU co-processing [19] to achieve maximum performance.

A fundamental data structure used in data processing systems, whose efficient implementation and performance depends on the hardware platform, is the hash table. Hash tables are used in many different operations, including indexing, aggregating, and joining. In this paper, we focus on fixed-size hash tables, as their performance is critical in, e.g., equi-joins [42]. The majority of previous work on the join operator has focused on partitioning strategies for parallelization [7–10, 16, 37, 42, 52]. However, hash table implementations have only received limited attention [66]. In particular, unlike parallelization, the design and implementation strategies for hash tables have not been characterized in the context of modern CPU hardware. While some prior work discusses ideas on vectorizing hash tables [13, 63, 65, 66], to the best of our knowledge, we are the first to implement, compare, and benchmark a wide variety of approaches and provide empirical insights to guide the design of vectorized hash tables on modern CPUs. For example, some works compare the hash table keys using vector instructions [63, 66], while other works introduce fingerprints, i.e., short hashes of keys, to compare more items at once [13], but these approaches have never been compared in a coherent setup.

In this paper, we provide insights into vectorization for hash table designs and how the underlying hardware influences the performance. We investigate whether insights we obtain on x86 CPUs also hold on ARM or Power CPUs, how to design vectorized hashing schemes depending on the vector instruction set architectures, and what impact the vector register size has on performance. In Figure 1, we show that our best performing vectorized implementation (Bucket-Based Comparison) outperforms the best scalar hashing scheme (Robin Hood) by 1.4x to 3x (2.1x on average) for high load factors. In this paper, we perform a detailed step-by-step analysis from scalar to fully vectorized hashing schemes, showing which vectorization design choices impact performance on three common CPU architectures. Overall, our contributions are as follows.

Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611485

- We systematically characterize and discuss three approaches to vectorized hashing from previous work that build on each other increasingly adjusted to vectorization: vectorized linear probing (VLP), vectorized fingerprinting (VFP), and bucket-based comparison (BBC).
- We implement these hashing schemes using six different vector instruction set architectures (ISAs) on x86 (SSE, AVX2, AVX-512), ARM (NEON and ARM), and Power (VSX). We discuss implementation variants and platform-specific optimizations.
- We conduct an extensive evaluation of insert and lookup performance of scalar and vectorized hashing schemes on four servers. We derive well-performing algorithmic variations for the hashing schemes, discuss the impact of vector register size, and compare our results across the systems.

We find that all platforms benefit from using vectorized hashing schemes, but the relative performance speedups differ between platforms. The naive vectorization of linear probing, does not perform well, showing that vectorization requires careful consideration.

2 BACKGROUND

In this section, we present background on in-memory hash tables (Section 2.1) and CPU architectures (Section 2.2).

2.1 Hash Tables

Hash tables are fundamental data structures that allow users to store key-value pairs (k, v) and retrieve the value v for a key k in constant lookup time $O(1)$. In-memory hash tables use a *hash function* h that maps a key k to an index or *slot* within an allocated block of memory. In case of conflicts (*collisions*), the *hashing scheme* determines how the the conflict is resolved. If the hash table can store n elements in its memory block, and x elements are currently inserted, we refer to $l = x/n$ as the *load factor* of the hash table. We call n the *capacity* and x the *size* of the hash table. Despite the asymptotic constant lookup and insertion time, the real-world run time of the hash tables operations depends on the number of collisions. A simple way to reduce the number of collisions is increasing the size of the allocated memory block, thereby decreasing the load factor l. This leads to fewer collisions and shorter conflict resolution probes.

2.1.1 Hash Functions. The hash function has a significant impact on the performance of a hash table. It has to balance computational cost and good hash randomization. To determine the slot for a key, the hash function must perform two steps [39]. In the first step, it maps the arbitrary key to a hash value $x \in 2^n$. In the second step, it applies a *finalizer*, which maps x to the range of the underlying memory block, which is $< 2^n$. To efficiently finalize a hash value without an expensive modulo operation, hash tables are commonly sized as a power of two. This allows us to determine the n bits for the slot via the cheap bitmask `x & size-1`.

2.1.2 Scalar Hashing Schemes. The hashing scheme describes how the table deals with hash collisions, i.e., when two keys get mapped to the same slot. We differentiate between chaining, i.e., all items that map to the same slot are stored in a linked list, and open addressing approaches, which iterate through the hash table until the next free slot is found. For a key k and hash function h, we set $i = h(k)$ as the index or slot within the table that the key is assigned to. The *displacement* describes the number of steps needed to find the slot holding the key. In the j-th probe iteration, we denote the current probe index as i_j, i.e., $i_0 = i$. Further schemes include Hopscotch hashing [32] and Cuckoo hashing [62].

Linear Probing. Given a key k and a hash function h, linear probing (LP) [38] determines the slot $i = h(k)$. For inserts, if the slot i is already occupied, we advance in the hash table, i.e., $i_{j+1} = i_j + 1$, and check the next slot. We probe in a circular fashion, such that if we reach the end of the hash table, we set $i = 0$. We probe until we find the first free slot. Lookups follow a similar algorithm, but instead of checking whether a slot is free, we check whether the key in the current slot i equals the key k we are searching for.

Linear probing is commonly used because of its simplicity. It allows for good cache locality and can be pipelined well by the CPU due to its concise implementation. However, for higher load factors, when many slots are full, we encounter long probing sequences.

Robin Hood. Robin hood (RH) hashing organizes the hash table such that probing chains stay small [23]. During inserts, we probe as in linear probing, but whenever the displacement of an element is larger than the displacement we are currently inserting, we swap the elements and continue. This avoids that the displacement of the elements to insert continues to increase, and instead allows the smaller displacement of the swapped elements to grow. RH uses the fact that any key in the probing sequence just needs to be *somewhere* along the sequence, not at its specific location assigned by LP. The lookup algorithm is analogous to LP.

Chained Hashing. Chained hash tables consist of a directory and buffer. The directory stores pointers, the buffer stores keys, values, and a next pointer. For inserts, we add the key-value pair to the buffer and set the pointer in the corresponding directory slot, while updating the new next pointer to the old location, i.e., chaining the entries. For lookups, we obtain the directory index and follow the pointers until we find the key or an invalid pointer.

2.2 SIMD Programming and CPU Architectures

In the following, we discuss CPU architectures used in this work.

x86-64. The x86 instruction set architecture (ISA) is the most prominent architecture for consumer platforms and off-the-shelf servers [36]. We differentiate between three major revisions of the x86 SIMD ISA, SSE4, AVX2, and AVX-512. SSE provides 16 128-bit vector registers and AVX2 has 16 256-bit vector registers. As x86 aims to be backward compatible, the lower 128 bits of the 256-bit registers can be used as SSE registers, such that programs that use SSE instructions still function on AVX2 CPUs without adaptation. AVX-512 is the latest major SIMD extension. It features 32 512-bit vector registers, from which the first 16 are backward compatible with AVX2 and SSE. AVX-512 introduces *opmask registers*. The opmask registers allow to control which elements take part in the next instruction. The AVX-512 VL Extension allows users to operate on 128-bit and 256-bit vector registers with support for opmasks.

For 128-bit registers (SSE), the data needs to be aligned to a multiple of 16 Byte; for 256-bit registers (AVX2), to a multiple of 32 Byte; and for 512-bit registers (AVX-512), to a multiple of 64 Byte [35]. x86 also supports explicit unaligned loads and stores.

ARM. The ARM architecture is designed and developed by ARM Limited. The ARM ISA can be licensed to develop CPUs. The most recent ISA version is ARMv9. ARM is the market leader for mobile devices [27] and powers the second fastest supercomputer [76].

Since ARMv8-A, the SIMD extension NEON is standard for all ARM CPUs. NEON features 16 128-bit vector registers. In 2016, ARM presented a new SIMD extension called Scalable Vector Extension (SVE) [74]. SVE features a novel *vector-length agnostic* programming model, meaning that the developer compiles applications once that then can be run on CPUs with vector register sizes up to 2048-bit [75]. SVE comes with 32 vector registers. The programmer and compiler do not have access to the register size at compile time, which reduces optimization potential. Similar to AVX-512, SVE implements *predicate registers*, allowing to select individual bytes of an SVE register (masking). As of 2022, SVE has been implemented by the Fujitsu A64FX CPU and the ARM Neoverse N2 and V1 CPUs. All chips implementing SVE are backward compatible with NEON. ARMv9 includes SVE2 as an addition to SVE [25, 68]. While SVE was built for HPC and machine learning applications, SVE2 is a functional general-purpose SIMD extension. ARM Limited expects NEON to be deprecated when ARMv9 becomes mainstream [6].

The ARM NEON and SVE specifications does not enforce any alignment on load and store operations. Processors implementing NEON often give hints in their handbooks on how to align the data for optimal performance [5], while SVE load and store operations should generally be 16-byte aligned [4] for best performance.

Power. The Power architecture is a RISC architecture designed and developed by IBM and the OpenPower Foundation. The ISA was made publicly available in 2019 and completely license-free in 2020 [17, 18]. The most recent Power CPU is the Power10 CPU, which was released in 2021 [67]. Power CPUs focus more on scale-up instead of scale-out [72], i.e., maximizing performance of individual compute nodes. With the integration of both on-chip accelerators, such as *matrix-math assist* engines, and of cache-coherent FPGAs using OpenCAPI, Power is an interesting alternative for data processing. Power currently implements the Vector Scalar Extensions (VSX) in version VSX-3. It supports 64 128-bit vector registers. There are no masking registers. On current Power CPUs, data should be 16 byte aligned for better performance [33]. Since POWER7, unaligned loads are supported as well.

3 HASHING SCHEME DESIGN

In this section, we describe implementation details of scalar hashing schemes (Section 3.1), and discuss the vectorized hashing schemes (Section 3.2). We follow the assumptions made for equi-joins in some data processing systems, i.e., the hash table size is fixed and we do not need to rehash [10, 42]. We briefly discuss how to implement the schemes for different architectures (Section 3.3). All details of our C++ implementation can be found in our GitHub repository.

3.1 Scalar Hashing Schemes

In this subsection, we discuss different memory layouts and implementation details for scalar hashing schemes. We evaluated manual loop unrolling and prefetching during a probe sequence, but did not find any performance improvements.

3.1.1 Memory Layout. For each hash table slot, both linear probing and robin hood store a key, a value, and the validity information, i.e., a field indicating whether a slot holds valid data. In previous work, this is often represented as a zero key. However, this hinders applicability of these approaches, because in real-world systems, we cannot pre-define certain keys as invalid if we do not control the user input key space. For linear probing and robin hood, we can either use an array-of-structs (AoS) or struct-of-arrays (SoA) layout. In an AoS layout, the hash table holds a single array of keys, values, and the validity information. The validity information requires a single bit, but due to struct padding, the memory usage of the hash table increases significantly. We disable struct padding using the compiler packed attribute, as it requires less memory and performs better or similar in our experiments. In an SoA layout, the hash table holds one array per field of an entry to avoid loading unnecessary data while probing, i.e., all keys are in a separate array, all values in a separate array, and all validity flags in a separate array. As an optimization, we store the key and validity information in a single array, as they are always needed together, avoiding two random memory accesses per probe. AoS and optimized SoA are depicted in Figure 2. We use huge pages, as our experiments show that they outperform regular pages due to reduced TLB misses [46, 61]. Note that when memory is unconstrained, the load factor can be arbitrarily chosen, such that LP becomes the optimal hashing scheme. Hence, we want to maximize performance while minimizing memory usage.

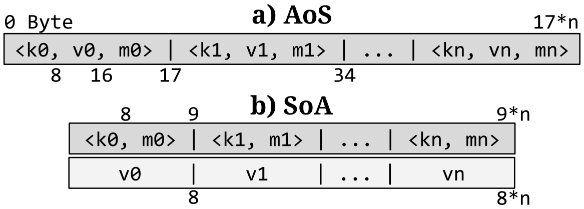
0 Byte
a) AoS
17*n
<k0, v0, m0> | <k1, v1, m1> | ... | <kn, vn, mn>
8
16
17
34
8
9
b) SoA
9*n
<k0, m0> | <k1, m1> | ... | <kn, mn>
v0 | v1 | ... | vn
8
8*n

Figure 2: Memory layout of AoS and SoA. *n* entries with 1-Byte validity metadata (m) and 8/8-Byte key (k)/value (v).

3.1.2 Robin Hood Details. During a lookup probe, RH aborts early if our current displacement becomes larger than the displacement of the current probe index (Section 2.1.2). This can be done either by *(1)* storing displacement information (*StoringRH*) or by *(2)* recalculating the displacement information (*RecalcRH*). We investigate the recalculating version since the memory overhead of storing is high. In the worst case, the displacement can be as large as the capacity of the hash table, meaning that a 64-bit integer is required to store and use it correctly. We use a branchless displacement recalculation that considers potential wrap-arounds during probing.

3.1.3 Chained Details. For chained hashing, given a memory budget, we have to determine how to size the directory and the buffer. Given the hash table size 2^n, we calculate the budget as 110 % of the memory required for a linear probing table of the same size, as in previous work [66]. We first determine how much memory the buffer requires, i.e., how much memory is required to store the entries that will be inserted. Then, we find the largest n' such that the directory with $2^{n'}$ entries, together with the buffer, still fits within in the budget, i.e., n' can be larger, smaller, or equal to n.

3.2 Vectorized Hashing Schemes

We identify three vectorized hashing schemes, vectorized linear probing (Section 3.2.1), vectorized fingerprinting (Section 3.2.2), and bucket-based comparison (Section 3.2.3) that have been described in previous work. They naturally extend and build upon each other, increasingly adjusted to vectorization. While vectorization of other hashing schemes such as Cuckoo and Hopscotch hashing has been considered [31, 71], these works show that the benefit of vectorization is low and they fall into a separate family of hashing schemes. Previous work and open-source implementations often follow one of the schemes discussed in the following.

3.2.1 Vectorized Linear Probing. The *vectorized linear probing* (VLP) hashing scheme [66] is a vectorization of the scalar linear probing hashing scheme. It only works for integer keys and requires an SoA memory layout with three arrays for keys, values, and the validity bits. We first determine the key's slot. Then, we load as many keys as can fit into a vector register and compare all of them at once to the key we are searching for. If the key is among the keys we compared, we obtain the index and return. Otherwise, we load the validity information into a vector register and check whether we have reached an invalid entry. If we have reached one, we terminate the search. Otherwise, we continue with the next set of keys. To terminate, we have to check the validity in each probe.

The first loads of keys and validity information might be unaligned because the first index is not guaranteed to lie at an aligned address. However, when iterating forward, we can make sure that we only perform aligned loads, by taking a smaller step in the first iteration to the next aligned address.

We propose two variants of VLP that differ in how they perform the comparison. In the first variant (*TEST*), we first test if there is *any* match for the key we are searching for, and only if there was a match, we extract the first matching index using further instructions. In the second variant (*NOTEST*), we skip the check whether there is any match, and just extract the first matching index, which might be an invalid index in case there was no match.

3.2.2 Vectorized Fingerprinting. VLP is a direct vectorized extension of linear probing but it comes with several drawbacks. It works only on integer keys, it has a low degree of parallelism, which depends on the key size, and it requires access to three arrays. To address these issues, previous work uses bucket-based comparison (BBC) as described in Section 3.2.3. As the introduction of BBC changes multiple dimensions of the hashing scheme at once, i.e., memory layout, fingerprinting, and key grouping, we present vectorized fingerprinting (VFP) as an intermediate step to isolate the reasons for potential performance gains. Instead of loading and comparing the keys directly in vector registers, we operate on fingerprints, i.e., 8- or 16-bits key hashes, and only perform conflict resolution in case of a match. VFP requires the SoA layout only for fingerprints and can use AoS for higher cache locality on the other fields (c.f. Figure 2). Fingerprints allow us to *i)* work independently of the key data type, as everything is either an 8-bit or 16-bit integer fingerprint, *ii)* compare more information at once because the fingerprints are smaller than 32-bit or 64-bit keys, *iii)* remove the extra validity information array by defining a certain fingerprint as *invalid*[1], e.g., 0, and, *iv)* store keys and values in the same array.

The algorithm itself works similar to VLP, but uses fingerprints instead. We maintain one array of fingerprints and another array of key-value pairs. To find a key, we first compute the fingerprint f and the index into the table. We load the fingerprints starting at the index and compare them against f. For each match, we compare the actual keys. If we find a key match, we return. Otherwise, we continue with the next fingerprints until we find a match or an invalid slot. The first load might be unaligned.

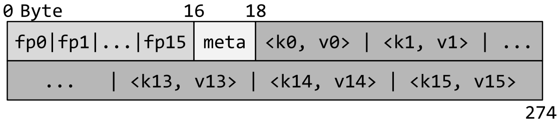
0 Byte
16
18
fp0|fp1|...|fp15
meta
<k0, v0> | <k1, v1> | ...
... | <k13, v13> | <k14, v14> | <k15, v15>
274

Figure 3: Memory layout of Bucket-Based Comparison. 16 entries with 1-Byte fingerprints and 16-Byte key/value pairs.

We evaluate both 8-bit and 16-bit fingerprints. 8-bit fingerprints allow for 256 unique values, and 16-bit fingerprints allow for 65 536 unique values. This lowers the chance of collisions exponentially. However, we can compare twice as many fingerprints in the same vector register when using 8-bit instead of 16-bit, and use half additional memory. We use either the most significant bits (MSB) or least significant bits (LSB) as the fingerprint. We do not investigate larger fingerprints since the benefits diminish. The memory footprint of fingerprints is small, compared to, e.g., StoringRH, since we only require 1-2 additional bytes per slot, leading to an increase by ~6 % for 8-bit fingerprints with 8-byte keys/values. For larger payloads, this expansion factor is even smaller.

3.2.3 Bucket-Based Comparison. The previous hashing schemes assume a large, contiguous block of memory through which we iterate. *Bucket-based comparison* (BBC) represents the hash table as an array of buckets [21, 28, 48]. As shown in Figure 3, a bucket holds fingerprints, metadata, and key-value pairs. BBC has three main advantages over VFP. First, we avoid the additional latency to obtain the key-value-pair, as the fingerprints and data are colocated. Second, we store a flag that indicates whether a bucket has overflowed, which offers opportunity for early termination. Third, BBC separates the interfaces of the hash table and buckets, making the implementation simpler and concise.

We maintain an array of as many buckets as we need to hold the requested amount of key-value-pairs (capacity). A bucket stores the next free index, a boolean flag whether the bucket has overflowed, i.e., we tried to insert a key into the bucket, but the bucket was already full, an array of fingerprints, and an array of key-value-pairs. The operations on BBC are nearly identical to VFP. Based on the key, its fingerprint and bucket index, we check if the bucket contains the key by comparing the fingerprints and then the keys when we find a match. If the bucket has overflowed, we continue with the next bucket. The two main differences compared to VFP are that the fingerprints are always aligned in a bucket, and that we can terminate the lookup if a bucket has not overflowed compared to scanning for validity information. We define the buckets to hold as many fingerprints as fit the SIMD register. On platforms such as x86 where we have SIMD registers of different sizes, it is unclear whether we should always use the largest register available or not. We evaluate all possible vector register/fingerprint combinations.

3.3 Building a SIMD Abstraction Layer

The hashing schemes perform common operations that need to be implemented on multiple ISAs. Existing libraries, e.g., xSIMD [82]

[1] This was not possible previously as the key range is user-defined, while the fingerprint range is application-defined.

Table 1: Num. of cores, base/turbo frequencies, cache, cache line, and maximum vector register sizes for our systems.

System	CPU	# Cores	# Threads	Freq. (GHz)	L1	L2	L3	CL Size	max. VR
Fujitsu RX2530	Xeon 5220S	18	36	2.7/3.9	576 KiB	18 MiB	24.75 MiB	64 B	512-bit
HPE XL225n	EPYC 7742	64	128	2.3/3.4	2 MiB	32 MiB	256 MiB	64 B	256-bit
HPE Apollo 80	A64FX	48+4	48	1.8/1.8	3 MiB	32 MiB	-	256 B	512-bit
IBM IC922	POWER9	16	32-128	3.4/4	512 KiB	8 MiB	120 MiB	128 B	128-bit

or TVL [80], abstract from ISAs. However, we use a custom abstraction because these libraries do not provide the required level of abstraction, e.g., logical operations, such as *extracting a matching index*. Determining a common abstraction is a trade-off, as higher abstractions are easier to maintain, but lower abstractions allow for more optimization to the hardware. Our evaluation shows that these different abstractions have an impact on performance.

For example, in VLP, we are interested in extracting the first match of the key in the register. For this, we compare two vectors and get a result where each entry represents whether there has been a match or not. In x86, we then create a *movemask*, i.e., an integer representation of this vector, on which we use scalar operations to determine the index. Compared to SSE/AVX2, AVX-512 does not require an explicit movemask instruction.

On NEON, there is no movemask instruction. We test two alternatives to extract the first match. Alternative 1 is using an emulation of the SSE movemask instruction by the SSE2NEON library [73]. Alternative 2 is the UMINV approach based on the `UMINV` instruction, which returns the minimum value in a vector. We use the comparison result as the input to a bitwise select operation on an *index vector* containing indices and an *invalid vector* containing a number representing an invalid entry. If we then extract the minimum from the resulting vector, we obtain the first index within the vector where there has been a match. On VSX, there is an intrinsic that implements this entire operation. This shows that identifying logical operators for the abstraction layer is important.

In VFP and BBC, we iterate over all matches. In the evaluation, on Power and NEON, we compare implementing a scalar iterator based on a (simulated) movemask to a vectorized iterator, which operates on a register instead.

4 EVALUATION

In this section, we perform experiments to analyze the read and write performance of the hash tables in various settings. We discuss the scalar hash tables in Section 4.2 and the vectorized hashing schemes that are increasingly adjusted to vectorization in Sections 4.3 to 4.5. Our stepwise vectorization evaluation shows which changes yield how much benefit, providing insights into which aspects developers should focus on when designing vectorized hash tables. We focus on key implementation choices to guide developers when implementing vectorized hash tables. Once all schemes are established, we evaluate various setups to cover a wide range of hash table performance. We compare string and integer keys (Section 4.6), discuss large payloads (Section 4.7), and show the impact of skewed requests (Section 4.8). Last, we evaluate multiple threads (Section 4.9). We use the four servers shown in Table 1.

x86. We use two x86 systems, Intel and AMD. Our Intel x86 server contains two 18-core Intel Xeon Gold 5220S Cascade Lake CPUs that support AVX-512. Our AMD x86 server contains two 64 core AMD EPYC 7742 Zen2 CPUs that support AVX2.

ARM. Our ARM node contains one Fujitsu A64FX CPU [54, 60]. The A64FX has 48 compute cores and 4 *assistant cores* at 1.8 GHz. It supports SVE with a maximum register length of 512-bit.

Power. We run our Power experiments on an IBM Power System IC922. The IC922 system is optimized for high-performance data analytics and comes with two POWER9 16-core CPUs [34].

4.1 Setup

For the read throughput benchmarks, we fill up the hash table up to a certain load factor (LF) $l \in [25\,\%, 50\,\%, 70\,\%, 90\,\%]$ with keys and values generated uniformly at random. Then, we generate a set of keys (*queries*) uniformly at random, some of which are in the hash table and some of which are not. We are interested in the read throughput of the hash table, i.e., the number of lookups per second, for this set of keys. The fraction of keys we query that have been inserted into the table is called *successful query rate* (SQR). Except for Section 4.9, all experiments are run single-threadedly. Unless stated otherwise, we choose our hash tables to hold 2^{27} keys and use 64 bit integers as keys and values as representatives of common primitive data types. Below, we show the different tables' memory consumption for 16-Byte key-value pairs.

LP	**RecalcRH**	**VLP**	**VFP** (8/16-bit FPs)	**BBC** (8/16-bit FPs)
2.13 GiB	2.13 GiB	3 GiB	2.13/2.23 GiB	2.25/2.5 GiB

In our evaluation, we focus on high load factors, as those balance memory consumption and performance. Given enough memory, most hashing schemes converge towards perfect hashing, removing the necessity for collision handling, i.e., most lookups are answered directly from the first slot. In this case, simple schemes such as linear probing or chaining perform best due to simpler lookup code. However, this comes at a high memory cost, which most systems try to avoid. In the design of hash tables, it is important to achieve high performance while also consuming little memory.

For the write microbenchmarks, we start with an empty hash table with a capacity of 2^{27} keys. For the load factors $l \in [25\,\%, 50\,\%, 70\,\%, 90\,\%]$, we insert $l \cdot 2^{27}$ keys and measure the number of insert operations per second. We use 64-bit multiply-shift (MS) hashing [24], as it was shown to have highest throughput [66]. We also tested MurmurHash3 and xxHash on both uniform and dense key distributions, but they did not bring performance improvements. MS is a simple hash function that consists of a multiplication with a constant and bit shifting. It does not require a finalizer, as the shift operation already maps the value into the correct range. We follow the SMHasher implementation [81]. We perform one warmup measurement and report the average of three measurements. We prefault each hash table before we benchmark, i.e., we write data to each index to avoid measuring the initial page fault overhead. The Intel and AMD x86 nodes run Ubuntu 20.04 LTS with Linux kernel 5.4. The Power and the A64FX systems run RHEL 8.3 with Linux kernel 4.18. We use gcc 11.2 with the `-O3` and native flags.

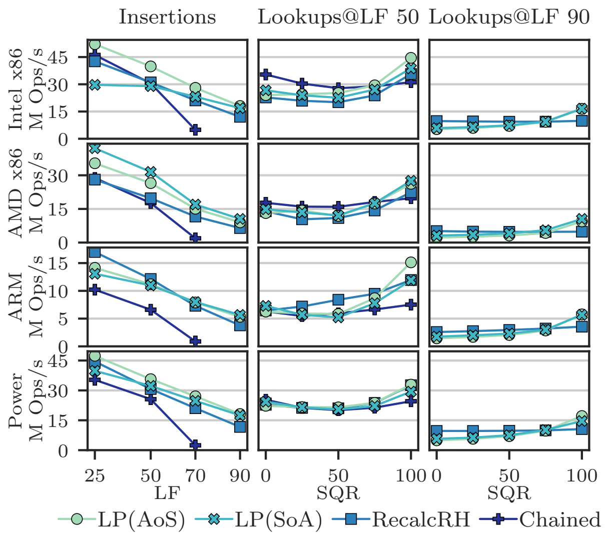

Figure 4: Performance of linear probing (*LP*) with AoS and SoA layouts, robin hood (*RecalcRH*), and chained hashing.

4.2 Scalar Baseline

In our first experiment, we establish the performance of scalar hashing schemes. Based on these results, we determine linear probing as our baseline for further evaluation. In Figure 4, we show the linear probing hash table with an AoS layout, with an SoA layout, the robin hood hash table, and the chained hash table.

SoA vs AoS. The SoA memory layout is the layout we use for the vectorized hashing schemes and requires one additional random memory access when packing validity information and key, but has higher information density during probing. In general, as discussed by Richter et al. [66], the shorter the probe sequence, the better the AoS layout performs. Longer probe sequences favor the SoA layout because the longer the sequence, the more unnecessary values we load in the AoS layout. For Intel, AMD, ARM, and Power, SoA has 1.05x, 1.29x, 1.11x, and 1.05x higher throughput than AoS.

We see that SoA performs good in low SQR/low LF scenarios as well. At LF 25 % and SQR 0 %, the SoA version has 1.19x the performance of AoS across all systems. At 100 % SQR, however, AoS has a 1.17x higher throughput. The reason for this is not only the shorter probing sequence itself. We have to consider that for unsuccessful queries in an SoA layout, we avoid the additional random DRAM access. For example, for SQR 0 %, we have the same number of random DRAM accesses in SoA and AoS, but a tighter loop and fewer unnecessary values loaded. To show this, we measure the number of probes per lookup operation. At SQR 0 %, we measure 1.39 probes per lookup. At SQR 100 %, we measure 1.17 probes per lookup. This only makes a difference of 0.22 probes. For LF 50 % and SQR 25 %, AoS is superior to SoA, and we measure 2.25 probes per lookup. These are more probes than at LF 25 % and SQR 0 %, where SoA is superior. This shows that the probing length itself is not the reason why SoA is superior to AoS for small SQRs.

However, the performance gains in SoA-favorable settings are smaller than the performance penalties when using SoA in AoS-favorable settings. Hence, in general purpose applications where the SQR is not known before, the AoS memory layout is preferable.

Robin Hood. For the recalculating robin hood hashing, the memory layout is the same as for linear probing. We find that it performs better than LP for high LF scenarios due to the shorter probe sequences. For short probe sequences (LF ≤ 50 %), the overhead of the additional checks leads to worse performance than LP.

Chained. The chained hash table has a memory budget for the directory and the buffer (c.f. Section 2.1). For LF 90 %, we exceed the memory budget. For LF 70 %, the performance degrades as the directory resizes to 2^{23}, leading to more collisions and pointer chasing. Hence, the chained table can only be used with a comparable memory footprint for LFs 25 % and 50 %. For LF 50 % and SQR 0 %, it has a 1.47x/1.35x/1.01x/1.14x higher throughput than AoS LP on Intel x86, AMD x86, A64FX, and Power. At SQR 100 %, LP performs better than chaining. For inserts, chaining is consistently outperformed by all other scalar hashing schemes.

4.3 Vectorized Linear Probing

Based on our scalar results, we show the performance of vectorized linear probing (VLP) as the naive vectorized extension of LP, as proposed in previous work [66]. Our results show that this direct change does not yield good performance, which is why show improvements of VLP using fingerprints (Section 4.4) and buckets (Section 4.5). Overall conclusions about the benefits of vectorization should be drawn only after the improvements. VLP operates on three arrays (keys, values, and validity information), and the validity information is not packed into one byte but instead stored in the key's type to match the key layout.

4.3.1 Implementation Choices. We first investigate implementation choices to provide hands-on insights into the impact of possible implementation variants, which can be directly applied by developers. The TEST variant of VLP checks if there is any match in the key vector before extracting the first match. The NOTEST variant always extracts the first match, which might be invalid. For all load factors, the NOTEST performance is worse than the TEST performance and drops even further for lower SQR. For example, for LF 50 % and SQR 0 %, the TEST variant has a 1.11x higher throughput averaged across all servers. Lower SQRs imply a lower probability for a match within a single key vector. Hence, testing for any match in the vector improves performance if extracting the match costs more than the check. As the NOTEST throughput is lower than or equal to the TEST throughput, the check has no measurable overhead. We always use the TEST variant in the following.

x86: AVX-512. On x86, we can address the 128-bit/256-bit registers either using SSE/AVX2 or using AVX-512. We find that AVX-512 performs slightly better, e.g., 1.02x higher read throughput compared to SSE on LF 70 %. Hence, we use AVX-512 on Intel. The AMD x86 system does not support AVX-512.

ARM: NEON and SVE. We can vary the NEON and SVE implementations. For NEON, we describe the SSE2NEON and the UMINV variant (Section 3.3). We find that UMINV provides a 1.06x higher average read throughput. For VLP, it provides better performance than porting the x86 approach, which shows that tuning the implementation to the CPU can increase performance. For SVE, we can perform the vector comparison against a scalar, or we broadcast the scalar into a vector and compare two vectors. We find no performance difference.

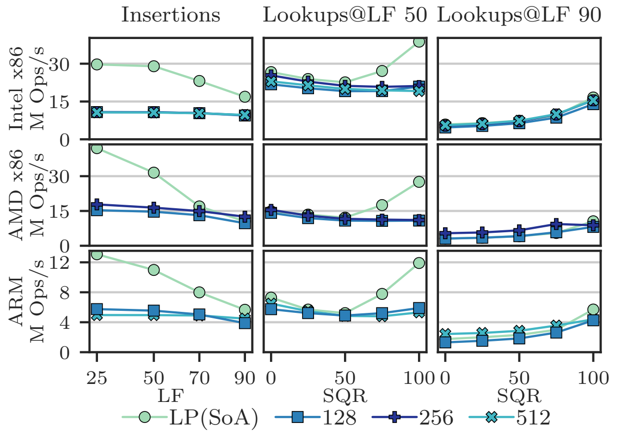

Figure 5: Performance of VLP, depending on the vector register size in bit.

4.3.2 Impact of Register Size. As a main design decision, we evaluate the impact of the vector register size to assist with the choice when implementing a vectorized scheme. For Intel, AMD, and ARM, the results are shown in Figure 5. For Power, VLP is always outperformed by LP. Using 128-bit registers, VLP does not outperform SoA LP due to the overhead of the SIMD probing algorithm and additional memory access for the validity information for both inserts and lookups. VLP performs explicit loads, check for matches, and extracts the match indices. We furthermore require additional logic to handle memory boundaries and alignment. The low degree of parallelism does not compensate for the overhead.

x86. For LF 25 %, the baseline consistently provides higher read throughput, even when using 512-bit vector registers. This is because the probing sequences are too short to benefit from vectorization. It is faster to check the first few keys in a scalar fashion than to setup the SIMD routine in this case. At LF 50 % and above, in low SQR settings, we measure higher throughput with larger registers on both Intel and on AMD. For example, at load factor 50 % and SQR 50 % on Intel, 256-bit VLP is 1.19x faster than LP. However, for SQR 100 %, VLP achieves only 0.76x of LP's throughput. This is due to the short average probing length of 1.5 at SQR 100 %.

Doubling the vector register size does not double the performance, but the AMD CPU benefits more from larger vector registers than the Intel CPU. For example, at the high load factor 90 %, on Intel, 128/256/512-bit VLP has 1.02x/1.16x/1.18x the performance of SoA LP. On AMD, 128-bit VLP has 1.18x and 256-bit VLP has 1.84x the performance of LP. For inserts, we find that on Intel, all vector sizes perform similar, while on AMD, 256-bit VLP always outperforms 128-bit VLP, and 128-bit VLP outperforms LP. In general, we advise to always use the largest supported register size.

ARM. On ARM SVE, which has 512-bit registers, we find similar behavior to Intel's 512-bit registers. For load factor 70 %, SVE outperforms linear probing for SQR 50 % and below. For load factor 90 %, linear probing is only better for SQR 100 %. At 0 % SQR and SQR 25 %, we find performance gains of 1.74x and 1.59x respectively, compared to the LP baseline. Averaged across all SQRs, SVE is 1.32x and 1.5x faster than NEON for LFs 70 % and 90 %. For inserts, we find that NEON performs better until LF 70 %, and SVE has higher performance for inserts to LF 90 %.

Similar to Intel, 512-bit SVE does not quadruple the performance of 128-bit NEON. First, the SVE algorithm is slightly more complicated. For example, we need to recreate the index vector required to extract the match each time the find function is called. Second, the A64FX is made for HPC operations, i.e., the SVE instructions that we use, such as `CMPEQ`, `INDEX`, and `BRKA`, have a high latency and low throughput [26].

Power. As Power only supports 128-bit vector registers, VLP does not provide good performance for 64-bit keys.

Discussion. VLP with 128-bit registers on 64-bit keys does not improve insert and lookup performance compared to regular linear probing, due to the small degree of parallelism and the additional algorithmic overhead. To improve VLP, (1) we need to reduce the two random memory loads per probe, (2) we need to increase our degree of parallelism to meaningfully use 128-bit registers and compensate for the algorithmic overhead, and (3) we should become independent of the key data type to not depend on, e.g., whether we use 32-bit or 64-bit keys.

4.4 Vectorized Fingerprinting

As VLP does not achieve higher performance than its scalar counterpart, we show the impact of adding vectorized fingerprints (VFP). The VFP algorithm stores 8- or 16-bit fingerprints and compares the fingerprints using vector instructions. This enables a higher degree of parallelism independent of the key data type and allows to define a fingerprint as invalid, such that we avoid unnecessary random memory access.

4.4.1 Implementation Choices. Similar to VLP, we find that the TEST variant of VFP always performs equivalent or better than the NOTEST variant. Following Bronson and Shi [21], we use branching hints, assuming that fingerprint clashes are unlikely.

x86. AVX-512 provides simpler masks than SSE/AVX2 when resolving hash collision. For 8-bit fingerprints, using AVX-512 to address 256-bit registers has 1.05x higher throughput on average. For smaller load factors, this number increases, e.g., for LF 25 %, there is a 1.09x speedup. We use AVX-512 on Intel for VFP.

ARM and Power. For ARM NEON and Power, we propose a vectorized iterator and a scalar iterator, which operates on a simulated movemask that x86 offers natively. The simulated movemask is 1.17x and 1.11x faster on ARM and Power. This is different to VLP, where UMINV was the best approach. This is because the overhead of iterating is very small, as it just consists of bit manipulations.

4.4.2 Fingerprints. One approach to determine the fingerprint is to take the LSBs of the hash for both the slot and the fingerprint, meaning that fingerprints are correlated with the index (*LSBLSB*). Another approach is to use the MSBs of the hash for the fingerprints (*LSBMSB*). For hash functions such as MS that do not require a finalizer, there are no additional bits we can use. For MS-style hashing, before the final shift, we take the LSBs that get shifted out. This gives us additional entropy for the fingerprint, while maintaining a true MS hash function for the index itself. Using these bits increases the throughput significantly. If we use 128-bit registers, LF 70 %, and 8-bit fingerprints, we obtain an average speedup of 1.34x across all systems. On all load factors, the speedup peaks at SQR 0 %. At LF 70 and SQR 0 %, we measure a peak speedup of 1.72x. For 16-bit fingerprints, we obtain similar speedups.

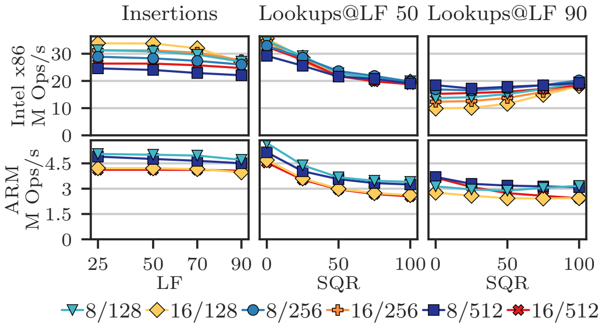

Figure 6: Performance of VFP depending on fingerprint / vector register size.

To understand why LSBMSB fingerprints are effective, consider the probing length. For LF 70 % and with LSBLSB fingerprints, we compare 17.23 fingerprints per find on average. During these probes, we encounter 0.53 collisions per find operation. If we switch to LSBMSB, we encounter only 0.03 collisions per find operation on average. This means we have to do less work, i.e., constructing an iterator and iterating over all matches. The speedup grows with lower SQRs, because lower SQRs imply longer probe sequences. In longer probe sequences, there is more potential for fingerprint collision. We experimentally confirm this, as for SQR 0 %, the difference in collisions between LSBLSB and LSBMSB is largest.

This also holds for 16-bit fingerprints. Even though the fingerprint itself is larger, it is correlated with the index. Hence 16-bit fingerprints cannot provide any advantage over 8-bit fingerprints without using LSBMSB fingerprints. When using LSBMSB fingerprints instead, 16-bit fingerprints can utilize their larger number space and thereby reduce the collisions per find to 0.00008, compared to 0.03 for 8-bit fingerprints.

4.4.3 Vector Register/Fingerprint Size. Having optimized some aspects of the VFP algorithm, we now discuss the interdependencies of vector register size and fingerprint size. In Figure 6, we show the throughput of 8-bit and 16-bit fingerprints depending on the register size on Intel and ARM. Other systems follow similar patterns.

Vector Register Size. Keeping the fingerprint size constant, on Intel, we find that for the low load factor 25 %, the smaller vector registers have higher read throughput than larger vector registers. This is because the larger vector instructions are more expensive, but not required, as the probe sequences are very short. For LFs 70 % and 90 %, larger vector registers perform better than the smaller. For LF 90 %, on Intel, 512-bit registers outperform 256-bit registers for 8-bit/16-bit fingerprints by 1.02x/1.14x and 128-bit registers by 1.16x/1.33x. On AMD, 256-bit registers outperform the 128-bit registers by 1.14x/1.16x because they shorten the number of vector probes. For inserts, we find that 128-bit registers performs best for all LFs except 90 %, where 256-bit registers performs better. 512-bit registers always have a lower insert throughput because the lookup preceding the insert performs worse for short sequences.

For ARM, we find similar behavior to x86. NEON outperforms SVE for low load factors and SVE is better for high load factors. For inserts, NEON always performs better than SVE.

Fingerprint Size. Keeping the vector size fixed, for load factors 70 % and 90 %, 8-bit fingerprints outperform 16-bit fingerprints on all systems. At load factor 90 %, averaged across all SQRs, they provide a 1.11x, 1.21x, 1.3x, and 1.19x higher throughput than 16-bit fingerprints, on Intel (512-bit), AMD (256-bit), ARM (512-bit), and Power, respectively. Increasing the fingerprint size to 16-bit exponentially decreases the chance of collision, but also halves our degree of parallelism. Let us consider load factor 90 % at SQR 0 %. Using 128-bit registers and 8-bit fingerprints, we perform 3.96 vectorized comparisons per find operation and resolve 0.24 conflicts on average. Using 16-bit fingerprints, we perform 7.09 vector probes per find, and resolve 0.0008 conflicts on average per find. We see that *(a)* while 16-bit fingerprints reduce the number of collisions per find to a very low number, even with 8-bit fingerprints, we have a conflict only in every fourth find operation. With that in mind, *(b)* as the number of collisions with 8-bit fingerprints is already low, and as collision resolution is not very expensive, the doubled degree of parallelism of 8-bit fingerprints and therefore the reduction of 7.09 probes per find to 3.96 probes per find explain why 16-bit fingerprints perform worse than 8-bit fingerprints.

For the lower load factors 25 % and 50 %, on ARM, 8-bit fingerprints always perform noticeably better than 16-bit fingerprints. On x86, sometimes, 16-bit fingerprints outperform 8-bit fingerprints in low load factor/low SQR scenarios, but only marginally.

For inserts, we find that on both x86 systems and Power, keeping the vector register size fixed, 16-bit fingerprints perform slightly better. On ARM, 8-bit fingerprints perform slightly better. However, these differences are minor. Overall, we suggest using 8-bit fingerprints for VFP, as it outperforms 16-bit fingerprints consistently for high load factors on all systems, and is only rarely outperformed by 16-bit fingerprints in read throughput.

Scalar Schemes. On x86 and Power, VFP consistently performs better than all scalar hashing schemes for LFs 70 % and 90 %, except for LF 70 % and SQR 100 %. While VLP was not able to outperform LP in most cases, VFP outperforms scalar schemes even with 128-bit registers (see Figure 8 for an overview). On ARM, we find that VFP improves upon LP at LF 90 %, but performs on par with robin hood hashing, even when using SVE. At LF 90 %, VFP has a 2.27x, 2.35x, 1.35x, and 2.18x higher throughput than the LP baseline, on Intel, AMD, ARM, and Power, respectively. Power strongly benefits from SIMD instructions. It only has a 128-bit vector register size but reaches an average read throughput of 17.72 MOps/s, while the Intel CPU, which has 512-bit registers, reaches 18.23 MOps/s. For smaller load factors (50 % and below), scalar hashing schemes are still superior. While VFP's insert performance stays consistent for all LFs, the insert performance of the scalar tables strongly depends on the load factor. For LF 90 %, VFP's insert performance is higher than RH and LP, but for all other load factors, VFP's insert performance is lower on most systems.

Discussion. VFP outperforms scalar tables mostly in high load factor settings. We emphasize that different systems benefit from SIMD instructions differently. Although the vector registers of the Power machine are four times smaller than those of the Intel machine, Power achieves a similar relative speedup and just slightly lower absolute throughput. This shows that we cannot transfer x86-specific insights to any CPU and just expect them to hold there. If one were to evaluate the benefits of vectorized hashing with 128-bit registers only on x86, the conclusion would be not to use vectorization, while an evaluation on Power would lead to the conclusion to

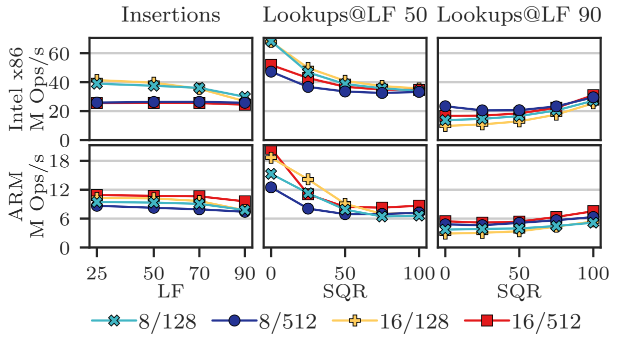

Figure 7: Performance of BBC depending on fingerprint / vector register size.

use vectorization. We evaluate the bucket-based comparison (BBC) hashing scheme next, which avoids unaligned loads, increases data locality, and makes the implementation more compact.

4.5 Bucket-Based Comparison

For BBC, similar to VFP, we verify that the TEST variant performs better than NOTEST. We always use LSBMSB fingerprinting, likely hints, AVX-512 on Intel, and scalar iterators on Power and NEON.

4.5.1 Bucket Size/Fingerprint Size. We investigate which combination of vector register size (bucket size), and fingerprint size performs best. For Intel and ARM, the results are shown in Figure 7. Other systems follow similar patterns.

Vector Register Size. If we keep the fingerprint size constant, we achieve similar results as for VFP, i.e., for lower load factors, smaller registers are better, while larger registers perform better with higher LFs. For LF 25 %, Intel x86, and 8-bit fingerprints, 128-bit registers have higher read throughput than 256-bit (1.02x) and 512-bit (1.19x). For load factor 70 %, 256-bit registers perform better than 128-bit (1.04x) and 128-bit perform better than 512-bit (1.11x). For load factor 90 %, the 512-bit vector registers perform best. On ARM, SVE performs better than NEON for LF 90 %, Regarding inserts, on Intel, the 512-bit registers are only better for LF 90 %, while on AMD, the 256-bit registers consistently outperform 128-bit.

Fingerprint Size. For a constant vector register size, the result is not as clear as for VFP. For LF 90 % on x86, 8-bit fingerprints are still preferable to 16-bit fingerprints. On average, on Intel x86 and 128-bit registers, 8-bit fingerprints are 1.25x faster than 16-bit fingerprints. On AMD x86, they are 1.1x faster. This is because at this load factor, 8-bit fingerprints reduce the average number of probed buckets from 3.71 to 2.29, as the buckets contain more fingerprints. While the number of collisions per find is nearly zero for 16-bit, we find that for 8-bit, it is 0.12. However, fewer collisions on 16-bit fingerprints are counterbalanced by the fact that the number of followed overflows per find increases when switching from 8-bit to 16-bit from 1.29 to 2.71. Overall, collision resolution is cheaper than following overflows, because when accessing the next bucket, we have to construct another iterator and perform another load. Hence, the larger buckets we get from 8-bit fingerprints are preferable over fewer collisions we get from 16-bit fingerprints.

For smaller LFs, we find that 16-bit fingerprints perform better than 8-bit fingerprints. At these LFs, the buckets overflow less frequently, even when using 16-bit fingerprints. This enables us to benefit from fewer collisions. For example, at LF 50 %, with 128-bit registers and 16-bit fingerprints, only 2.32 % of all buckets overflow. The larger 16-bit fingerprints can thus improve performance by reducing the number of collisions. For inserts, like VFP, 16-bit fingerprints perform better on all systems.

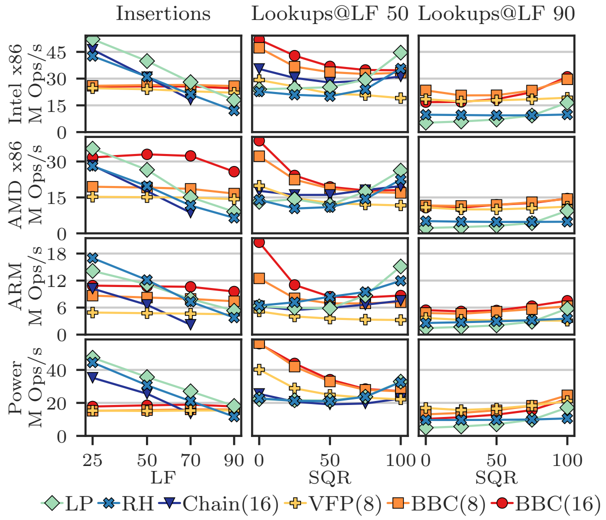

Figure 8: Performance of BBC (8-bit and 16-bit fingerprints) and VFP (8-bit FPs), linear probing (AoS), robin hood (recalculating), and chained hashing (16-bit FP budget). We show the largest available register size.

Overall, there is no one-fits-all fingerprint. While 16-bit fingerprints perform better for smaller LFs and for inserts, they also require double the amount of memory for the fingerprints. In the following, we either show both 8-bit and 16-bit BBC, or only 8-bit BBC, as it is the memory-friendlier option.

4.5.2 Comparison to VFP and Scalar Tables. We compare BBC to VFP, linear probing, chained hashing, and robin hood hashing on all systems in Figure 8. For chained hashing, we include 16-bit fingerprints in the budget calculation. BBC improves upon VFP in almost all scenarios. For LF 70 %, BBC with 8-bit fingerprints provides a 1.5x, 1.46x, 2.06x, and 1.26x higher read throughput than VFP, for Intel x86, AMD x86, the A64FX, and Power, respectively. For the very high LF 90 %, the performance gap between BBC and VFP becomes smaller. BBC has a 1.29x, 1.19x, and 1.63x higher throughput than VFP, on Intel, AMD, and the A64FX. On Power, it achieves only 0.95x of VFP throughput. Power does not have a movemask instruction, so the high number of mask-iterators that are created impact performance. The BBC insert performance is consistently higher than the VFP insert performance for all LFs.

Most notably, BBC significantly improves the read performance of VFP for the smaller LFs 50 % and 25 %. At LF 50 %, BBC with 8-bit fingerprints has a 1.59x, 1.5x, 2.13x, and 1.32x higher throughput than VFP on Intel x86, AMD x86, the A64FX, and Power. For lower SQRs, BBC has a better performance than scalar hashing schemes, which VFP does not. Across all systems, for LF 50 %, 8-bit BBC is 1.39x faster than AoS LP and 1.37x faster than chaining. For LF 70 %, these numbers increase to 2.28x (AoS LP) and 3.88x (chaining),

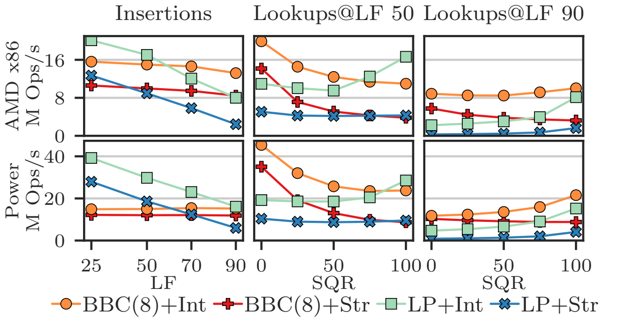

Figure 9: 8-bit BBC and LP (AoS) with integer and string keys.

and for LF 90 %, the improvement is even higher. Similar to VFP, Intel and Power have comparable speedups, despite Intel's four times larger registers. Also similar to VFP, we see that BBC's insert performance stays almost consistent across all LFs. The insert performance of scalar tables is skewed towards low LFs due to shorter sequences, which is why they exceed BBC for LF < 50 % but perform worse with higher load factors.

BBC performs better than VFP because *(a)* BBC ensures that the first load is aligned instead of unaligned, which is especially noticeable in low LF scenarios where we often only perform a single load, *(b)* we only have to perform a single comparison and not check for any invalid fingerprints in BBC, and *(c)* because the overflow logic allows us to terminate earlier than in VFP in some cases, if the bucket is full, but did not overflow. Both in BBC and VFP, compared to scalar hashing schemes, we can abort earlier due to the check if there has been any match in the bucket with the TEST variant. For low SQR scenarios, BBC performs only one comparison per bucket, whereas VFP needs to check for invalid fingerprints, which is unsuccessful for the first vectors. To summarize, BBC increases VFP performance for both high and low load factors and therefore provides a good general-purpose hashing scheme.

After establishing the baseline performance of the vectorized and scalar hashing schemes, we focus on different setups to show the performance variance and robustness of the individual designs. To this end, we first compare integer with string keys, and then investigate large payloads, skewed workloads, and multithreading.

4.6 String Keys

Storing strings and non-fixed sized values is an important feature for hash tables. When using strings instead of integers, we face two challenges. First, while we can store integer keys inline, strings are represented as pointers to heap memory, due to their variable length. This implies that looking up a string key requires a random memory access. Second, there is no assembly string comparison instruction. String comparison is usually implemented in standard-library functions, such as `std::strcmp`. For string keys, we use the xxHash hash function instead of multiply-hashing because it supports hashing of variable-length data.

We compare linear probing (AoS) and BBC for both integer and string keys. The results for AMD and Power are shown in Figure 9. Other systems have similar patterns. We show the results for integer keys using xxHash as well, to isolate the effects of string keys. For LP, in each probing step, we perform a string comparison, while for BBC, we only do a string comparison if we have a fingerprint match.

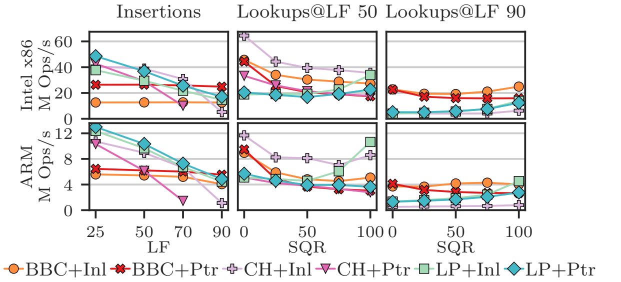

Figure 10: Performance of AoS LP, 8-bit BBC, and Chained Hashing (8-bit fingerprint budget) for 32 B values, stored inline (+Inl) or as pointers (+Ptr).

This means that for SQR 0 %, BBC performs a string comparison only if there is a random fingerprint collision. Hence, the lower the SQR, the more the BBC string performance approaches the BBC integer performance. For load factor 70 %, AMD x86, and BBC, integer keys have a 1.5x, 2.5x, and 3.02x higher throughput for SQRs 0 %, 50 %, and 100 %. For LP we find that the gap between integers and strings does not get smaller for lower SQRs. It even increases for higher SQRs for both LP and BBC, because compared to integer keys, we perform another random memory access to lookup the string, which becomes the bottleneck in high SQRs scenarios. This is also why BBC string performance approaches LP string performance for SQR 100 %. For inserts, string performance is lower than integer performance due to the string comparison overhead, but the relative trend does not change.

For string keys, with the example of AMD x86 and load factor 70 %, BBC is 6.04x faster than LP for SQR 0 %, but only 1.07x faster for SQR 100 %. Vectorized hashing schemes provide a much higher throughput than scalar hashing schemes especially for lower SQRs. For lower SQRs, BBC achieves almost integer key performance. For very high SQRs, random memory access to the strings is the bottleneck, i.e., BBC and LP performance converge.

4.7 Large Payloads

In the previous sections, we use 64-bit integer values. In join workloads, tuples are often larger than 8 B [10]. We benchmark the hashing schemes using 32 B payloads. For larger values, it is important to differentiate whether the hash table stores the values inline (flat layout) or pointers to values (node layout). We compare both approaches on Intel and ARM in Figure 10. Other systems follow similar patterns. When considering larger payloads (e.g., 128 B), the observed effects amplify for the inline setting, and node layout size stays consistent, as the pointers are always of the same size.

4.7.1 Flat Layout. The open addressing schemes allocate memory for all slots, even though not all slots will be filled. The chained hash table is able to allocate a buffer with the exact required size, and use the rest of the memory for the dictionary.

For load factors 25 % and 50 %, chained hashing has 1.48x and 1.38x higher performance than BBC, averaged over all systems. However, with LF 70 %, chained hashing has similar performance to BBC, which drops to around 20 % of BBC's performance for LF 90 %. This is because the lower the load factor, the higher the over-allocation of open addressing. For higher load factors, chained

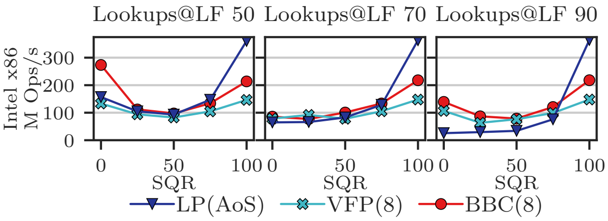

Figure 11: Lookup performance with Zipfian distribution.

hashing has to allocate more memory to the buffer, and hence cannot utilize the larger directory. For LF 25 %, the chained table increases the directory to hold 2^{29} entries, which explains the high performance at lower load factors. This is also noticeable for inserts, where for all LFs except 90 %, chained hashing performs best and then significantly drops below the others for 90 %.

When using 32 B payloads, we find a larger absolute performance discrepancy between small and large values for lower load factors than for high load factors. For example, on Intel, at LF 25 % using linear probing, small values have an 1.23x higher throughput on average than larger values, equal to a difference of 8.6 MOps/s. At LF 90 %, they have a 1.24x higher average throughput, which is, however, equivalent to only 1.6 MOps/s, as the bottleneck shifts from memory to compute for probing. For BBC, similarly to string keys, we observe that the lower the SQR, the closer the large value performance gets to small value performance. This is because we do not actually have to touch and load the large values for non-matches in BBC, unlike AoS LP that iterates through the values, no matter whether there is a match.

4.7.2 Node Layout. In a node layout, we avoid the over-allocation of memory in open addressing schemes. The values may already reside in a buffer, such that we can store pointers into this buffer.

The open addressing schemes outperform the chained hash table for LFs 50 % and above. For LF 50 %, BBC is 1.23x faster than chained, for LF 70 %, it achieves 5.81x. The chained table cannot fit all items into the memory budget at LF 90 %.

We compare the node layout to the flat layout on BBC. For lookups, BBC clearly prefers inline values on Intel. For the A64FX, inline is not better as clearly as on Intel. We find that pointer insert performance is always higher than inline insert performance. Pointers have a 1.98x and 1.36x higher insert throughput on Intel x86 and the A64FX, for inserts to LF 90 %. Furthermore, storing values inline requires more memory and hence the node layout should be considered for large values.

4.8 Skewed Workloads

In this experiment, the successful and unsuccessful keys are sampled from a Zipfian distribution with $\alpha = 1.5$ instead of uniformly at random. This means that some keys are very frequently requested, while others are only rarely. The results for Intel are shown in Figure 11. Other systems follow similar patterns.

Performance under the Zipfian distribution is higher than under uniform distribution due to caching. On average, for BBC and LF 90 %, Zipfian performance is 5.84x higher. For SQR 100 %, LP is the best scheme, but for lower SQRs, BBC is the best scheme on all systems. Additionally, at SQR 0 %, BBC performance increases more than with a uniform distribution. To ensure correct SQRs, we sample from two distributions. This results in a slight bias towards SQRs 0 % and 100 %, as they sample from only one distribution. Regardless of this, we see that caching benefits all load factors and SQRs. Overall, for very high SQRs, under Zipfian distribution, LP should be used; otherwise, BBC works well on all systems.

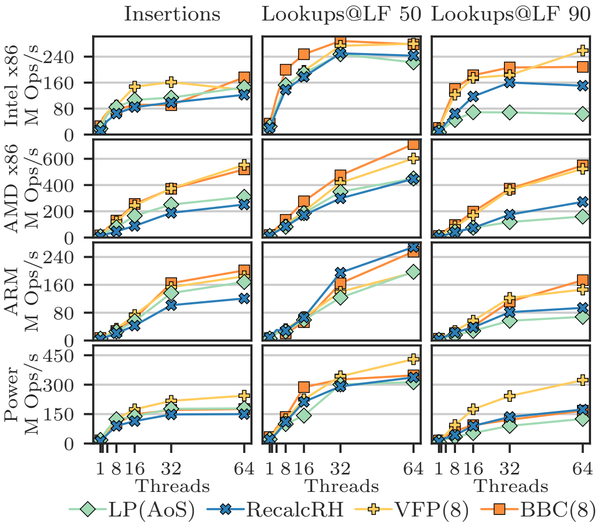

Figure 12: Insert (LF 90 %) and lookup (SQR 50 %) performance depending on the thread count.

4.9 Multithreading

In our final experiment, we use $n > 1$ threads. Each thread operates on a hash map of size $2^{27}/n$, fills it to the load factor, and queries it with the respective SQR. We show the overall throughput for SQR 50 % in Figure 12 and mention other SQRs below.

For lookups, we find that on x86 and ARM, BBC almost always performs best. On Intel, in the single-thread setting at LF 90 %, BBC is 2.93x and 1.16x faster than LP and VFP. Averaged over all thread counts, we obtain similar speedups of 3.02x and 1.08x. The relative benefit of BBC is independent of the thread count. On Power, the advantage of VFP over BBC at LF 90 % is amplified. With one thread, VFP is 1.07x faster than BBC, but averaged over all threads, VFP is 1.49x faster. We attribute this again to Power's high overhead of movemask creation in combination with the high increase in bucket overflows at LF 90 %. For all thread counts, it holds that for very low SQRs, the benefit of vectorization is higher, and for very high SQRs, LP performance approaches BBC performance. Last, for inserts to LF 90 %, BBC resp. VFP outperform scalar schemes, but for inserts to smaller LFs, scalar schemes perform better. Different SQRs shift the results in favor of vectorization (lower SQRs) or scalar (higher SQRs) due to the shorter probing sequences. For SQR 100 %, on Intel and averaged over all thread counts, the speedup of BBC over LP is 1.65x. For SQR 0 %, we obtain a speedup of 4.74x.

5 DISCUSSION

In this section, we present key takeaways from our benchmarks of the scalar and vectorized hashing schemes.

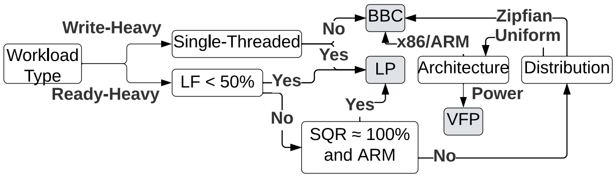

Figure 13: Decision guide for hashing scheme selection.

(1) **Vectorized Hashing > Scalar Hashing:** For high LF scenarios, platforms benefit from vectorized hashing with BBC, compared to scalar hashing schemes. Naive vectorization such as VLP does not perform well, which shows that vectorization requires careful consideration and re-design of existing approaches. In Figure 13, we give a decision guide to decide which hashing scheme to use based on the workload. We recommend BBC for most high LF read scenarios, with exceptions for uniform distribution on Power, and 100 % SQR on ARM. Throughput all experiments, we observe that BBC is a very robust choice, while LP varies more and should be chosen only if the workload characteristics are known and suitable for LP.

(2) **Differences between Platforms:** The relative performance speedups of vectorization differ between platforms. For example, for both VFP and BBC, Power has the same relative performance improvement with 128-bit registers as Intel has x86 with 512-bit registers, despite a four times lower degree of parallelism. It is challenging to draw general conclusions about the efficiency of vectorization by evaluating a single platform.

(3) **Memory Budget:** The performance of hash tables must always be considered together with memory consumption, as most hashing schemes converge towards perfect hashing given enough memory. Our evaluation shows that vectorized schemes outperform scalar ones for high load factors, striking a balance between performance and memory consumption. But they often perform worse with low load factors due to more complex lookup logic. Thus, vectorized schemes should only be chosen given limited memory resources.

(4) **Engineering Overhead:** The hashing schemes need to be optimized for each platform at hand, and the algorithm design varies between platforms. For example on SVE, creating the index vector per probe instead of per find operation massively degraded performance. This leads to a big engineering overhead during initial development, where an in-depth understanding of each platform is necessary. When developing for a single system, the abstractions can be optimized towards that system. Finding the best level of abstraction over multiple architectures is difficult, as seen on the example on movemasks, which are natively supported by x86, but very expensive on Power. Our SIMD abstraction layer alone consists of over 2 000 LOC with many metaprogramming and preprocessor statements to check for available hardware.

(5) **Vector Register Size:** For VLP, larger registers lead to better performance. For VFP and BBC, the largest register does not always perform best, e.g., with low load factors, SSE performs better than AVX-512. We still advise to use the largest available register for better performance with high load factors.

(6) **Fingerprints:** For VFP, using 8-bit fingerprints always performs best. For BBC, 16-bit fingerprints perform a little better for low to medium load factors, but require more memory. It is important to carefully think about how to obtain the fingerprint, depending on the hash function, i.e., do not overlap the bits that define the index with the bits that define the fingerprint.

(7) **Implementation Details:** When extracting matches from a movemask, it is beneficial to check whether there has been any match (TEST). For iterating over multiple matches on ARM and Power, which do not natively support movemasks, simulating a movemask instead of working with a native vectorized iterator performs better. Pre-initialization of re-used vector registers (e.g., index vectors) is important, especially on the A64FX, as the SVE instructions to generate these vectors are very expensive.

6 RELATED WORK

We build upon previous research in the area of hash tables, hash joins, and platform-aware computing.

Hash Tables. Richter et al. [66] conduct an analysis of scalar hashing schemes and derive a decision guide that focuses on the workload at hand. They implement a variant of VLP on AVX2. Polychroniou et al. [65] differentiate horizontal and vertical vectorization. Vertical vectorization looks up multiple keys in parallel, which requires scatter/gather operations and bulk inserts/lookups. Horizontal vectorization serves as a drop-in replacement for scalar hash tables. Pietrzyk et al. [63] implement a conflict detection-aware version of vertical VLP. Behrens et al. [13] use OpenCL to implement vertical VFP. Meta's F14 [20, 21] and Google's Abseil containers [28] are industry implementations of BBC using SSE/NEON.

Hash Joins. Hash joins have been researched intensively as they outperform sort-merge joins [7, 37]. Most works discuss how to parallelize the join [8–11, 40, 55, 64, 69]. As all strategies benefit from a faster hash table, our work is orthogonal to this research.

Platform-Aware Computing. Several works have shown the benefits of adapting applications to the underlying platform, e.g., by using SIMD [41, 44, 45, 57]. IBM Power systems are used in previous work, as they integrate well with various accelerators [49–51]. Bari et al. [12] find that A64FX single-thread performance is low, in line with our findings. The A64FX performs well on finely-tuned, multi-threaded HPC workloads [1, 15, 22, 47].

7 CONCLUSION

In this paper, we compare three different approaches to vectorized hashing and implement them on three architectures with six different vector ISAs. We perform an exhaustive benchmark set, varying parameters such as load factor and successful query rate, and show the impact of various algorithmic and system-specific tuning knobs. We show that the BBC hashing scheme provides a good general-purpose hashing scheme that outperforms scalar linear probing often by over 2x, reaching over 6x for string keys. We discuss our findings and make our implementation available to improve the understanding of adapting key database components to increasingly heterogeneous environments.

ACKNOWLEDGMENTS

This work was partially funded by the Swiss National Science Foundation (200021_204620), the German Research Foundation (414984028), and the European Union's Horizon 2020 (957407).

REFERENCES

[1] Christie Alappat, Nils Meyer, Jan Laukemann, Thomas Gruber, Georg Hager, Gerhard Wellein, and Tilo Wettig. 2021. Execution-Cache-Memory modeling and performance tuning of sparse matrix-vector multiplication and Lattice quantum chromodynamics on A64FX. *Concurrency and Computation: Practice and Experience* (2021). https://doi.org/10.1002/cpe.6512

[2] Amazon Press Releases. 2022. AWS Announces General Availability of Amazon EC2 C7g Instances Powered by AWS-designed Graviton3 Processors. AWSAnnouncesGeneralAvailabilityofAmazonEC2C7gInstancesPoweredbyAWS-designedGraviton3Processors

[3] Apple. 2020. Apple unleashes M1. https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/

[4] ARM Limited. 2020. *Arm Architecture Reference Manual Supplement - The Scalable Vector Extension (SVE), for Armv8-A*. https://developer.arm.com/documentation/ddi0584/latest/

[5] ARM Limited. 2022. *Cortex-A57 Software Optimization Guide*. https://developer.arm.com/documentation/uan0015/b/

[6] ARM Limited. 2022. Introducing SVE2. https://developer.arm.com/documentation/102340/0001/Introducing-SVE2

[7] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proceedings of the VLDB Endowment* 7, 1 (2013). https://doi.org/10.14778/2732219.2732227

[8] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the International Conference on Data Engineering (ICDE)*. https://doi.org/10.1109/icde.2013.6544839

[9] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2015. Main-Memory Hash Joins on Modern Processor Architectures. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2015). https://doi.org/10.1109/tkde.2014.2313874

[10] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. https://doi.org/10.1145/3448016.3452831

[11] Ronald Barber, Guy M. Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, Gopi K. Attaluri, Naresh Chainani, Sam Lightstone, and David Sharpe. 2014. Memory-Efficient Hash Joins. *Proceedings of the VLDB Endowment* 8, 4 (2014). https://doi.org/10.14778/2735496.2735499

[12] Md Abdullah Shahneous Bari, Barbara Chapman, Anthony Curtis, Robert J. Harrison, Eva Siegmann, Nikolay A. Simakov, and Matthew D. Jones. 2021. A64FX performance: experience on Ookami. In *Proceedings of the International Conference on Cluster Computing (CLUSTER)*. https://doi.org/10.1109/Cluster48925.2021.00106

[13] Tobias Behrens, Viktor Rosenfeld, Jonas Traub, Sebastian Breß, and Volker Markl. 2018. Efficient SIMD Vectorization for Hashing in OpenCL. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. https://doi.org/10.5441/002/EDBT.2018.54

[14] Lawrence Benson and Tilmann Rabl. 2022. Darwin: Scale-In Stream Processing. In *Proceedings of Annual Conference on Innovative Data Systems Research (CIDR)*.

[15] Robert Bird, Nigel Tan, Scott V. Luedtke, Stephen Lien Harrell, Michela Taufer, and Brian Albright. 2022. VPIC 2.0: Next Generation Particle-in-Cell Simulations. *Transactions on Parallel and Distributed Systems* 33, 4 (2022). https://doi.org/10.1109/TPDS.2021.3084795

[16] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of International Conference on Management of Data (SIGMOD)*. https://doi.org/10.1145/1989323.1989328

[17] Hugh Blemings. 2019. The Next Step in the OpenPOWER Foundation Journey. https://openpower.foundation/blog/the-next-step-in-the-openpower-foundation-journey/

[18] Hugh Blemings. 2020. Final Draft of the Power ISA EULA Released. https://openpower.foundation/blog/final-draft-of-the-power-isa-eula-released/

[19] Sebastian Breß, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2019. An Overview of Hawk: A Hardware-Tailored Code Generator for the Heterogeneous Many Core Age. https://doi.org/10.18420/BTW2019-WS-07

[20] Nathan Bronson and Xiao Shi. 2019. F14 Readme. https://github.com/facebook/folly/blob/b634e420d75e72cc96762e1ec4fb90f44d59c464/folly/container/F14.md

[21] Nathan Bronson and Xiao Shi. 2019. Open-sourcing F14 for faster, more memory-efficient hash tables. https://engineering.fb.com/2019/04/25/developer-tools/f14/

[22] Andrew Burford, Alan Calder, David Carlson, Barbara Chapman, Firat Coskun, Tony Curtis, Catherine Feldman, Robert Harrison, Yan Kang, Benjamin Michalowicz, Eric Raut, Eva Siegmann, Daniel Wood, Robert DeLeon, Mathew Jones, Nikolay Simakov, Joseph White, and Dossay Oryspayev. 2021. Ookami: Deployment and Initial Experiences. In *Proceedings of Practice and Experience in Advanced Research Computing (PEARC)*. https://doi.org/10.1145/3437359.3465578

[23] Pedro Celis, Per-Ake Larson, and J. Ian Munro. 1985. Robin Hood Hashing. In *Proceedings of The Annual Symposium on Foundations of Computer Science (SFCS)*. https://doi.org/10.1109/sfcs.1985.48

[24] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. 1997. A Reliable Randomized Algorithm for the Closest-Pair Problem. *Journal of Algorithms* 25, 1 (1997). https://doi.org/10.1006/jagm.1997.0873

[25] Andrei Frumusanu. 2021. Arm Announces Armv9 Architecture: SVE2, Security, and the Next Decade. https://www.anandtech.com/show/16584/arm-announces-armv9-architecture

[26] Fujitsu. 2022. A64FX Microarchitecture Manual v1.7. https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_Microarchitecture_Manual_en_1.7.pdf

[27] William Gayde. 2020. How Arm Came to Dominate the Mobile Market. https://www.techspot.com/article/1989-arm-inside/

[28] Google Inc. 2022. Abseil Containers. https://abseil.io/docs/cpp/guides/container#abseil-containers

[29] Ferdinand Gruber, Maximilian Bandle, Alexis Engelke, Thomas Neumann, and Jana Giceva. 2023. Bringing Compiling Databases to RISC Architectures. *Proceedings of the VLDB Endowment* 16, 6 (2023). https://doi.org/10.14778/3583140.3583142

[30] Philipp M. Grulich, Sebastian Breß, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. https://doi.org/10.1145/3318464.3389739

[31] Bala Gurumurthy, David Broneske, Marcus Pinnecke, Gabriel Campero, and Gunter Saake. 2018. SIMD Vectorized Hashing for Grouped Aggregation. In *Proceedings of Advances in Databases and Information Systems (ADBIS)*. https://doi.org/10.1007/978-3-319-98398-1_8

[32] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch Hashing. In *Proceedings of the International Symposium on Distributed Computing (DISC)*. https://doi.org/10.1007/978-3-540-87779-0_24

[33] IBM. 2019. *POWER9 Processor User's Manual*. https://www.ibm.com/systems/power/openpower/posting.xhtml?postingId=FC2A3168C5821FA78525803D00719802

[34] IBM. 2020. IBM Power System IC922: Technical Overview and Introduction. https://www.redbooks.ibm.com/redpapers/pdfs/redp5584.pdf

[35] Intel Corporation. 2022. *Intel 64 and IA-32 Architectures Software Developer's Manual*. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[36] Dashveenjit Kaur. 2021. Intel and AMD reign supreme in record 2021 server market. https://techhq.com/2021/08/intel-and-amd-reign-supreme-in-record-2021-server-market/

[37] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment* 2, 2 (2009). https://doi.org/10.14778/1687553.1687564

[38] Donald Knuth. 1963. Notes On Open Addressing.

[39] Donald Knuth. 1973. *The Art Of Computer Programming, Volume 3: Sorting And Searching*. Addison-Wesley.

[40] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. 2013. Massively Parallel NUMA-Aware Hash Joins. In *Proceedings of the International Workshop on In-Memory Data Management and Analytics (IMDM)*. https://doi.org/10.1007/978-3-319-13960-9_1

[41] Geoff Langdale and Daniel Lemire. 2019. Parsing gigabytes of JSON per second. *The VLDB Journal* 28, 6 (2019). https://doi.org/10.1007/s00778-019-00578-5

[42] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. https://doi.org/10.1145/2588555.2610507

[43] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *Proceedings of the International Conference on Data Engineering (ICDE)*. https://doi.org/10.1109/icde.2018.00026

[44] Daniel Lemire and Wojciech Mula. 2022. Transcoding billions of Unicode characters per second with SIMD instructions. *Software: Practice and Experience* 52, 2 (2022). https://doi.org/10.1002/spe.3036

[45] Daniel Lemire and Christoph Rupp. 2017. Upscaledb: Efficient integer-key compression in a key-value store using SIMD instructions. *Information Systems* 66 (2017). https://doi.org/10.1016/j.is.2017.01.002

[46] Linux Kernel Development Community. 2022. Linux Kernel Guide: Transparent Hugepage Support. https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html

[47] Mathieu Lobet, Francesco Massimo, Arnaud Beck, Guillaume Bouchard, Frederic Perez, Tommaso Vinci, and Mickael Grech. 2022. Simple Adaptations to Speed-up the Particle-In-Cell Code Smilei on the ARM-Based Fujitsu A64FX Processor. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (Workshops)*. https://doi.org/10.1145/3503470.3503475

[48] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: scalable hashing on persistent memory. *Proceedings of the VLDB Endowment* 13, 8 (2020). https://doi.org/10.14778/3389133.3389134

[49] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. https://doi.org/10.1145/3318464.3389705

[50] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. https://doi.org/10.1145/3514221.3517911

[51] Tobias Maltenberger, Ivan Ilic, Ilin Tolovski, and Tilmann Rabl. 2022. Evaluating Multi-GPU Sorting with Modern Interconnects. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. https://doi.org/10.1145/3514221.3517842

[52] Tobias Maltenberger, Till Lehmann, Lawrence Benson, and Tilmann Rabl. 2022. Evaluating In-Memory Hash Joins on Persistent Memory. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. https://doi.org/10.48786/EDBT.2022.23

[53] Bernard Marr. 2018. How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read. https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read

[54] Satoshi Matsuoka. 2021. Fugaku and A64FX: the First Exascale Supercomputer and its Innovative Arm CPU. In *Proceedings of the Symposium on VLSI Circuits*. https://doi.org/10.23919/vlsicircuits52068.2021.9492415

[55] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed operator fusion for in-memory databases: making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment* 11, 1 (2017). https://doi.org/10.14778/3151113.3151114

[56] Timothy Prickett Morgan. 2022. Nvidia embraces the CPU world with "GRACE" ARM server chip. https://www.nextplatform.com/2022/03/25/nvidias-grace-arm-server-chip-is-a-game-changer/

[57] Wojciech Mula and Daniel Lemire. 2020. Base64 encoding and decoding at almost the speed of a memory copy. *Software: Practice and Experience* 50, 2 (2020). https://doi.org/10.1002/spe.2777

[58] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[59] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. https://doi.org/10.1145/2723372.2749436

[60] Ryohei Okazaki, Takekazu Tabata, Sota Sakashita, Kenichi Kitamura, Noriko Takagi, Hideki Sakata, Takeshi Ishibashi, Takeo Nakamura, and Yuichiro Ajima. 2020. Supercomputer Fugaku CPU A64FX Realizing High Performance, High-Density Packaging, and Low Power Consumption. In *Fujitsu Technical Review*.

[61] Oracle. 2017. Database Administrator's Reference: HugePages. https://docs.oracle.com/database/121/UNXAR/appi_vlm.htm#UNXAR391

[62] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004). https://doi.org/10.1016/j.jalgor.2003.12.002

[63] Johannes Pietrzyk, Annett Ungethüm, Dirk Habich, and Wolfgang Lehner. 2019. Fighting the Duplicates in Hashing: Conflict Detection-aware Vectorization of Linear Probing. In *Proceedings of Datenbanksysteme für Business, Technologie und Web (BTW)*. https://doi.org/10.18420/btw2019-04

[64] Constantin Pohl, Kai-Uwe Sattler, and Goetz Graefe. 2019. Joins on high-bandwidth memory: a new level in the memory hierarchy. *The VLDB Journal* 29, 2-3 (2019). https://doi.org/10.1007/s00778-019-00546-z

[65] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. https://doi.org/10.1145/2723372.2747645

[66] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proceedings of the VLDB Endowment* 9, 3 (2015). https://doi.org/10.14778/2850583.2850585

[67] John Russel. 2021. IBM Introduces its First Power10-based Server, the Power E1080; Targets Hybrid Cloud. https://www.hpcwire.com/2021/09/08/ibm-introduces-power10-based-server-the-power-e1080-targets-hybrid-cloud/

[68] David Schor. 2021. Arm Launches ARMv9. https://fuse.wikichip.org/news/4646/arm-launches-armv9/

[69] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. https://doi.org/10.1145/2882903.2882917

[70] Agam Shah. 2021. Apple is beginning to undo decades of Intel, x86 dominance in PC market. https://www.theregister.com/2021/11/12/apple_arm_m1_intel_x86_market/

[71] Dipti Shankar, Xiaoyi Lu, and Dhabaleswar K. DK Panda. 2019. SimdHT-Bench: Characterizing SIMD-Aware Hash Table Designs on Emerging CPU Architectures. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*. https://doi.org/10.1109/iiswc47752.2019.9042069

[72] Steve Sibley. 2018. POWER9 Scale-Up servers designed to fuel innovation. https://www.ibm.com/blogs/systems/ibm-power9-enterprise-servers/

[73] SSE2NEON Contributors. 2022. SSE2NEON Movemask Implementation. https://github.com/DLTcollab/sse2neon/blob/dcf6a2a802986341939fc2e1e02bf33702633d63/sse2neon.h/#L4752

[74] Nigel Stephens. 2016. Technology Update: Scalable Vector Extension (SVE) for Armv8-A. https://community.arm.com/arm-community-blogs/b/high-performance-computing-blog/posts/technology-update-the-scalable-vector-extension-sve-for-the-armv8-a-architecture

[75] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. 2017. The ARM Scalable Vector Extension. *IEEE Micro* 37, 2 (2017). https://doi.org/10.1109/mm.2017.35

[76] The TOP500 Project. 2022. TOP500 List: November 2022. https://www.top500.org/lists/top500/2022/11/

[77] Georgios Theodorakis, Alexandros Koliousis, Peter Pietzuch, and Holger Pirk. 2020. LightSaber: Efficient Window Aggregation on Multi-core Processors. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. https://doi.org/10.1145/3318464.3389753

[78] Trendforce. 2022. ARM-based Server Penetration Rate to Reach 22 % by 2025 with Cloud Data Centers Leading the Way, Says TrendForce. https://www.trendforce.com/presscenter/news/19700101-11178.html

[79] Mark Tyson. 2022. Ampere's Altra Max 80 Core Arm CPU Gets Benchmarked, Delidded, Measured. https://www.tomshardware.com/news/ampere-altra-max-80-ccore-arm-delidded

[80] Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Alexander Krause, Dirk Habich, Wolfgang Lehner, and Erich Focht. 2020. Hardware-Oblivious SIMD Parallelism for In-Memory Column-Stores. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[81] Reini Urban. 2021. SMHasher 128-bit Multiply Shift. https://github.com/rurban/smhasher/blob/28de33a868763a00439a5fc408b56b20f2d86f7c/Hashes.cpp#L906

[82] xSIMD Contributors. 2022. xSIMD Github Repository. https://github.com/xtensor-stack/xsimd

[83] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. Analyzing efficient stream processing on modern hardware. *Proceedings of the VLDB Endowment* 12, 5 (2019). https://doi.org/10.14778/3303753.3303758